DORA MARÍA BALLESTEROS

INVESTIGACIÓN EN CIENCIA DE DATOS.

UN LIBRO QUE ENSEÑA A ESCUCHAR LOS DATOS ANTES DE MODELARLOS

INVESTIGACIÓN EDUCATIVA & PEDAGÓGICA IBEROAMERICANA

editorial redipe

© 2026

Título original:

Libro de investigación.

INVESTIGACIÓN EN CIENCIA DE DATOS.
UN LIBRO QUE ENSEÑA A ESCUCHAR LOS DATOS ANTES DE MODELARLOS
.

Autor: DORA MARÍA BALLESTEROS

ISBN: 978-1-957395-63-0
Primera edición: FEBRERO DE 2026

SELLO Editorial
Editorial REDIPE (95857440), Nueva York – Cali
Red de Pedagogía S.A.S. NIT: 900460139-2
© de la ilustración de la cubierta

Comité Editorial:

Valdir Heitor Barzotto, Universidad de Sao Paulo, Brasil
Carlos Arboleda A. PhD Investigador Southern Connecticut State University, Estados Unidos
Agustín de La Herrán Gascón, Ph D. Universidad Autónoma de Madrid, España
Mario Germán Gil Claros, Grupo de Investigación Redipe
Rodrigo Ruay Garcés, Chile. Coordinador Macroproyecto Investigativo Iberoamericano EvaluaciónEducativa
Julio César Arboleda, Ph D. Dirección General Redipe. Grupo de investigación Educación y Desarrollo humano, Universidad de San Buenaventura

Queda prohibida, salvo excepción prevista en la ley, la reproducción (electrónica, química, mecánica, óptica, de grabación o de fotocopia), distribución, comunicación pública y transformación de cualquier parte de ésta publicación -incluído el diseño de la cubierta- sin la previa autorización escrita de los titulares de la propiedad intelectual y de la Editorial. La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual. Los Editores no se pronuncian, ni expresan ni implícitamente, respecto a la exactitud de la información contenida en este libro, razón por la cual no puede asumir ningún tipo de responsabilidad en caso de error u omisión.

Red Iberoamericana de Pedaogía
editorial@rediberoamericanadepedagogia.com
www.redipe.org

Impreso en Cali, Colombia
Printed in Cali, Colombia

INVESTIGACIÓN EN CIENCIA DE DATOS

Un libro que enseña a escuchar los datos antes de modelarlos

DORA MARIA BALLESTEROS

2026

SOBRE EL AUTOR

Dora María Ballesteros es profesora e investigadora de la Universidad Militar Nueva Granada (UMNG) desde 2007. Ingeniera Electrónica con Maestría y Doctorado en Ingeniería Electrónica, ha dedicado más de veinte años al estudio del procesamiento de señales y, en la última década, a la investigación en inteligencia artificial y ciencia de datos.

A lo largo de su carrera ha publicado cuatro libros y alrededor de 90 artículos científicos en revistas nacionales e internacionales. Su trabajo reciente se ha caracterizado por integrar teoría, experimentación y aplicaciones reales, con un interés particular en el desarrollo de herramientas basadas en inteligencia artificial que contribuyan a mitigar el uso no ético de contenido *deepfake*, un desafío tecnológico y social de creciente relevancia.

Este libro surge de esa combinación de experiencia, investigación y práctica acumulada durante años de trabajo en proyectos dentro de la UMNG. Su propósito es ofrecer una visión clara, estructurada y aplicada de la ingeniería y la ciencia de datos, mostrando al lector cómo estos conceptos se materializan en problemas reales y en soluciones que impulsan la innovación y la comprensión en un mundo cada vez más guiado por los datos.

AGRADECIMIENTO

Gracias, Alejo. Sin ti, este sueño de escribir un libro de Ciencia de Datos no habría sido posible.

A todos mis estudiantes de pregrado y posgrado, que durante años han confiado en mí para acompañarlos en su proceso formativo en investigación: su apoyo, compromiso y curiosidad han sido fundamentales en mi consolidación como investigadora.

PRÓLOGO

En el trabajo investigativo, los datos no son simples números ni registros aislados: son señales de un mundo que intenta decirnos algo. Aprender a trabajar con ellos implica, ante todo, aprender a escuchar. En ese proceso, el rol del ingeniero, del científico y del analista de datos deja de ser una tarea mecánica para convertirse en un ejercicio de interpretación y sentido. El ingeniero prepara el terreno; el científico formula preguntas, plantea hipótesis y busca comprender el *porqué* detrás de los fenómenos; y el analista transforma esas señales en comprensión, en historias que los datos, por sí solos, no podrían contar. En ese camino, el análisis exploratorio de datos y la ingeniería de características dejan de ser simples pasos técnicos para convertirse en momentos de descubrimiento, instantes en los que el dato comienza a hablar y la ciencia de datos se revela como un acto creativo, donde convergen intuición, método y curiosidad.

Las ideas que aquí se presentan se apoyan en mi tesis doctoral, *"Métodos de comunicación encubierta de voz basados en un principio bio-inspirado"*, y en diversos proyectos de investigación desarrollados en la Universidad Militar Nueva Granada, entre ellos *"A video forensic solution for integrity assurance, object recognition and tampering detection: Phase I"* y *"FakeVoiceFinder: Sistema de identificación de voz clonada para mitigar los riesgos del mal uso de la inteligencia artificial"*. Más que una recopilación de resultados, este libro es una invitación a escuchar a los datos y a construir conocimiento con rigor y sentido.

PAG

INTRODUCCIÓN... 15

CAPÍTULO 1.

INTRODUCCIÓN A LA CIENCIA DE DATOS E INGENIERÍA DE DATOS.. 19

1.1. Fundamentos y ciclo de vida de los datos.......... 20
1.2. Ingeniero de datos vs. Científico de datos vs. Analista de datos.. 22
1.2.1. Límites y sinergias entre roles............................ 24
1.2.2. Sobre la transformación y el FE.......................... 24
1.2.3. Aplicación práctica en investigaciones del autor.. 25
1.3. Machine Learning: problemas, métricas y validación.. 26
1.3.1. Métricas de clasificación y regresión.................. 28
1.3.2. Métricas para clasificación multiclase................. 30
1.3.3. Importancia de las métricas en la interpretación de resultados.. 32
1.4. Model-centric vs. Data-centric............................ 34
1.4.1. Model-centric.. 34
1.4.2. Data-centric... 37
1.4.3. Enfoque híbrido.. 38

CAPÍTULO 2.

TIPOS DE DATOS E INGENIERÍA DE CARACTERÍSTICAS...... 41

2.1. Tipos de datos según su estructura.................... 42
2.1.1. Datos estructurados.. 42
2.1.2. Datos semiestructurados.................................... 43
2.1.3. Datos no estructurados....................................... 44

PAG

2.2. Niveles de los datos... 45
2.2.1. Nivel Nominal.. 45
2.2.2. Nivel Ordinal... 46
2.2.3. Nivel Intervalo... 47
2.2.4. Escala de razón.. 47
2.3. Ingeniería de Características............................ 49
2.3.1. Mejoramiento de Características (Feature Improvement).. 49
2.3.2. Construcción de Características (Feature Construction) 51
2.3.3. Selección de Características............................ 51
2.3.4. Extracción de Características........................... 52
2.3.5. Aprendizaje Automático de Características....... 53

CAPÍTULO 3.

DATOS ESTRUCTURADOS.. 55
3.1. Caso de estudio 1: Análisis Exploratorio dataset COVID-19 de Bogotá.. 55
3.1.1. Pre-visualización del dataset.............................. 56
3.1.2. Tipo de datos en el dataset................................ 58
3.1.3. Distribución de los estados reportados del dataset.. 58
3.1.4. Distribución de los estados reportados vs. Localidad.. 60
3.1.5. Conversión de la fecha de diagnóstico............... 61
3.1.6. Distribución temporal de los estados.................. 64
3.1.7. Casos de contagio por Género........................... 66
3.1.8. Casos de contagio por Día de Diagnóstico.......... 68
3.1.9. Evolución acumulada de casos de COVID-19 en Bogotá... 70
3.1.10. Distribución de casos de COVID-19 por edad.. 72
3.1.11. Distribución de fallecidos por COVID-19 según edad... 75

PAG

3.1.12. Casos de contagio por Localidad, Fallecidos por Localidad, y Tasa de muerte por Localidad......................... 78
3.1.13. Tasa de muerte por Localidad........................... 82
3.2. Caso de estudio 2: EDA y FE al Bank Churn dataset... 85
3.2.1. Análisis Exploratorio de Datos Bank churn dataset 88
3.2.2. Transformación de columnas del dataset: de object a integer... 90
3.2.3. Imputación de datos al interior de una columna del dataframe.. 95
3.2.4. Eliminar columnas innecesarias del dataset........ 97
3.2.5. Modelamiento... 99

CAPÍTULO 4.

SERIES DE TIEMPO... 108
4.1. Conceptos Básicos de Series de Tiempo............ 108
4.1.1. Componentes fundamentales de una Serie de Tiempo... 108
4.1.2. Clasificación de Series de Tiempo..................... 110
4.1.3. Autocorrelación y memoria temporal (en series financieras no estacionarias)........................ 111
4.1.4. Diferencias entre el análisis exploratorio clásico y el EDA en series de tiempo.. 113
4.2. Ingeniería de características en Series de Tiempo... 115
4.2.1. Características con retardos (Lag Features)........ 116
4.2.2. Ventanas deslizantes (Rolling Window Features) 116
4.2.3. Ventanas expansivas . (Expanding Window Features) 117
4.3. Caso de Estudio 1 de Series de Tiempo: clima.. 117
4.3.1. EDA en el dataset Clima..................................... 120
4.3.2. Modelamiento del dataset Clima (baseline)......... 126

PAG

4.3.3. Ingeniería de Características aplicada al dataset de Clima.. 131
4.4. Caso de Estudio 2 de Series de Tiempo: Twlo prices.. 139
4.4.1. Análisis Exploratorio y construcción de la variable de salida.. 140
4.4.2. Modelamiento en twlo_prices dataset antes de FE.. 149
4.4.3. Ingeniería de características con selección estadística.. 153
4.4.4. Entrenamiento del modelo y predicción con FE.. 162
4.5. Conclusiones de cierre del capitulo................... 165

CAPÍTULO 5.

AUDIO SINTÉTICO Y DEEPFAKES DE VOZ: DE LA INVESTIGACIÓN DOCTORAL A LAS HERRAMIENTAS DE DETECCIÓN.. 167
5.1. Método de Imitación... 171
5.1.1. Código de Imitación... 172
5.1.2. Código de Recuperación (reverse)..................... 177
5.2. Deep4SNet: identificación de audio sintético...... 179
5.3. FakeVoiceFinder: una librería para identificar audio sintético.. 181
5.3.1. Datasets: Fake Audio Dataset (ElevenLabs & Respeecher) y TTS/V2V Audio Deepfake Dataset... 183
5.3.2. Experimento básico dentro de FakeVoiceFinder.. 184
5.3.3. Inferencia dentro de FakeVoiceFinder................. 202
5.1.1. Cierre del capítulo... 213

CAPÍTULO 6.

CONCLUSIONES Y REFLEXIONES DE CIERRE....................... 215

INTRODUCCIÓN

Este libro nació de una motivación personal y académica: la convicción de que un ingeniero que entienda, transforme y otorgue valor agregado a los datos está mejor preparado que aquel que se limita al entrenamiento de modelos. Desde que inicié mis primeros trabajos en *machine learning* y ciencia de datos, he podido presenciar la rápida evolución del campo y, al mismo tiempo, los vacíos conceptuales que aún persisten en la formación de muchos profesionales. Con frecuencia, los estudiantes se maravillan con los resultados de los modelos de aprendizaje automático, pero desconocen las estructuras invisibles que los hacen posibles: la ingeniería de datos detrás de cada tipo de información, su calidad, su trazabilidad y su significado.

Fue precisamente desde esa observación, y desde más de una década de trabajo académico e investigativo en procesamiento de señales, visión por computador y detección de audio sintético, que surgió la idea de este libro. Su propósito es guiar a los futuros ingenieros hacia una comprensión integral del ciclo de vida de la ciencia de datos, combinando el rigor técnico con la curiosidad científica. No se trata solo de aprender librerías o ejecutar modelos, sino de pensar con datos, de cuestionarlos, de entender su contexto y su propósito.

El enfoque de este libro se enmarca en la metodología de Aprendizaje Basado en Proyectos (ABP), lo que significa que cada capítulo propone retos y proyectos que conectan la teoría con la práctica investigativa. De esta manera, el aula se convierte en un laboratorio de exploración, donde el estudiante no solo reproduce experimentos, sino que diseña los suyos, analiza resultados, documenta hallazgos y aprende a comunicar evidencia científica.

A lo largo de los capítulos, el lector recorrerá un camino progresivo que inicia con los fundamentos conceptuales de la ciencia e ingeniería de datos y culmina con un caso aplicado de alto impacto: la detección de audio sintético mediante modelos de aprendizaje profundo. El primer y segundo capítulo introduce los pilares teóricos de la ciencia de datos, la diferencia entre los

enfoques *model-centric* y *data-centric*, y la importancia de la reproducibilidad y la evaluación ética. El tercer capítulo profundiza en la ingeniería de datos estructurados y la construcción de *pipelines* reproducibles, con ejemplos reales de optimización y compresión de modelos. El cuarto explora las series de tiempo y el procesamiento de señales, mostrando cómo los datos pueden narrar comportamientos, por ejemplo, en el sector bancario. Finalmente, el quinto capítulo integra todas las dimensiones anteriores en un proyecto de detección de voz sintética, donde confluyen la ingeniería de datos, el procesamiento espectral y la ética en la era de los *deepfakes*.

Los temas y experimentos de este libro se nutren de investigaciones publicadas entre 2019 y 2025, fruto de colaboraciones científicas en torno a la eficiencia de modelos de aprendizaje profundo, la interpretabilidad de las redes neuronales y la detección de falsificaciones de voz e imagen. Trabajos como *Deep4SNet* y *FlexiPrune*, sirven de base experimental y metodológica para los capítulos, aportando ejemplos reales de cómo la ingeniería de datos se convierte en una herramienta de investigación aplicada. Esta integración permite al lector no solo estudiar los conceptos, sino entender cómo se construyen y validan en la práctica científica.

En el contexto profesional, la ingeniería de datos se ha consolidado como una de las competencias más valoradas en sectores como las telecomunicaciones, multimedia, la banca, la seguridad digital y el Internet de las Cosas. Dominar el ciclo de vida del dato, desde su captura hasta su modelado, permite construir sistemas de inteligencia artificial eficientes, confiables y éticamente responsables. Por ello, este libro también es una invitación a pensar la ingeniería de datos no solo como una disciplina técnica, sino como una práctica con responsabilidad social.

El ecosistema tecnológico que lo acompaña incluye las principales librerías de Python empleadas en ciencia e ingeniería de datos, como *pandas*, *numpy*, *scikit-learn*, *tensorflow*, *torch*, *librosa*, *torchaudio*, entre otras. Cada una se presenta no como una herramienta aislada, sino como parte de una arquitectura integral de trabajo que el lector aprenderá a ensamblar, documentar y optimizar. Los ejemplos se basan en conjuntos de datos reales provenientes tanto de entornos industriales como de investigación académica, lo que facilita la conexión entre teoría y práctica.

Al finalizar este recorrido, el lector no solo habrá adquirido habilidades técnicas, sino una forma distinta de mirar los datos: con rigor, con curiosidad y con conciencia. Comprenderá que cada modelo de aprendizaje automático es tan sólido como el conjunto de decisiones que lo preceden, y que, en esa cadena, desde la adquisición hasta la inferencia, reside el verdadero valor de la ingeniería. En última instancia, este libro es una invitación a explorar, a cuestionar y a crear, con mente abierta, manos en el código y espíritu de investigador.

INTRODUCCIÓN A LA CIENCIA DE DATOS E INGENIERÍA DE DATOS

En la última década hemos sido testigos de un crecimiento vertiginoso en el campo del aprendizaje automático. Los modelos han aprendido a reconocer rostros, traducir idiomas, detectar fraudes, anticipar comportamientos y hasta imitar la voz humana con una precisión que, hace unos años, parecía impensable. Sin embargo, detrás de esa aparente perfección tecnológica se esconde una verdad que muchos investigadores, entre ellos *Andrew Ng*, han resaltado con claridad: el verdadero éxito de un modelo no depende únicamente de su arquitectura o de cuántas capas tenga, sino de la calidad de los datos con los que se alimenta.

Y es que los datos son mucho más que números o registros: son historias, contextos y decisiones humanas convertidas en información. Cuando esos datos son incompletos, sesgados o mal comprendidos, los modelos aprenden una versión distorsionada de la realidad. Por eso, en los últimos años, la comunidad científica ha comenzado a mirar más allá del modelo y a preguntarse por el corazón mismo de todo sistema inteligente: los datos.

De esa reflexión surge la idea de que no basta con diseñar algoritmos cada vez más potentes; es necesario comprender el origen de los datos, su estructura, su limpieza y su significado. Esa es, precisamente, la tarea del ingeniero de datos: construir los caminos por los que la información viaja, cuidando que lleguen limpias, ordenadas y listas para convertirse en conocimiento.

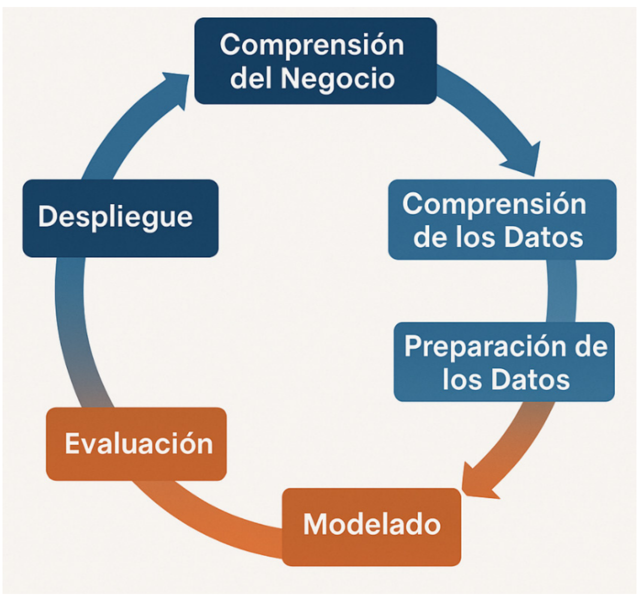
Este capítulo busca invitarte a mirar la ciencia de datos con ojos nuevos. A no verla solo como un conjunto de técnicas, sino como una manera de pensar el

mundo a través de la información. Aquí hablaremos del ciclo de vida del dato, del papel del ingeniero que lo acompaña desde su captura hasta su transformación, y de los dos grandes paradigmas que hoy guían la inteligencia artificial: el *model-centric* y el *data-centric*. Más que definiciones, encontrarás aquí una conversación: una oportunidad para entender por qué los datos importan tanto como, o incluso más que, los modelos que intentan aprender de ellos.

1.1. **Fundamentos y ciclo de vida de los datos**

Una de las metodologías más utilizadas por los científicos y los ingenieros de datos se denomina CRISP-DM (*Cross Industry Standard Process for Data Mining*). Fue desarrollada hace más de veinte años, pero sigue siendo plenamente vigente como marco de trabajo para proyectos de ciencia de datos. Más que un conjunto de pasos rígidos, CRISP-DM propone un ciclo flexible y reflexivo: comprender el problema, conocer los datos, prepararlos, modelar, evaluar y desplegar. Cada fase representa un momento en la vida del dato, una etapa donde la ingeniería, la estadística y la interpretación se encuentran. Estas fases se presentan en la Figura 1.

Figura 1. Ciclo de vida de los datos: Metodología CRISP-DM. Generada con ChatGPT por el autor.



Todo comienza con la **comprensión del problema o del contexto**. Antes de escribir una sola línea de código, el equipo debe entender cuál es el propósito de los datos, es decir, qué se pretende predecir u obtener con su uso. Esta fase da sentido al proyecto: define el objetivo y permite que las siguientes decisiones técnicas mantengan coherencia con el propósito inicial.

Posteriormente llega la **comprensión de los datos** (*data understanding*). Es el momento de realizar un **EDA (Exploratory Data Analysis)**, que consiste en conocer su comportamiento estadístico, su relación con otras características (*features*) y con la salida (si existe). Además, en esta etapa se realizan tareas esenciales como la limpieza de datos, que permite detectar valores faltantes, eliminar duplicados y corregir errores que podrían afectar el modelado posterior.

A continuación, se desarrolla la fase de **preparación de los datos**, en donde la "magia" de los ingenieros y científicos de datos cobra vida. Se corrigen errores, se unifican formatos y se eliminan inconsistencias, pero también se mejoran y construyen nuevas variables mediante el ***feature Engineering*** **(FE)**, un proceso que busca crear o seleccionar las características más representativas para que el modelo aprenda de forma más efectiva. En esencia, esta etapa convierte los datos crudos en información útil, lista para el análisis, asegurando que lo que llegue al modelo refleje con fidelidad la realidad que se intenta comprender.

Con los datos listos, se pasa a la fase de **modelado**, donde entran en juego las técnicas de aprendizaje automático y aprendizaje profundo. Se prueban distintos algoritmos, se ajustan hiperparámetros y se comparan resultados. Pero aquí aparece una verdad importante: el modelado no es el final del proceso, sino una consecuencia de las decisiones tomadas en las fases anteriores. Un modelo solo puede aprender aquello que los datos le permiten aprender.

Después llega la **evaluación**, un momento de análisis crítico. No basta con obtener una buena métrica; es necesario revisar si el modelo responde a la pregunta original, si los datos fueron suficientes o si existen sesgos que puedan distorsionar los resultados. Esta etapa tiene un carácter científico y ético: invita a detenerse, revisar y validar antes de avanzar.

Finalmente, el ciclo culmina con la **implementación o despliegue**, cuando los resultados del proyecto se integran en un entorno real o se presentan

como parte de una investigación aplicada. Pero el ciclo no termina allí. CRISP-DM propone un retorno constante a las fases anteriores, una mejora continua que reconoce que los datos, como la realidad que representan, están siempre en movimiento.

1.2. Ingeniero de datos vs. Científico de datos vs. Analista de datos

En el ecosistema contemporáneo de la ciencia de datos, tres perfiles articulan el ciclo completo de generación de conocimiento: el ingeniero de datos, el científico de datos y el analista de datos. Aunque sus funciones a veces se superponen, cada uno representa una mirada distinta sobre la misma materia prima: el dato. Comprender esas diferencias permite construir equipos más sólidos y sistemas más confiables.

El **ingeniero de datos** diseña y mantiene la infraestructura que permite que los datos fluyan. Se encarga de construir pipelines, integrar fuentes, garantizar la calidad y disponibilidad de la información.

El **científico de datos** transforma esos datos en conocimiento. Explora, limpia, crea *features*, entrena modelos y evalúa resultados.

El **analista de datos**, finalmente, convierte ese conocimiento en comprensión y acción. Interpreta resultados, comunica hallazgos y los traduce en decisiones.

En la Tabla 1 se presenta una comparación entre los tres roles descritos anteriormente.

Tabla 1. Tabla comparativa entre Ingeniero de datos, Científico de Datos y Analista de datos.

Criterio	Ingeniero de Datos	Científico de Datos	Analista de Datos
Enfoque principal	Infraestructura y flujo del dato	Modelado, experimentación y descubrimiento de patrones	Interpretación y comunicación de resultados
Responsabilidades clave	Integración de fuentes, construcción de pipelines, aseguramiento de calidad, almacenamiento y orquestación	Limpieza analítica, *EDA*, *feature engineering*, entrenamiento y validación de modelos, evaluación de métricas	Análisis descriptivo, consultas, visualización, elaboración de reportes y *dashboards*
Competencias técnicas	SQL, Python, Spark, Kafka, Airflow, DVC, bases de datos, arquitecturas distribuidas	Python, R, pandas, scikit-learn, TensorFlow, PyTorch, estadística, machine learning	SQL, Excel, Power BI, Tableau, herramientas de visualización y *storytelling*
Tipo de limpieza de datos	Limpieza estructural (formatos, duplicados, integridad)	Limpieza analítica (*outliers*, sesgos, coherencia estadística)	Validación de coherencia y consistencia visual
Participación en el EDA	Soporte en la estructuración de datos para análisis	Ejecución principal: análisis exploratorio, correlaciones, visualización y generación de hipótesis	Interpretación de resultados exploratorios
Participación en FE	Implementa transformaciones recurrentes o automatizadas en pipelines	Diseña, crea y evalúa las características más relevantes para el modelo	Apoya en la interpretación y documentación de *features*

Criterio	Ingeniero de Datos	Científico de Datos	Analista de Datos
Herramientas y entornos	Airflow, Prefect, Kafka, DVC, bases SQL/NoSQL, sistemas cloud	Jupyter, Python, R, frameworks de ML, Git, MLflow	Power BI, Tableau, Google Data Studio, hojas de cálculo
Objetivo final	Garantizar datos confiables, accesibles y reproducibles	Extraer conocimiento útil y desarrollar modelos predictivos o explicativos	Traducir resultados técnicos en información clara para la toma de decisiones

1.2.1. **Límites y sinergias entre roles**

En la práctica, las fronteras entre estos tres perfiles son débiles. En entornos de investigación o *startups*, un mismo profesional puede desempeñar varios de estos papeles. Sin embargo, la distinción conceptual sigue siendo útil:

- El ingeniero asegura la existencia y estructura del dato.
- El científico le da sentido y valor predictivo.
- El analista le otorga contexto y aplicabilidad.

Estas tres perspectivas conforman una secuencia que une la ingeniería, la ciencia y la interpretación. Cuando los límites se entienden como zonas de colaboración, no de división, el flujo del dato se convierte en un proceso integral de conocimiento.

1.2.2. **Sobre la transformación y el FE**

A menudo se confunde la *T* del proceso ETL (Extract, Transform, Load) con el Feature Engineering (FE).

Aunque ambos implican transformar datos, su propósito es distinto:

- En ETL, la transformación busca usabilidad operacional: estandarizar, limpiar o estructurar datos para su almacenamiento o consulta.

- En Feature Engineering, la transformación busca utilidad analítica o predictiva: crear variables que representen mejor los fenómenos o mejoren el aprendizaje del modelo.

Por tanto, el *feature engineering* puede verse como una extensión inteligente de la ingeniería de datos, donde la técnica se combina con la intuición científica. Su ubicación natural está en la frontera entre ambos mundos: depende de la infraestructura creada por el ingeniero, pero responde a la curiosidad analítica del científico.

1.2.3. **Aplicación práctica en investigaciones del autor**

La visión que articula este libro no surge únicamente del estudio teórico del ciclo de vida de los datos, sino también de una trayectoria investigativa que ha integrado, en la práctica, los tres grandes roles del ecosistema analítico: el ingeniero, el científico y el analista de datos. Los proyectos desarrollados en los últimos años, constituyen ejemplos concretos de cómo la ingeniería de datos se expande hacia lo científico y lo interpretativo, formando un continuo de conocimiento aplicado.

En el contexto investigativo, la ingeniería de datos abarca no solo la integración y transformación de información, sino también la construcción deliberada de datasets. Esta tarea implica diseñar colecciones de datos representativas, documentadas y reproducibles, donde cada decisión, desde la selección de fuentes hasta el formato de almacenamiento, constituye un acto de ingeniería. En proyectos como *h-voice* (*A dataset of histograms of original and fake voice recordings*) y *CG-1050* (*A dataset of 1050-tampered color and grayscale images*), la creación de los conjuntos de datos se concibe como un proceso estructurado de diseño y validación que traduce la teoría en evidencia experimental tangible.

En el ámbito de la ciencia de datos, el primer trabajo representativo fue *Deep4SNet*, donde se desarrolló un proceso de *feature engineering* dual: se extrajeron valores estadísticos de los audios (media, varianza, asimetría y curtosis) para alimentar un modelo de *machine learning* tradicional, y se generaron histogramas de las grabaciones para su procesamiento mediante una red neuronal convolucional (*CNN*). Este estudio marcó una transición entre la

caracterización clásica y la representación profunda del audio, evidenciando cómo la forma del dato define la capacidad del modelo para aprender. Posteriormente, *FakeVoiceFinder* consolidó esta línea de investigación al integrar múltiples representaciones espectrales (Mel, log, DWT y CQT) dentro de un marco abierto y reproducible, disponible en https://github.com/DEEP-CGPS/FakeVoiceFinder/tree/main. Este *framework* permitió comparar arquitecturas, medir el impacto de las transformaciones y establecer una base metodológica sólida para la detección de audio sintético, convirtiéndose en un ejemplo de ingeniería de datos aplicada a la investigación científica y a la generación de conocimiento reproducible.

En el nivel analítico, el trabajo *Is My Pruned Model Trustworthy? PE-Score: A New CAM-Based Evaluation Metric* representa una forma avanzada de análisis e interpretación de resultados. Este estudio propuso una métrica visual y cuantitativa basada en *Class Activation Maps* (CAM) para evaluar la coherencia y confiabilidad de los modelos podados, integrando interpretabilidad y rendimiento en un mismo marco evaluativo. De este modo, la labor del analista de datos trasciende la visualización descriptiva y se convierte en una evaluación crítica del comportamiento interno de los modelos, aportando transparencia y explicabilidad al proceso científico.

En conjunto, estas líneas de investigación demuestran que los roles de ingeniero, científico y analista de datos no son compartimentos estancos, sino dimensiones interdependientes de una misma práctica de conocimiento. En el contexto académico, esta convergencia permite que la ingeniería de datos trascienda su definición operacional y se convierta en un método de indagación científica: una manera estructurada, medible y ética de descubrir patrones, validar hipótesis y construir sentido a partir de la información.

1.3. **Machine Learning: problemas, métricas y validación**

El aprendizaje automático (*Machine Learning*) constituye una de las áreas más dinámicas de la ingeniería de datos, orientada a construir modelos capaces de aprender patrones a partir de la información disponible. Su aplicación no se limita a la selección de un algoritmo, sino que comienza con la identificación del tipo de problema que se desea resolver. En términos generales, los modelos de *Machine Learning* pueden agruparse en tres grandes categorías: regresión,

clasificación **y** clustering, cada una con un propósito y una naturaleza de salida diferentes.

En los problemas de **regresión**, el objetivo consiste en predecir valores numéricos continuos a partir de variables de entrada. Estos modelos estiman relaciones funcionales entre las características y la salida, buscando minimizar la diferencia entre el valor real y el predicho. Ejemplos clásicos incluyen la estimación del precio de un bien, la predicción de temperatura o la modelación de una señal en el tiempo. Su salida es, por tanto, un número real que representa una magnitud o tendencia.

En los problemas de **clasificación**, el propósito es asignar a qué clase o categoría pertenece el dato de entrada. El modelo aprende de los *features* de los datos y produce una salida discreta, la cual puede ser binaria o multiclase. En el caso binario tenemos como ejemplos *spam/no spam*, *voz real/voz sintética* o *tumor benigno/maligno*. En casos multiclase, puede predecir a qué animal pertenece una foto, de un grupo finito de opciones.

Por su parte, los problemas de **clustering** pertenecen al ámbito del *aprendizaje no supervisado*, donde el modelo no dispone de etiquetas previas (es decir, los datos no se entregaron con valores de salida tipo regresión o clase) y su tarea consiste en descubrir patrones o agrupaciones naturales dentro de los datos que permita agruparlos. El resultado no es una predicción explícita, sino la asignación de pertenencia a grupos o clústeres que comparten similitudes estructurales. Esta técnica se utiliza, por ejemplo, en la segmentación de clientes, la agrupación de señales o la detección de comportamientos anómalos.

Cada uno de estos tipos de problemas determina no solo el enfoque algorítmico, sino también las métricas de evaluación, los criterios de validación y la interpretación de resultados**.** En la práctica, comprender qué se espera del modelo (ej. un valor continuo, una categoría previamente definida o una estructura oculta) es el primer paso para construir experimentos de *Machine Learning* coherentes, reproducibles y científicamente sólidos.

1.3.1. Métricas de clasificación y regresión

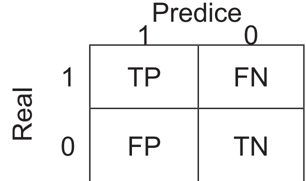
Como se mencionó previamente, el tipo de problema de *Machine Learning* determina las métricas adecuadas para evaluar el rendimiento del modelo. En problemas de regresión, las métricas más comunes son:

- MAE (Mean Absolute Error)
- MSE (Mean Squared Error)
- RMSE (Root Mean Squared Error)
- R^2 (coeficiente de determinación)

Estas permiten cuantificar el error entre los valores reales y los valores predichos.

En problemas de clasificación, las métricas derivan de la matriz de confusión, la cual registra la cantidad de aciertos y errores por clase. Para el caso binario, la matriz se representa así:

Cuando se tiene clasificación binaria, la matriz de confusión es la siguiente:



Donde

- TP (*True Positives*): instancias de la clase 1 correctamente clasificadas.
- TN (*True Negatives*): instancias de la clase 0 correctamente clasificadas.
- FP (*False Positives*): muestras de la clase 0 clasificadas incorrectamente como 1.
- FN (False Negatives): muestras de la clase 1 clasificadas incorrectamente como 0.

A partir de estos valores, se definen las siguientes métricas:

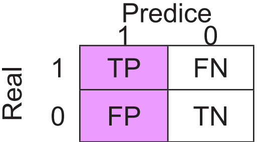
$$accuracy\ (acc) = \frac{TP + TN}{TP + TN + FP + FN}$$

$$Precision\ (P) = \frac{TP}{TP + FP}$$

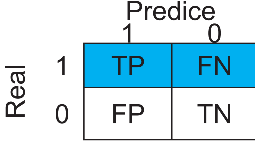
$$Recall\ (R) = \frac{TP}{TP + FN}$$

Es importante notar que *Precision* y *Recall* comparten el mismo numerador (TP), diferenciándose únicamente por el denominador, el cual representa la naturaleza del error que cada métrica penaliza.

Para P, es:



Y para R, es:



De acuerdo con lo anterior:

- *Accuracy* (precisión global): proporción de predicciones correctas sobre el total de muestras.
- *Precision*: fracción de predicciones positivas que realmente son positivas.
- *Recall* (sensibilidad): fracción de positivos reales que el modelo logra identificar.

Adicionalmente, se puede calcular la media armónica entre *P* y *R*, conocida como *F1-score*, así:

$$F1_{score} = 2\,\frac{P * R}{P + R}$$

1.3.2. Métricas para clasificación multiclase

En problemas de clasificación multiclase (con K clases), la matriz de confusión se extiende a una matriz de tamaño K × K, donde cada fila representa la clase real y cada columna la clase predicha. A diferencia del caso binario, los valores TP, FP y FN deben calcularse por clase, aplicando un enfoque uno contra todos (one-vs-all).

Cálculo de métricas por clase (para cada clase *i*):

- TPi: número de muestras de la clase i correctamente clasificadas.
- FNi: muestras de la clase i clasificadas como otra clase.
- FPi: muestras de otras clases clasificadas como clase i.
- TNi: combinaciones correctas restantes.

Las métricas por clase se definen como:

$P_i = TPi\,/\,(TPi + FPi)$

$R_i = TPi\,/\,(TPi + FNi)$

$F1_i = 2 * (P_i * R_i)\,/\,(P_i + R_i)$

Dado que en multiclase se tienen métricas por clase, se requiere un método para obtener un valor agregado global. Los más utilizados son:

a) *Macro-average*

Promedio simple de las métricas por clase.

$$P_macro = \frac{1}{K}\sum_{i=1}^{K} P_i$$

$$R_macro = \frac{1}{K}\sum_{i=1}^{K} R_i$$

$$F1_macro = \frac{1}{K}\sum_{i=1}^{K} F1_i$$

La ventaja de *macro-average* es que da el mismo peso a todas las clases, lo cual es útil cuando todas las clases están balanceadas. Por otra parte, la desventaja es que puede ser engañoso cuando hay clases muy pequeñas, es decir, cuando el problema es desbalanceado.

b) *Micro-average*

En problemas de clasificación con múltiples clases, y especialmente cuando existe desbalance, las métricas *micro-average* permiten evaluar el desempeño global del modelo acumulando los verdaderos positivos, falsos positivos y falsos negativos de todas las clases antes de calcular cada métrica. Esto es especialmente útil cuando el dataset presenta desbalance entre clases, ya que otorga a cada ejemplo la misma importancia y evita que las clases mayoritarias dominen el cálculo.

$$P_micro = (\Sigma TPi) / (\Sigma TPi + \Sigma FPi)$$

$$R_micro = (\Sigma TTPi) / (\Sigma TPi + \Sigma FNi)$$

$$F1_micro = 2 * (P_micro * R_micro) / (P_micro + R_micro)$$

c) *Weighted-average*

El enfoque *weighted-average* calcula las métricas ponderando el desempeño de cada clase según su tamaño relativo dentro del dataset. A diferencia del *macro-average*, que asigna el mismo peso a todas las clases, el *weighted-average* refleja la proporción real de cada categoría. Esto evita que las métricas se distorsionen cuando existen clases muy pequeñas o altamente desbalanceadas.

En conjunto, las métricas *weighted-average* ofrecen una representación más fiel del rendimiento global del modelo cuando el dataset presenta distribuciones desiguales entre clases.

$$P_weighted = \Sigma\left((N_i / N_{total}) \cdot P_i \right)$$

$$R_weighted = \Sigma\left(((N_i / N_{total}) \cdot R_i \right)$$

$$F1_weighted = \Sigma\left((N_i / N_{total}) \cdot F1_i \right)$$

1.3.3. Importancia de las métricas en la interpretación de resultados

Con el siguiente ejemplo ilustraremos la necesidad de seleccionar adecuadamente las métricas al momento de interpretar qué tan bien lo está haciendo un modelo. Partamos del siguiente caso:

Se tiene un problema de clasificación binaria de detección de cáncer de mama. El 90 % de las muestras no tienen cáncer y el 10 % restante sí. Asumimos que la clase 1 corresponde a "cáncer" y que la clase 0 corresponde a "saludable". El modelo de clasificación arrojó la siguiente matriz de confusión:

		Predice	
		1	0
Real	1	2	8
	0	2	88

Si evaluamos el desempeño del clasificador con *acc*, tendríamos que:

$$acc = \frac{2 + 88}{2 + 8 + 2 + 88} = \frac{90}{100} = 0.9$$

Es decir, clasifica correctamente el 90 % de los casos. No obstante, que el modelo se equivoque en la mayoría de los casos de cáncer no es deseable.

Vamos entonces a revisar las métricas P y R:

$$P = \frac{2}{2+2} = \frac{2}{4} = 0.5P = \frac{2}{2+2} = \frac{2}{4} = 0.5, \quad \text{y} \quad R = \frac{2}{2+8} = \frac{2}{10} = 0.2$$

$$R = \frac{2}{2+8} = \frac{2}{10} = 0.2$$

Es decir, tiene una precisión del 50 % y un *recall* del 20 %. Con estos valores, el F1-score es:

$$F1_{score} = 2\,\frac{0.5 * 0.2}{0.5 + 0.2} = 2 * \frac{0.1}{0.7} = 0.29$$

Con el valor de F1 encontrado, es evidente que el modelo no funciona adecuadamente, dado que este valor está muy lejos de 1.

Miremos ahora este otro escenario:

Real \ Predice	1	0
1	7	3
0	7	83

Calculemos de nuevo todas las métricas anteriores:

$$acc = \frac{7 + 83}{100} = \frac{90}{100} = 0.9$$

$$P = \frac{7}{7+7} = \frac{7}{14} = 0.5P = \frac{7}{7+7} = \frac{7}{14} = 0.5, \quad \text{y} \quad R = \frac{7}{7+3} = \frac{7}{10} = 0.7$$

$$R = \frac{7}{7+3} = \frac{7}{10} = 0.7$$

$$F1_{score} = 2\,\frac{0.5 * 0.7}{0.5 + 0.7} = 2 * \frac{0.35}{1.2} = 0.58$$

Este modelo obtuvo la misma *acc* del primer modelo. No obstante, su F1-score es más alto. Adicionalmente, los valores de P y R están más equilibrados que en el primer caso, lo que evidencia un mejor compromiso entre P y R. Por lo que este modelo es muchísimo mejor en el escenario de de detección de cáncer, mejor que el anterior.

1.4. Model-centric vs. Data-centric

Hasta este punto del capítulo hemos recorrido, casi como si siguiéramos el pulso natural de un proyecto real, el ciclo de vida de la ciencia de datos: entendimos el contexto, analizamos el papel de los distintos perfiles técnicos, revisamos los tipos de problemas y conversamos sobre las métricas que permiten decir, con evidencia, qué tan bien se está comportando un modelo.

Ahora, nos detenemos en un cruce de caminos que aparece en casi todos los proyectos, aunque muchas veces pasa desapercibido: la elección entre centrar los esfuerzos en el modelo o centrar los esfuerzos en los datos. No es un dilema absoluto, sino una manera distinta de mirar la misma pregunta: *¿cómo logramos que un sistema funcione mejor?*

Durante años, la comunidad científica se inclinó con fuerza hacia el primer enfoque, el *model-centric*. Era natural: cada nueva arquitectura parecía prometer una revolución, un salto en precisión, un récord nuevo. Sin embargo, en los últimos años ha surgido una reflexión distinta y profundamente necesaria, encabezada por investigadores como Andrew Ng: *no existe modelo que brille si los datos que lo alimentan no son buenos*.

Dicho de otra forma: el desempeño no depende solo del ingenio del modelo, sino de la calidad del mundo que le mostramos a través de los datos.

Por eso, en esta sección vamos a explorar ambos enfoques, no como teorías aisladas, sino como dos lentes que nos permiten interpretar y mejorar un mismo sistema.

1.4.1. Model-centric

En el ciclo de vida que discutimos en la Sección 1.1, la etapa de modelado aparece casi al final. Y aun así (o quizás precisamente debido a eso) ha sido la etapa que más atención ha recibido históricamente. Aquí es donde los investigadores han propuesto arquitecturas nuevas, con capas que se conectan de maneras inesperadas, de redes capaces de “captar” patrones que antes parecían invisibles.

Por lo cual, cuando hablamos del enfoque centrado en el modelo, nos referimos a *ajustar sus hiperparámetros*, que típicamente se clasifican en:

- **Hiperparámetros de arquitectura:** incluye aumentar o disminuir la profundidad de la red, modificar la cantidad de filtros en una CNN, cambiar cómo se conectan las capas (secuencial, residual, paralela), o incluso probar familias completamente distintas como Transformers, ResNets o ConvNeXt. Cada una tiene una forma particular de interpretar la información, casi como si cada arquitectura *leyera el mismo texto* con un acento diferente.

- **Hiperparámetros de entrenamiento:** corresponden a cambios más sutiles, pero no menos poderosos como la tasa de aprendizaje, el optimizador, el número de épocas o el tamaño del *batch* (lote). También decidimos si el modelo debe aprender desde cero (como quien llega por primera vez a un terreno desconocido), o si utilizamos *transfer learning*, es decir, transferir conocimientos previos (en este caso los pesos) aprendidos en otro dataset.

La fortaleza del enfoque *model-centric* está en su disciplina comparativa: mantener quieta la entrada (los datos) y modificar únicamente la arquitectura y los hiperparámetros para medir cuánto cambian las métricas del modelo. Con este enfoque respondemos a la pregunta:

¿qué modelo es mejor para este problema, bajo estas condiciones?

Un ejemplo concreto de model-centric se encuentra en uno de los artículos de mi co-autoría titulado *Influence of Hyperparameters in Deep Learning Models for Coffee Rust Detection”* (Ver Figura 2).

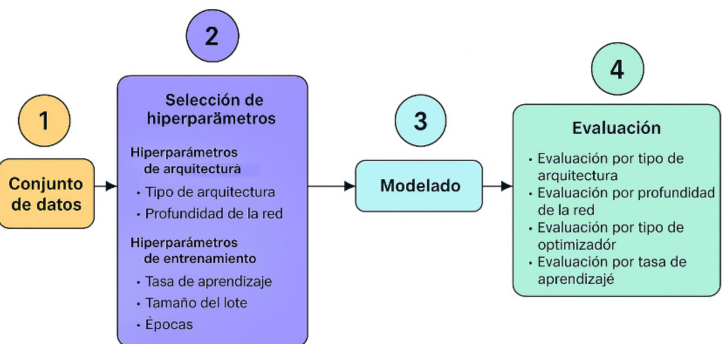


Figura 2. Ciclo de vida proyecto Coffee Rust Detection. Generada con ChatGPT por el autor.

Precisamente se evaluó el impacto de diferentes hiperparámetros como la profundidad de la red, el número de filtros, el tamaño de batch y la tasa de aprendizaje, en el desempeño del modelo al detectar roya del café. Fue un ejercicio casi quirúrgico: mantener constantes las imágenes y su preprocesamiento, y centrarnos en identificar qué combinaciones arquitectónicas y de entrenamiento producían la mayor capacidad de generalización. Ese estudio mostró, en un contexto agrícola real, cómo pequeñas variaciones en los hiperparámetros pueden transformar por completo la capacidad de un modelo para detectar enfermedades en cultivos. Por ejemplo, con la arquitectura DenseNet201 los valores de F1-score llegaron a ser un 40% mejores que los obtenidos con ResNet50, o que el optimizador podría hacer aumentar el F1 score en casi un 10% para la misma arquitectura; o variaciones del 5% cuando se cambiaba la tasa de aprendizaje y los demás hiperparámetros se mantenían fijos.

Es claro que este tipo de enfoque es importante dentro del ciclo de vida de un proyecto de datos, pero ahora, exploraremos en la siguiente sección por qué también el enfoque centrado en datos es necesario.

1.4.2. Data-centric

Mientras el enfoque *model-centric* enfoca sus esfuerzos y tiempo en las arquitecturas, el enfoque ***data-centric*** regresa al origen y vea a los datos como materia prima del proyecto. Aquí, antes de pensar en capas, hiperparámetros o métricas, aparece un gesto fundamental: comprender. Y esa comprensión empieza con un EDA (*Exploratory Data Analysis*) honesto y profundo, un recorrido inicial por la textura y el comportamiento de los datos, sean registros tabulares, series de tiempo con pulsos y estacionalidades, o señales más complejas como el audio, para descubrir qué cuentan y qué ocultan.

Desde esta mirada, el *dataset* deja de ser un simple insumo para el entrenamiento y se convierte en el terreno donde la solución realmente se construye. El propósito ya no es entrenar un modelo, sino *representar un fenómeno*, y eso exige que los datos representen ese fenómeno con claridad.

Por eso, el trabajo *data-centric* pone especial cuidado en la ingeniería de características (*feature engineering*) cuyo propósito principal se centra en transformar datos crudos creando nuevas variables que capturen patrones invisibles, transformando aquellas que necesitan una escala común, estandarizando representaciones y completando valores faltantes de manera coherente con el contexto. Es un proceso creativo y técnico a la vez: una mezcla entre intuición, conocimiento del dominio y aplicación de buenas prácticas.

Lo que se busca es simple de decir, pero profundo en su trasfondo: que los datos transformados permitan que el modelo aprenda mejor.

Cuando esta etapa se trabaja con cuidado, el efecto es tangible: modelos más estables, comportamientos más consistentes, métricas que dejan de oscilar caprichosamente y una comprensión más fina del problema real. Incluso arquitecturas sencillas pueden alcanzar un desempeño notable cuando están apoyadas en un *dataset* bien construido, bien entendido y bien acondicionado.

El *data-centric* no es un complemento, es la base. Es la parte del proceso que nos recuerda que, antes de afinar un modelo, debemos asegurarnos de que los datos le hablen con claridad. Y cuando lo hacen, el aprendizaje se vuelve más eficiente, más confiable y cercano a la realidad que intentamos representar.

1.4.3. Enfoque híbrido

En la práctica, pocos proyectos se desarrollan bajo un enfoque exclusivamente model-centric o puramente data-centric. La mayoría de los procesos reales (esos que involucran datos imperfectos, restricciones de tiempo y objetivos cambiantes), requieren un enfoque **híbrido**, capaz de integrar decisiones tanto sobre la arquitectura como sobre los datos.

Este enfoque reconoce que el desempeño de un modelo no depende de un solo factor, sino de la interacción entre varios elementos: la calidad y profundidad del dataset, la riqueza de las características diseñadas, la forma en que se representan las señales, y la capacidad del modelo para capturar relaciones complejas. En un proyecto híbrido, no se parte de una única respuesta, sino de una pregunta constante:

¿Dónde podremos impactar de mejor manera, transformando datos o modificando los hiperparámetros del modelo?

A veces, una transformación adecuada en los datos puede desbloquear mejoras significativas. Otras veces, es la arquitectura la que necesita ajustarse, ya sea agregando capas, probando una red preentrenada, cambiando la función de activación, o incluso replanteando el tipo de modelo usado.

La riqueza del enfoque híbrido está en esa flexibilidad: en la capacidad del equipo para iterar entre los datos y el modelo, sin casarse con una sola dimensión del problema. No se trata de mezclar al azar, sino de establecer ciclos de experimentación bien definidos, donde cada modificación en los datos se prueba con diferentes modelos, y cada nuevo modelo se evalúa bajo condiciones de datos controladas y documentadas.

Un ejemplo destacado de este enfoque es el proyecto **FakeVoiceFinder**, en el cual he venido trabajando desde 2025. Es un *framework* en Python, publicado en GitHub, que permite realizar análisis *data-centric* y *model-centric* para la detección de audio sintético generado con IA. FakeVoiceFinder cubre casi todo el ciclo de vida del proyecto y solo necesita un insumo inicial: un dataset de audios naturales y sintéticos etiquetados. A partir de allí, automatiza el resto del proceso: partición de datos, ajuste de longitud de los audios, generación de representaciones espectrales, ajuste de hiperparámetros, selección y

entrenamiento de arquitecturas, cálculo de métricas y visualización comparativa de resultados. Desde la perspectiva del usuario, su uso es sencillo: puede seleccionar los valores de cada etapa o, si lo prefiere, dejar que el *framework* ejecute todo con los parámetros predeterminados.

Con esta estructura, el usuario puede comparar el desempeño de todas las arquitecturas para una misma representación espectral o, en sentido inverso, evaluar las cuatro representaciones utilizando una arquitectura específica. Además, *FakeVoiceFinder* permite analizar ambos elementos de manera conjunta, de modo que es posible identificar qué modelos son más sensibles al tipo de transformación y qué representaciones se ajustan mejor a cada arquitectura, revelando relaciones que no serían evidentes desde un único enfoque.

Por ello, el enfoque híbrido no es solamente una estrategia técnica; es también una actitud metodológica. Implica mirar el proyecto con humildad, evitar suponer que un solo componente determina el éxito o el fracaso, y mantener siempre una lógica experimental, comparativa y reproducible. Es, quizás, la forma más realista y más poderosa de abordar proyectos complejos, en los que los datos y los modelos se conciben y se refinan juntos, como partes de un mismo sistema vivo.

TIPOS DE DATOS E INGENIERÍA DE CARACTERÍSTICAS

Después del Capítulo 1, en el cual sentamos las bases de este libro, empezamos ahora a abordar los tipos de datos que lo componen. Sin embargo, antes de entrar en los detalles propios de los datos estructurados, de series de tiempo y de audio sintético, es necesario revisar algunos conceptos transversales que acompañan cualquier trabajo investigativo en ciencia de datos.

En este capítulo estudiaremos cómo se clasifican los datos según su forma (estructurados, semiestructurados y no estructurados) y según la escala en la que se encuentran (nominal, ordinal, de intervalo o de razón), pues estas decisiones determinan qué operaciones son válidas y qué transformaciones son pertinentes. Presentaremos también los fundamentos de la ingeniería de características: mejorar variables existentes, construir nuevas, seleccionar las más relevantes, extraer representaciones y permitir que ciertos modelos aprendan las suyas propias.

Estas ideas serán el punto de partida para los análisis, retos y proyectos que desarrollaremos a lo largo del libro. Comprender la naturaleza del dato no es un paso accesorio: es la base para diseñar experimentos más claros, tomar decisiones mejor informadas y avanzar con rigor en cada uno de los tres universos de datos que exploraremos.

2.1. Tipos de datos según su estructura

Cuando iniciamos un proyecto de ciencia de datos, una de las primeras preguntas que debemos responder es: **¿qué forma tienen los datos con los que vamos a trabajar?** La estructura del dato condiciona la forma en que podemos explorarlo, las transformaciones que serán posibles y el tipo de modelos que podrán utilizarse con mayor efectividad. Por eso, antes de cualquier análisis, es fundamental reconocer en qué categoría se encuentra la información disponible.

2.1.1. Datos estructurados

Los datos estructurados son, en general, los más sencillos de manipular, porque están organizados en formato tabular: **filas** que representan observaciones y **columnas** que corresponden a variables o características. En muchos casos, una de estas columnas corresponde a la **variable objetivo o salida**, es decir, aquella que deseamos predecir o explicar.

Este tipo de datos requiere poco espacio de almacenamiento y se encuentra con frecuencia en archivos .csv, bases de datos relacionales o sistemas transaccionales.

Para ilustrarlo, supongamos que contamos con información sobre los *precios de viviendas* de un barrio, tomada de anuncios publicados en un buscador de inmuebles. De cada anuncio podemos extraer variables como:

- tamaño del apartamento (m^2),
- antigüedad del edificio,
- número de habitaciones y baños,
- piso en el que se ubica,
- cantidad de parqueaderos,
- presencia de depósito, terraza o gimnasio, entre otros.

Cada una de estas características se convierte en una columna del dataset, y serán las entradas de un modelo de predicción. La columna del *precio* será la salida a estimar. Con datos históricos, un modelo de regresión puede

aprender patrones y predecir el valor de otros inmuebles del mismo sector, útil tanto para vendedores que desean fijar un precio justo como para compradores que quieren evaluar si un apartamento está sobrevalorado, subvalorado o dentro del rango esperado.

Tabla 2. Ejemplo de dataset estructurado: precio de vivienda (en millones de pesos).

M2	Años	Hab.	Baños	Piso	Parq.	Depósito	Terraza	Gimnasio	Precio
100	2	3	3	8	1	0	1	1	500
60	2	2	2	6	1	0	1	1	320
80	2	2	3	7	1	0	1	1	400
100	15	3	3	4	1	0	1	0	400
…	…	…	…	…	…	…	…	…	…

Con este tipo de datos, un modelo podría identificar, por ejemplo, que la antigüedad del inmueble tiene un impacto mayor en el precio que el número de habitaciones, o que un apartamento pequeño pero nuevo puede tener un valor similar a uno más grande pero más antiguo.

En este libro, trabajaremos con dos grandes categorías de datos estructurados:

- datos tabulares básicos, y
- series de tiempo, una forma especial de datos estructurados donde el orden temporal es esencial.

El primer tipo se aborda en el Capítulo 3, mientras que el segundo se desarrolla en el Capítulo 4.

2.1.2. Datos semiestructurados

Los datos semiestructurados no siguen un formato tabular rígido, pero sí cuentan con una organización interna que facilita su interpretación. Suelen venir en formatos como *JSON*, *XML* o en archivos de *logs*, donde la información se representa mediante pares clave vs. valor, jerarquías o etiquetas.

Aunque no pueden analizarse directamente como un dataset tabular, pueden transformarse mediante procesos de limpieza e *ingeniería de características*, o convertirse en tablas para análisis posteriores. En investigación es común enriquecer estos datos para integrarlos en pipelines más complejos.

Ejemplos frecuentes:

- respuestas de APIs en JSON,
- datos provenientes de sistemas web,
- archivos XML con metadatos,

2.1.3. Datos no estructurados

Los datos no estructurados carecen de un formato predefinido o un esquema fijo. Aquí encontramos información almacenada en formatos libres: texto, imágenes, audio o video**.** Para extraer valor de estos datos se requieren técnicas especializadas como procesamiento de lenguaje natural, visión por computador o análisis de audio.

En este libro trabajaremos especialmente con un subtipo muy particular: audio sintético**,** un dato no estructurado que, al transformarse en representaciones espectrales (como Mel, log-Mel o CQT), puede analizarse mediante métodos clásicos y modelos profundos.

Ejemplos comunes de datos no estructurados:

- documentos de texto,
- fotografías y videos,
- grabaciones de voz,
- comentarios en redes sociales.

La Figura 3 resumen las diferencias entre datos estructurados y datos no estructurados.

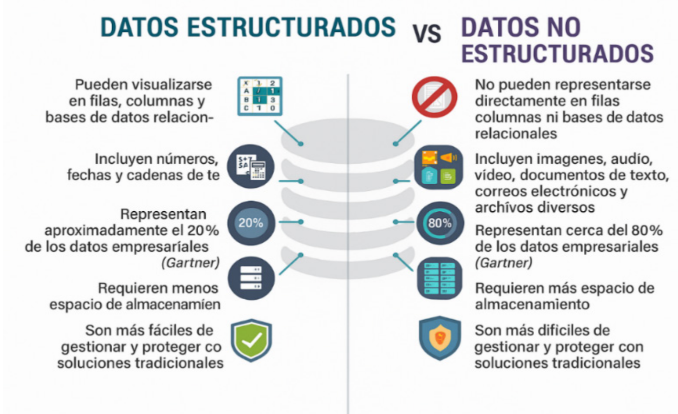


Figura 3. Figura comparativa datos estructurados vs. datos no estructurados.

2.2. Niveles de los datos

Una vez definidos los tipos de datos, vamos ahora a revisar los niveles de los datos, que típicamente encontramos en datos estructurados y semiestructurados. Estos son: nivel nominal, nivel ordinal, nivel intervalo, y escala de razón.

2.2.1. Nivel Nominal

Corresponde a variables o *features* que no poseen ningún tipo de jerarquía entre sí. Supongamos, por ejemplo, que contamos con la información de las localidades de una ciudad como Bogotá (Colombia). En ese caso, podríamos encontrar nombres como Usaquén, Chapinero, Barrios Unidos y Teusaquillo.

Cada una de estas localidades debe transformarse en un valor numérico que pueda ser interpretado por un modelo de aprendizaje de máquina durante una tarea de predicción.

Para este tipo de variables, la transformación más adecuada es *one-hot encoding*, en la cual se crean tantas columnas nuevas como categorías tenga la variable. En nuestro ejemplo de cuatro localidades, se generarían cuatro columnas, y para cada registro solo se activaría una de ellas, así:

LOCALIDAD
USAQUÉN
CHAPINERO
BARRIOS UNIDOS
TEUSAQUILLO

One-hot-encoding →

USAQUÉN	CHAPINERO	BARRIOS UNIDOS	TEUSAQUILLO
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

Figura 4. Ejemplo transformación de datos tipo nominal

La ventaja de esta transformación es que evita introducir sesgos derivados de asignar números consecutivos a cada categoría (por ejemplo, Usaquén = 1, Chapinero = 2, etc.), lo cual podría interpretarse erróneamente como un orden o una distancia entre ellas. En cambio, con *one-hot encoding*, cada categoría existe únicamente en su propia columna.

No obstante, cuando una variable tiene un número muy elevado de categorías, esta transformación puede incrementar de manera significativa la cantidad de columnas del dataset, aumentando la complejidad del modelo.

2.2.2. Nivel Ordinal

A diferencia del nivel anterior, en este nivel sí existe un sentido de jerarquía. Se utiliza, por ejemplo, cuando estamos asignando una valoración cualitativa al nivel de satisfacción de un servicio.

Supongamos que queremos calificar qué tan satisfechos estamos con la clase de la electiva de Data Engineering. Las opciones son: muy contento, contento, neutral, insatisfecho, muy insatisfecho. En este caso, es evidente que “muy contento” es mejor que “muy insatisfecho”, por lo que podemos asignar una escala cuantitativa con jerarquía, así:

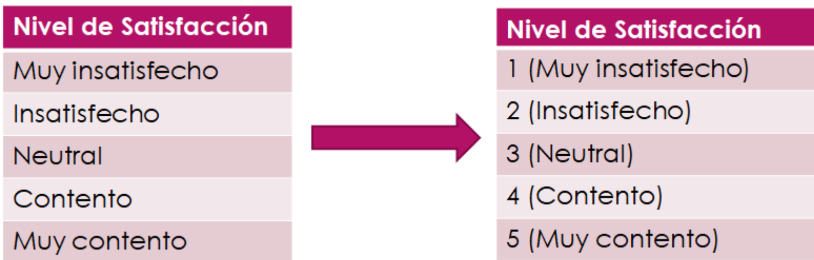


Figura 5. Ejemplo transformación de datos tipo ordinal

2.2.3. Nivel Intervalo

En este nivel existe un sentido claro de orden, y los valores están expresados en una escala numérica continua (enteros o flotantes). Sin embargo, el cero no representa ausencia absoluta de la magnitud medida.

Un ejemplo típico es la temperatura. Sabemos que en determinada ciudad los valores pueden oscilar entre 10 °C y 22 °C. En este tipo de variables es posible calcular promedios, medianas, mínimos, máximos o desviaciones estándar sin problema.

2.2.4. Escala de razón

A diferencia del nivel intervalo, en este caso sí existe un cero absoluto real, lo que permite interpretar las proporciones de manera significativa. Ejemplos de este tipo incluyen: moneda, edad, altura y peso.

Esto permite afirmaciones como:

- una persona pesa el doble que otra,
- un adulto mide el doble que un niño, o
- alguien dispone de la tercera parte del presupuesto de otra persona.

Cuando los datos pertenecen a esta categoría, es válido aplicar operaciones como la media aritmética, geométrica o armónica, pues estas respetan la proporcionalidad de las magnitudes.

Supongamos que contamos con dos columnas: Satisfacción del usuario (CSAT) y Índice de promotores (NPS). La primera se mide en una escala de 0–5 y la segunda en una escala de 0–10. Nuestro objetivo es construir una nueva columna que represente la calificación global de cada departamento.

Tabla 3. Ejemplo de columnas tipo escala de razón.

Departamento	Satisfacción Usuario (CSAT)	Índice de promotores (NPS)
A	4.0	7.5
B	3.0	9.0

Media aritmética

$A = (4.0 + 7.5) / 2 = 5.75$

$B = (3.0 + 9.0) / 2 = 6.00$

Media geométrica

$A = \sqrt{(4.0 \times 7.5)} = \sqrt{30} \approx 5.48$

$B = \sqrt{(3.0 \times 9.0)} = \sqrt{27} \approx 5.20$

Media armónica

$A = 2 / (1/4.0 + 1/7.5) = 2 / 0.3833 \approx 5.22$

$B = 2 / (1/3.0 + 1/9.0) = 2 / 0.4444 \approx 4.50$

Con la media aritmética, la calificación más alta la obtiene el Departamento B. Sin embargo, con la media geométrica y la media armónica, la nota más alta corresponde al Departamento A. La mayor diferencia se observa con la media armónica, ya que esta penaliza los casos en los que los valores presentan una mayor dispersión entre sí.

2.3. Ingeniería de Características

La ingeniería de características (*Feature Engineering*, FE) es una de las etapas más determinantes en un proyecto de ciencia de datos. Incluso el mejor modelo no puede compensar datos mal representados. Por eso, antes de entrenar cualquier algoritmo, debemos transformar, depurar y reformular las variables para que capturen mejor los patrones relevantes en los datos.

En esta sección revisamos cinco enfoques fundamentales: mejoramiento, construcción, selección, extracción y aprendizaje automático de características. Cada técnica tiene un propósito distinto, pero todas buscan lo mismo: mejorar la calidad de la representación de los datos.

Figura 6. Consolidados tipos de Feature Engineering.

A continuación, se explicará cada una de ellas.

2.3.1. Mejoramiento de Características (Feature Improvement)

El mejoramiento consiste en optimizar las variables existentes para que sean más estables, interpretables y útiles.

Tareas típicas:

- Imputación de valores faltantes, usando:
  - Moda para variables nominales/ordinales.
  - Mediana para datos numéricos asimétricos.
  - Media, en distribuciones simétricas.
- Estandarización, llevando los datos a media 0 y desviación estándar 1.
- Normalización, ajustando valores a un rango como [0, 1].
- Tratamiento de *outliers*, por winsorización o transformación logarítmica.

Para ilustrar esta técnica de FE, supongamos que contamos con un dataset de precios de vivienda acompañado del área en metros cuadrados. En una de las filas, falta el valor del área. En términos generales, tenemos dos opciones: eliminar la fila con datos faltantes o imputar el valor con una medida representativa, como la mediana.

Partimos del siguiente ejemplo:

Tabla 4. Ejemplo precio de vivienda, antes de imputación de datos.

Área	Precio
60	350
NaN	420
48	395

El valor faltante del área puede imputarse con la mediana (54 m²), quedando así:

Tabla 5. Ejemplo precio de vivienda, después de imputación de datos.

Área	Precio
60	350
54	420
48	395

Esto no crea nuevas variables, pero mejora la calidad de las ya existentes.

2.3.2. Construcción de Características (Feature Construction)

La construcción de características consiste en crear nuevas variables a partir de la información disponible, o integrar datos externos que aumenten la capacidad predictiva del modelo.

Estrategias comunes:

- Combinar variables: razones, diferencias, promedios, proporciones.
- Codificar variables categóricas (one-hot encoding, ordinal encoding, etc.).
- Cruzar características (interacciones entre variables).
- Integrar nuevas fuentes de datos (datos geográficos, climatológicos, demográficos, etc.).

Recordando el nivel nominal, visto en la sección anterior, podemos crear nuevas características a partir de una variable de tipo nominal, con una columna por cada opción tenga esa característica. Así, tendremos tantas nuevas columnas como opciones tiene esa característica.

2.3.3. Selección de Características

En *datasets* con gran cantidad de variables, no todas aportan información relevante. Algunas incluso pueden introducir ruido o distorsiones en el modelo. Por ejemplo, en el caso de predicción del precio de un apartamento, podríamos tener las siguientes características

Área, número habitaciones, número baños, total parqueaderos, total depósito, gimnasio, terraza, piso dentro del edificio, color de las paredes, tipo de baldosa, tipo de cortinas, ….

Es evidente que variables como **el color de las paredes** no son determinantes para predecir el precio del inmueble. Dos apartamentos con colores similares pueden tener precios completamente distintos debido a

características estructurales. Por lo tanto, esta variable debería eliminarse antes del modelamiento.

Dentro de las motivaciones principales para realizar la selección de características, es:

- Evitar la maldición de la dimensionalidad.
- Eliminar características redundantes o altamente correlacionadas.
- Aumentar la interpretabilidad del modelo.
- Mejorar el rendimiento y la estabilidad del modelo.

Técnicas frecuentes:

- Filtro: pruebas estadísticas, correlaciones, chi-cuadrado.
- Wrapper: RFE (eliminación recursiva).
- Embedded: árboles de decisión, regularización L1 (Lasso).

2.3.4. Extracción de Características

La extracción aplica técnicas matemáticas o estadísticas para transformar un conjunto de variables en otro más informativo o más compacto. Aquí no creamos variables manualmente; las derivamos mediante transformaciones.

Ejemplos de técnicas:

- PCA (reduce dimensionalidad identificando componentes principales).
- SVD (útil en sistemas de recomendación).
- TF-IDF o Bag-of-Words para texto.
- Espectrogramas para audio.

Este último caso será utilizado en el capítulo final del libro, donde transformaremos señales de audio en representaciones visuales de su energía espectral.

2.3.5. Aprendizaje Automático de Características

En esta técnica, un modelo, generalmente redes profundas, aprende de manera automática las representaciones internas o latentes de los datos, sin necesidad de especificarlas explícitamente. Es fundamental en el procesamiento de datos no estructurados.

Modelos frecuentes:

- Autoencoders → crean representaciones comprimidas.
- CNNs → extraen patrones espaciales (imágenes).
- Transformers → extraen relaciones contextuales (texto).
- GANs → aprenden distribuciones complejas.

Retomando el proyecto *FakeVoiceFinder*, dentro de la CNN las capas convolucionales y de *pooling* aprenden patrones relevantes sin ser instruidas explícitamente. Estas capas detectan irregularidades de timbre, texturas acústicas y artefactos característicos de voces sintéticas. Con estas representaciones, el bloque final del modelo puede discernir si un audio es natural o generado artificialmente.

CAPÍTULO III

DATOS ESTRUCTURADOS

En este capítulo abordaremos el trabajo en ciencia de datos utilizando datos estructurados, aplicando la metodología de aprendizaje basado en proyectos (ABP) a través de varios casos de estudio. Comenzaremos con un Análisis Exploratorio de Datos (EDA) aplicado a un dataset de reporte de casos de COVID-19 en la ciudad de Bogotá (Colombia). Posteriormente, realizaremos tanto EDA como Ingeniería de Características (FE) a un dataset de deserción de clientes bancarios.

3.1. Caso de estudio 1: Análisis Exploratorio dataset COVID-19 de Bogotá

En el primer semestre de 2020, cuando inició la pandemia por COVID-19, la Alcaldía de Bogotá publicó de forma abierta los datos de reporte diario de casos positivos, con el fin de poner esta información a disposición de toda la comunidad. Este dataset hace parte del portal **Datos Abiertos Bogotá**, bajo el nombre *"Casos confirmados de COVID-19 en Bogotá D.C."*. La actualización más reciente se encuentra disponible en: https://n9.cl/5uyga

Con el dataset del **18 de abril de 2020**, se elaboró un video explicativo publicado en el canal de YouTube de Dora María Ballesteros, disponible en: https://youtu.be/kezqpilbKyA

Para este capítulo, utilizaremos el dataset del **19 de mayo de 2020**, sobre el cual se realizó un proceso de EDA en un **Jupyter Notebook** (para propósitos académicos, recomiendo Google Colaboratory).

El análisis se realizó en Python, iniciando con la importación de librerías:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
data = pd.read_csv('/content/osbcovid19_Mayo9.csv', encoding = "ISO-8859-1")
data_copy=data.copy()
```

En este caso:

- numpy se emplea para el manejo eficiente de arreglos.
- matplotlib se usa para la visualización de datos.
- pandas es la herramienta principal para la manipulación de dataframes.

Con read_csv() realizamos la lectura del dataset desde la ubicación donde fue almacenado.

3.1.1. Pre-visualización del dataset.

Ahora, podemos determinar el tamaño del dataset y realizar una primera visualización del contenido:

```
registro=len(data)
print(registro)
data.head(registro)
```

Este dataset contiene 8 columnas y 3824 filas. La figura 7 presenta una pre-visualización del dataset.

	Fecha de diagnóstico	Ciudad de residencia	Localidad de residencia	Edad	Sexo	Tipo de caso	Ubicación	Estado
0	10/04/2020	Bogotá	Usaquén	19	F	Importado	Casa	Recuperado
1	10/04/2020	Bogotá	Engativá	22	F	Importado	Casa	Recuperado
2	10/04/2020	Bogotá	Engativá	28	F	Importado	Casa	Recuperado
3	10/04/2020	Bogotá	Fontibón	36	F	Importado	Casa	Recuperado
4	10/04/2020	Bogotá	Kennedy	42	F	Importado	Casa	Recuperado
...	...	...	...	...	...	...	...	...
3819	7/05/2020	Bogotá	San Cristóbal	42	M	En estudio	Casa	Moderado
3820	7/05/2020	Bogotá	Teusaquillo	36	M	En estudio	Casa	Moderado
3821	7/05/2020	Bogotá	Engativá	56	F	En estudio	Hospital	Severo
3822	8/05/2020	Bogotá	Engativá	33	M	En estudio	Casa	Moderado
3823	8/05/2020	Bogotá	Sin Dato	41	M	En estudio	Hospital	Severo

3824 rows × 8 columns

Figura 7. Pre-visualización dataset COVID-19 Bogotá

3.1.2. Tipo de datos en el dataset

Continuamos con la exploración del dataset. Es importante no solo identificar cuántas columnas lo conforman, sino también conocer el **tipo de dato** asociado a cada una de ellas. Por ejemplo: ¿son variables numéricas o categóricas?

Para esto utilizamos el siguiente comando:

```
data.info()
```

El resultado se muestra en la Figura 8.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3824 entries, 0 to 3823
Data columns (total 8 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   Fecha de diagnóstico      3823 non-null   object
 1   Ciudad de residencia      3824 non-null   object
 2   Localidad de residencia   3824 non-null   object
 3   Edad                      3824 non-null   int64
 4   Sexo                      3824 non-null   object
 5   Tipo de caso              3824 non-null   object
 6   Ubicación                 3824 non-null   object
 7   Estado                    3824 non-null   object
dtypes: int64(1), object(7)
memory usage: 239.1+ KB
```

Figura 8. Tipo de datos del dataset COVID-19 Bogotá

Con esta información observamos que, de las ocho columnas disponibles, **solo una es numérica** (la variable *Edad*, de tipo entero). Las demás corresponden a **variables categóricas** (texto), incluida la **fecha de diagnóstico**, la cual inicialmente aparece como una cadena de caracteres y no como un tipo de dato temporal (datetime).

3.1.3. Distribución de los estados reportados del dataset

Posteriormente, podemos visualizar el estado en el que se encuentran los casos reportados de COVID-19 en el dataset. Para ello utilizamos:

```
from matplotlib import pyplot as plt
import seaborn as sns
data.groupby('Estado').size().plot(kind='barh', color=sns.palettes.mpl_palette('Dark2'))
plt.gca().spines[['top', 'right',]].set_visible(False)
```

Y obtenemos:

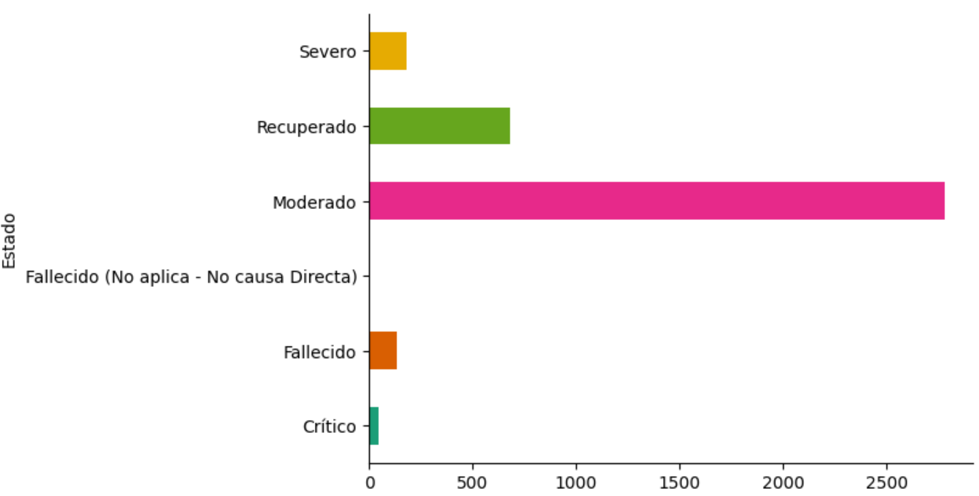


Figura 9. Distribución de los estados de los casos reportados en el dataset COVID-19 Bogotá

¿Por qué es importante este reporte?

Porque permite a las autoridades tomar decisiones basadas en la evolución del estado clínico de los casos. Para la fecha del dataset, la mayoría correspondía a casos moderados; sin embargo, con el paso del tiempo, la distribución cambió hacia casos severos y, lamentablemente, fallecidos.

Estas variaciones en las proporciones son fundamentales para activar alertas tempranas, ajustar políticas de salud pública y gestionar adecuadamente la capacidad hospitalaria.

3.1.4. Distribución de los estados reportados vs. Localidad

Vamos a realizar una gráfica que nos permita identificar el estado de los pacientes en cada una de las localidades de Bogotá. Utilizaremos una gráfica tipo **heatmap**:

```
from matplotlib import pyplot as plt
import seaborn as sns
import pandas as pd

plt.subplots(figsize=(7, 7))

df_2dhist = pd.DataFrame({
    x_label: grp['Estado'].value_counts()
    for x_label, grp in data.groupby('Localidad de residencia')
})
sns.heatmap(df_2dhist, cmap='crest')
plt.xlabel('Localidad de residencia')
_ = plt.ylabel('Estado')
```

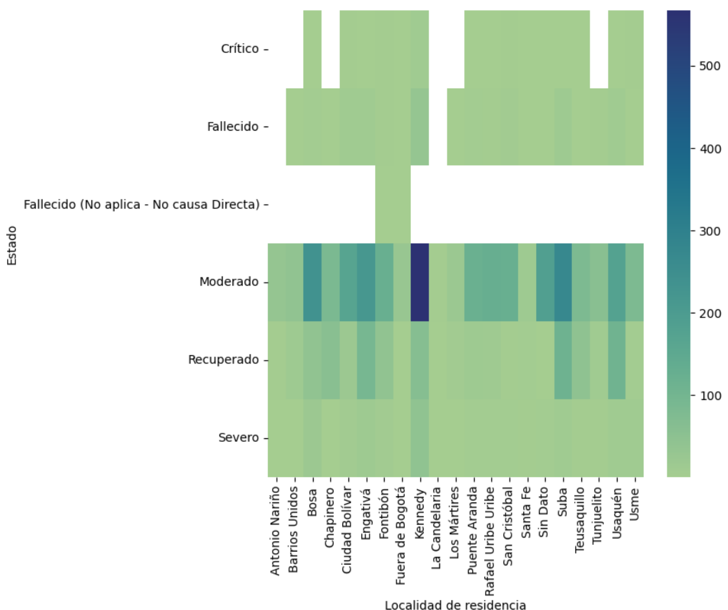


Figura 10. Distribución del estado vs. localidad de los casos reportados en el dataset COVID-19 Bogotá

¿Por qué es importante este reporte?

En su momento, esta información le permitió a la Alcaldesa tomar decisiones relacionadas con el "cierre" de localidades. Por ejemplo, para la fecha de este dataset, la localidad de Kennedy presentaba el mayor número de casos en estado Moderado, mientras que La Candelaria registraba la menor cantidad en ese mismo estado.

Si observamos el estado crítico, algunas localidades aún no tenían pacientes reportados en dicha condición.

Por esta razón, no era necesario cerrar todas las localidades de forma simultánea: se buscaba proteger la economía sin perder de vista el comportamiento real del impacto de la pandemia al interior de la ciudad.

3.1.5. Conversión de la fecha de diagnóstico

Como vimos previamente, la columna "Fecha de diagnóstico" es de tipo *object*, lo que indica que aún no se encuentra en un formato de fecha que nos permita trabajar con ella. Aunque los datasets con información temporal los abordaremos en el capítulo de Series de Tiempo, en este caso trabajaremos con esta columna sin convertirla en el índice del dataset, tratándola simplemente como un dato estructurado.

Queremos transformar esta columna, originalmente un texto, en un número que represente el consecutivo del año: 1 para el 1 de enero, 2 para el 2 de enero, y así sucesivamente. Para ello escribimos el siguiente código en Python:

```
data["Fecha de diagnóstico"] = pd.to_datetime(data["Fecha de diagnóstico"], dayfirst='True').dt.strftime("%Y%m%d") # quita / (ej. 20200410)
data['Fecha de diagnóstico'] = data['Fecha de diagnóstico'].astype('datetime64[ns]') # separa con - (ej. 2020-04-10)
data['Fecha de diagnóstico'] = data['Fecha de diagnóstico'].dt.dayofyear # convierte al día del año (ej. 101)
```

Este proceso se realiza en tres pasos: primero, convertimos el texto inicial a un formato de fecha indicando que el día aparece antes que el mes, y lo transformamos temporalmente a una cadena YYYYMMDD para eliminar caracteres separadores. Luego, reconvertimos esta cadena a un tipo .datetime64[ns] para que Python la interprete correctamente como fecha. Finalmente, extraemos el número de día correspondiente dentro del año mediante .dt.dayofyear, obteniendo un entero entre 1 y 365 (o 366 en años bisiestos). Este valor numérico facilita realizar análisis temporales sin necesidad de trabajar aún con series de tiempo completas.

Adicionalmente, vamos a renombrar la columna de Fecha de diagnóstico, dado que ahora corresponde es al “Día de diagnóstico”

```
data = data.rename(columns={'Fecha de diagnóstico':
'Día de diagnóstico'})
data.head(registro)
```

Y obtenemos:

	Día de diagnóstico	Ciudad de residencia	Localidad de residencia	Edad	Sexo	Tipo de caso	Ubicación	Estado
0	101.0	Bogotá	Usaquén	19	F	Importado	Casa	Recuperado
1	101.0	Bogotá	Engativá	22	F	Importado	Casa	Recuperado
2	101.0	Bogotá	Engativá	28	F	Importado	Casa	Recuperado
3	101.0	Bogotá	Fontibón	36	F	Importado	Casa	Recuperado
4	101.0	Bogotá	Kennedy	42	F	Importado	Casa	Recuperado
...	...	...	...	...	...	...	...	...
3819	128.0	Bogotá	San Cristóbal	42	M	En estudio	Casa	Moderado
3820	128.0	Bogotá	Teusaquillo	36	M	En estudio	Casa	Moderado
3821	128.0	Bogotá	Engativá	56	F	En estudio	Hospital	Severo
3822	129.0	Bogotá	Engativá	33	M	En estudio	Casa	Moderado
3823	129.0	Bogotá	Sin Dato	41	M	En estudio	Hospital	Severo

Figura 11. Pre-visualización del dataset posterior a la conversión de la fecha de diagnóstico

Al comparar la Figura 11 con la Figura 7, observamos que fechas como **10/04/2020** han sido reemplazadas por **101**, que corresponde al consecutivo del año. Contar ahora con esta columna en formato numérico nos permite realizar varios análisis exploratorios que antes no eran viables.

3.1.6. Distribución temporal de los estados

La Figura 9 nos mostraba la distribución de los estados, pero no nos aportaba información en relación a cómo había evolucionado a lo largo de los días.

Para ello, continuaremos con gráfica tipo *heatmap*, con el siguiente código:

```
from matplotlib import pyplot as plt
import seaborn as sns
import pandas as pd
plt.subplots(figsize=(8, 5))
df_2dhist = pd.DataFrame({
    x_label: grp['Fecha de diagnóstico'].value_counts()
    for x_label, grp in data.groupby('Estado')
})
sns.heatmap(df_2dhist, cmap='crest')
plt.xlabel('Ubicación')
_ = plt.ylabel('Dia de diagnóstico')
```

Este código genera un **mapa de calor** que muestra la frecuencia de aparición del **Día de diagnóstico** en cada uno de los estados reportados. Primero, se agrupan los datos por la columna *Estado* y, dentro de cada grupo, se calcula cuántas veces aparece cada valor de *Día de diagnóstico*. Con ello se construye un nuevo dataframe (df_2dhist), donde las columnas representan los estados y las filas los días del año. Finalmente, sns.heatmap() visualiza esta matriz utilizando la paleta *crest*, permitiendo identificar patrones temporales asociados a cada estado.

Obteniendo:

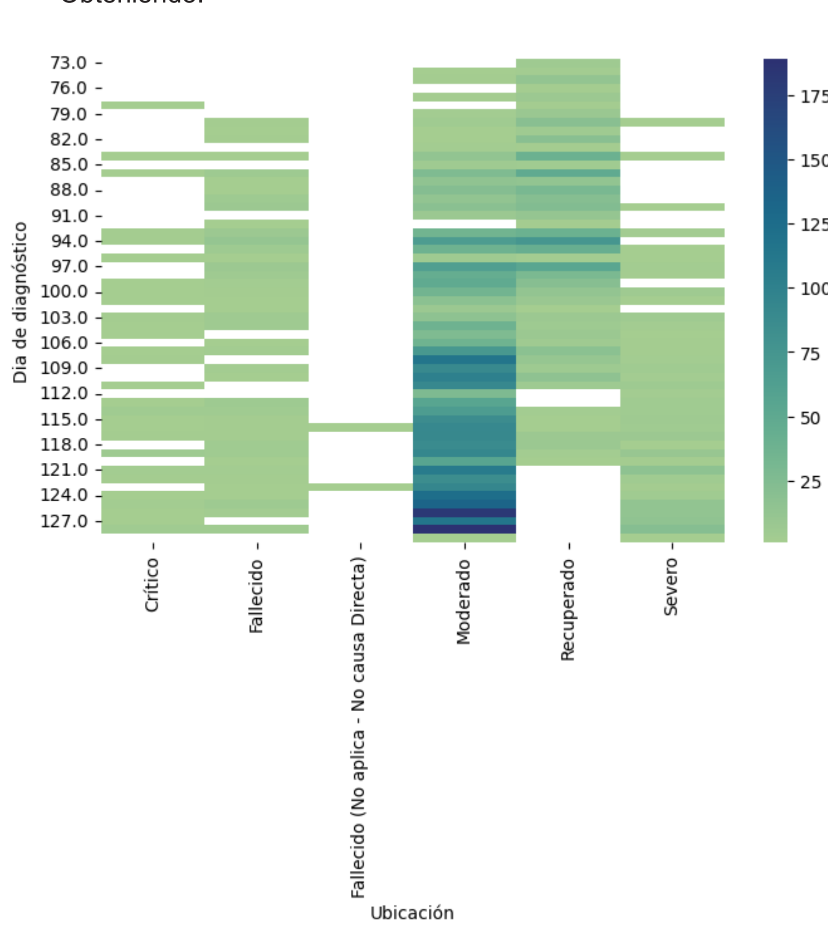


Figura 12. Distribución del estado vs. día de diagnóstico de los casos reportados en el dataset COVID-19 Bogotá

¿Por qué es importante este reporte?

Porque permite identificar cómo evolucionan los diferentes estados clínicos a lo largo del tiempo. Al observar la frecuencia del *Día de diagnóstico* para cada estado, es posible detectar patrones temporales, picos de contagio, momentos de mayor presión sobre el sistema de salud y periodos críticos que requieren intervención. Este tipo de visualización ayuda a comprender no solo cuántos casos hubo, sino **cuándo** ocurrieron y **cómo variaron entre estados**, ofreciendo información clave para la toma de decisiones en salud pública.

3.1.7. Casos de contagio por Género

Una de las preguntas que surgió al inicio de la pandemia era si los hombres o las mujeres eran más susceptibles al contagio. Para responderla, utilizamos la columna “Sexo” del dataset y determinamos cuántos registros corresponden a “M” y cuántos a “F”.

El siguiente código calcula el número total de personas contagiadas por COVID-19 en Bogotá y desglosa esta cantidad por sexo, obteniendo tanto las cifras absolutas como el porcentaje que cada grupo representa dentro del total.

Posteriormente, se genera una gráfica de barras estética que compara visualmente el número de casos entre hombres y mujeres, incorporando colores diferenciados, etiquetas con valores y porcentajes, y un diseño limpio adecuado para el análisis exploratorio de datos.

```
cantidad_contagiados = len(data)
cantidad_contagiados_M = len(data[data[“Sexo”] == “M”])
porc_M = cantidad_contagiados_M * 100 / cantidad_contagiados
cantidad_contagiados_F = len(data[data[“Sexo”] == “F”])
porc_F = cantidad_contagiados_F * 100 / cantidad_contagiados

print(‘cantidad de contagiados por COVID en Bogotá:’,
cantidad_contagiados)
print(‘cantidad de hombres contagiados por COVID:’,
cantidad_contagiados_M,
    f’({porc_M:.2f}%)’)
print(‘cantidad de mujeres contagiadas por COVID:’,
cantidad_contagiados_F,
    f’({porc_F:.2f}%)’)

plt.figure(figsize=(6,4))
labels = [‘Hombres’, ‘Mujeres’]
values = [cantidad_contagiados_M, cantidad_contagiados_F]
porcentajes = [porc_M, porc_F]

plt.figure(figsize=(6,5))
bars = plt.bar(labels, values,
```

```
               color=['#1aa6b7', '#a3bac3'],  # turquesa + gris azulado
               edgecolor='#444444')

plt.title('Casos confirmados de COVID-19 en Bogotá por
sexo', fontsize=12)
plt.ylabel('Número de casos', fontsize=16)

ax = plt.gca()
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)

plt.grid(axis='y', linestyle='--', alpha=0.3)

for bar, pct in zip(bars, porcentajes):
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2,
             height,
             f'{int(height)}\n({pct:.1f}%)',
             ha='center', va='bottom',
             fontsize=9)

plt.tight_layout()
plt.show()
```

Y se obtiene la siguiente figura:

```
cantidad de contagiados por COVID en Bogotá: 3824
cantidad de hombres contagiados por COVID: 1931 (50.50%)
cantidad de mujeres contagiadas por COVID: 1893 (49.50%)
<Figure size 600x400 with 0 Axes>
```

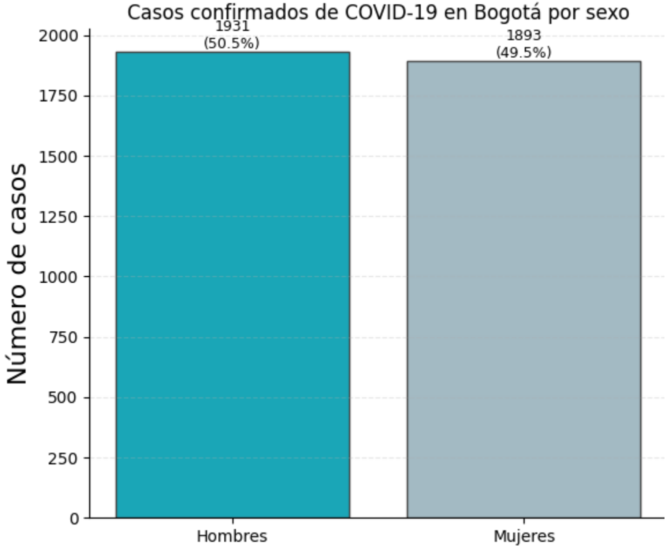


Figura 13. Distribución de casos reportados por género

De acuerdo con la figura anterior, se puede derrumbar el mito que los hombres se contagiaban mucho más que las mujeres.

3.1.8. Casos de contagio por Día de Diagnóstico

La siguiente inquietud que se generó tanto en la comunidad como en las autoridades de salud, era que tan rápido estaba creciendo la tasa de contagios. Es decir, a medida que pasaban los días, cuantos nuevos casos iban apareciendo. Para ello, utilizamos el siguiente código en Python:

```
# Número de días únicos reportados
num_dias = data['Día de diagnóstico'].nunique()

# Histograma estilo "elegante"
plt.figure(figsize=(10,4))
plt.hist(data['Día de diagnóstico'],
         bins=num_dias,
         color='#1aa6b7',        # Turquesa elegante
```

```
        edgecolor='#2e4756',   # Gris azul oscuro para delinear
        alpha=0.85)

plt.xlabel('Fecha de diagnóstico', fontsize=10)
plt.ylabel('Número de casos', fontsize=10)
plt.title('Histograma de casos diarios de COVID-19 en
Bogotá', fontsize=12)
# Estilo visual más limpio
ax = plt.gca()
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
plt.xticks(rotation=45)
plt.grid(axis='y', linestyle='--', alpha=0.3)
plt.tight_layout()
plt.show()
```

Iniciamos calculando cuántos días únicos aparecen en la columna "Día de diagnóstico" con las instrucción nunique(), lo cual permite identificar cuántas fechas diferentes de reporte existen en el dataset. Con este valor, se genera un histograma en el que cada *bin* representa exactamente un día de la pandemia. Esto permite visualizar la distribución diaria de casos sin agrupar varios días en un mismo intervalo. Finalmente, se añaden etiquetas a los ejes, un título descriptivo y se rota el texto del eje horizontal para facilitar la lectura de las fechas, mostrando así un histograma claro y detallado del número de casos por día.

Obtenemos la siguiente gráfica:

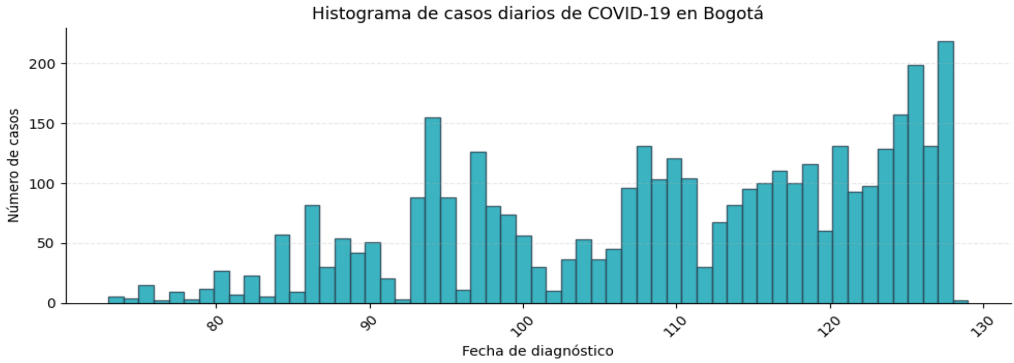


Figura 14. Histograma de casos diarios de COVID-19 en Bogotá

¿Por qué es importante este reporte?

El histograma permite observar de manera directa la evolución de los contagios diarios de COVID-19 en Bogotá, mostrando cómo se pasó de una fase inicial con pocos casos a un crecimiento progresivo y sostenido hacia el final del periodo analizado. Este tipo de visualización es fundamental en un EDA porque hace visibles patrones, picos y tendencias que no se aprecian al revisar la tabla de datos, facilitando así la identificación de momentos críticos, cambios en la dinámica de transmisión y posibles puntos de interés para un análisis más profundo o una toma de decisiones informada. Por ejemplo, al inicio de la pandemia (alrededor del día 70), la cantidad de casos reportados por día era muy baja (menor de 20); posteriormente, se observa un primer pico alrededor del día 95, seguido de una disminución en los contagios diarios (muy seguramente como resultado de las medidas de restricción y cierres por localidades). Sin embargo, hacia el día 120 del año (es decir, iniciando abril) se evidencia un aumento considerable en el número de contagios, superando los 100 casos diarios y marcando una fase de expansión más acelerada del virus.

3.1.9. Evolución acumulada de casos de COVID-19 en Bogotá

Como complemento a la gráfica anterior, podemos también dibujar el acumulado de casos, y así fácilmente observar cómo va cambiando la pendiente de casos acumulados, es decir, que tan rápido está creciendo en un momento dado frente al inicio de la pandemia.

Para ello, primero ordenamos la columna “Día de diagnóstico” de manera cronológica y luego agrupamos los registros para contabilizar cuántos casos corresponden a cada día. A continuación, aplicamos una suma acumulada para obtener el total progresivo de contagios a lo largo del tiempo. Finalmente, graficamos estos valores en una curva que permite visualizar cómo se incrementaron los casos de forma acumulada durante el periodo analizado.

```
# Ordenar por fecha
data = data.sort_values('Día de diagnóstico')

# Contar casos diarios
casos_por_dia = data.groupby('Día de diagnóstico').size()
```

```
# Calcular los contagios acumulados
acumulado = casos_por_dia.cumsum()

plt.figure(figsize=(10,4))
plt.plot(acumulado.index, acumulado.values,
        color='#1aa6b7', linewidth=2.5)

plt.xlabel('Día de diagnóstico', fontsize=10)
plt.ylabel('Contagios acumulados', fontsize=10)
plt.title('Evolución acumulada de casos de COVID-19 en Bogotá',
fontsize=12)

ax = plt.gca()
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)

plt.grid(axis='y', linestyle='--', alpha=0.3)
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

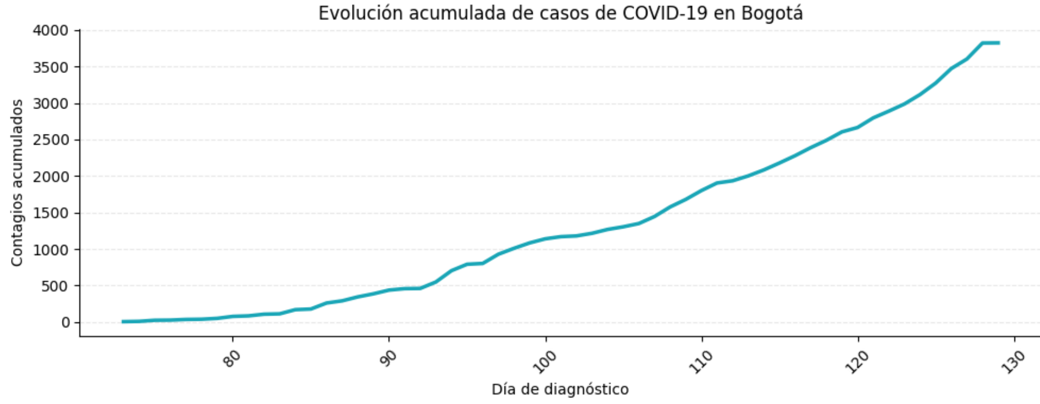


Figura 15. Evolución acumulada de casos de COVID-19 en Bogotá

3.1.10. Distribución de casos de COVID-19 por edad

Cuando hablamos de distribución, normalmente nos referimos a un histograma. Por ello, nuevamente utilizaremos este tipo de gráfica para nuestra figura, así:

```
# Asegurar que la columna Edad sea numérica
data['Edad'] = pd.to_numeric(data['Edad'], errors='coerce')

# Calcular número de bins: uno por cada edad en el rango del dataset
edad_min = int(data['Edad'].min())
edad_max = int(data['Edad'].max())
num_bins = edad_max - edad_min

plt.figure(figsize=(10,4))
plt.hist(
    data['Edad'].dropna(),
    bins=num_bins,
    color='#1aa6b7',      # turquesa
    edgecolor='#2e4756',   # gris oscuro
    alpha=0.85
)

plt.xlabel('Edad', fontsize=10)
plt.ylabel('Número de casos', fontsize=10)
plt.title('Distribución de casos de COVID-19 por edad', fontsize=12)

ax = plt.gca()
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)

plt.grid(axis='y', linestyle='--', alpha=0.3)
plt.tight_layout()
plt.show()
```

Comenzamos verificando que los datos de la columna “Edad” sean de tipo numérico y, posteriormente, calculamos la edad mínima y máxima reportadas en el dataset. Con estos valores determinamos la cantidad de barras

del histograma, de modo que cada *bin* corresponda a una edad específica. La parte final del código se encarga de ajustar la estética del gráfico para obtener una visualización más clara y profesional, como se presenta en la Figura 16.

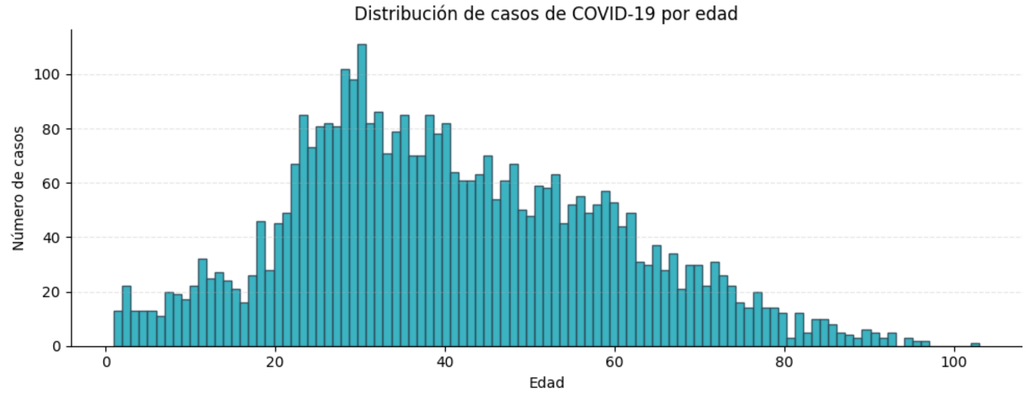


Figura 16. Distribución de casos de COVID-19 por edad

Como complemento a la información obtenida en la gráfica anterior, ahora podemos determinar el total de casos por rango de edad y su porcentaje respecto al total de casos reportados a la fecha. Para ello utilizamos un código que inicialmente define los *bins* para los rangos, por ejemplo: bins = [0, 20, 40, 60, 80, 100], y posteriormente agrupa los datos según estos intervalos utilizando la instrucción pd.cut.

Para nuestro caso:

- data['Edad'] es la columna original con valores numéricos.
- bins indica los límites de cada intervalo (0 a 20, 20 a 40, etc.).
- labels asigna un nombre a cada intervalo para facilitar su interpretación.
- right=False significa que los intervalos son cerrados por la izquierda y abiertos por la derecha; por ejemplo:
    - **0–20** incluye edades desde 0 hasta 19,
    - **20–40** incluye desde 20 hasta 39,y así sucesivamente.

El resultado es una nueva columna llamada **GrupoEdad**, donde cada persona queda clasificada en su rango correspondiente. Posteriormente, el código utiliza value_counts() para contar cuántos casos hay en cada intervalo y luego calcula su porcentaje respecto al total del dataset. Finalmente, estos resultados se organizan en un dataframe que muestra tanto el número de casos como el porcentaje asociado a cada grupo etario.

El código se presenta a continuación:

```
# Definir rangos de edad
bins = [0, 20, 40, 60, 80, 100]
labels = ['0-20', '20-40', '40-60', '60-80', '80-100']

# Crear una nueva columna con el grupo de edad
data['GrupoEdad'] = pd.cut(data['Edad'], bins=bins,
labels=labels, right=False)

# Calcular total de casos por grupo
casos_por_grupo = data['GrupoEdad'].value_counts().sort_index()

# Calcular porcentajes
porcentajes = (casos_por_grupo / len(data)) * 100

# Unir resultados en una sola tabla
resultado = pd.DataFrame({
    'Casos': casos_por_grupo,
    'Porcentaje (%)': porcentajes.round(2)
})
print(resultado)
```

Y se obtiene el siguiente resultado:

```
           Casos  Porcentaje (%)
GrupoEdad
0-20         408           10.67
20-40       1580           41.32
40-60       1171           30.62
60-80        566           14.80
80-100        98            2.56
```

Figura 17. Reporte casos por rango de edad y porcentaje

¿Por qué es importante este reporte?

Nos permite no solo identificar las edades con mayor número de contagios, sino también comprender las dinámicas de exposición asociadas a cada grupo etario. La mayor concentración de casos entre los **20 y 40 años** coincide con la población en edad laboral activa, lo que indica que el incremento de contagios en este grupo no se debe únicamente a la edad, sino a sus **mayores niveles de exposición**. Durante la pandemia, quienes debían desplazarse para trabajar, utilizar transporte público o desempeñar actividades presenciales estuvieron más expuestos al virus.

Para las autoridades de salud, esta distribución señala que las estrategias de control deben considerar **los patrones de movilidad y la actividad laboral**, priorizando medidas para trabajadores presenciales y sectores esenciales. En este sentido, la dinámica de contagio estuvo fuertemente influenciada por la exposición y no exclusivamente por la vulnerabilidad etaria.

3.1.11. Distribución de fallecidos por COVID-19 según edad

Continuando con nuestro EDA, es importante identificar cuántas personas han fallecido según su edad. Para ello, lo primero que hacemos es filtrar del dataset aquellas filas en las que la columna *Estado* corresponde a “Fallecido”. Posteriormente, calculamos nuevamente el número de *bins* a partir de la edad mínima y máxima registradas en este grupo, y utilizamos una gráfica tipo histograma para visualizar la distribución de edades de las personas fallecidas.

```
# Filtrar y crear copia para evitar warnings
fallecidos = data[data['Estado'] == 'Fallecido'].copy()

# Calcular número de bins (no necesariamente es el mismo
del grupo completo - todos los estados)
edad_min = int(fallecidos['Edad'].min())
edad_max = int(fallecidos['Edad'].max())
num_bins_fallecidos = edad_max - edad_min

plt.figure(figsize=(10,4))
plt.hist(
   fallecidos['Edad'].dropna(),
```

```
        bins=num_bins_fallecidos,
        color='#1aa6b7',
        edgecolor='#2e4756',
        alpha=0.85
    )

    plt.xlabel('Edad', fontsize=10)
    plt.ylabel('Número de casos', fontsize=10)
    plt.title('Distribución de fallecidos por COVID-19 según
    edad', fontsize=12)

    ax = plt.gca()
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)

    plt.grid(axis='y', linestyle='--', alpha=0.3)
    plt.tight_layout()
    plt.show()
```

Obteniendo:

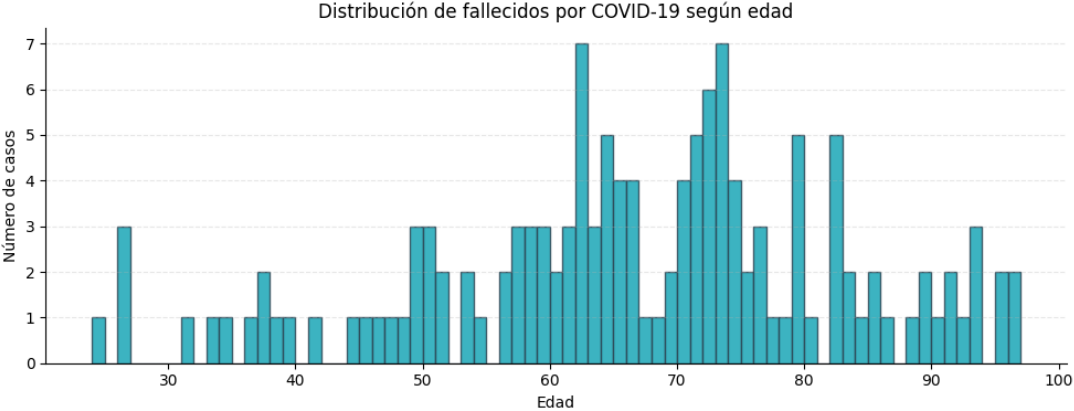


Figura 18. Distribución de fallecidos por COVID-19 según edad

Como complemento, calculamos la cantidad de fallecidos por grupo de

edad, y su porcentaje respecto al total de fallecidos, así:

```
# Clasificar en grupos de edad
fallecidos['GrupoEdad'] = pd.cut(
    fallecidos['Edad'],
    bins=bins,
    labels=labels,
    right=False
)

# Calcular cantidad de fallecidos por grupo
fallecidos_por_grupo = fallecidos['GrupoEdad'].value_counts().sort_
index()

# Calcular porcentaje respecto al total de fallecidos
porcentajes = (fallecidos_por_grupo / len(fallecidos)) * 100

# Crear tabla final
tabla_fallecidos = pd.DataFrame({
    'Fallecidos': fallecidos_por_grupo,
    'Porcentaje (%)': porcentajes.round(2)
})

print(tabla_fallecidos)
```

Obteniendo:

```
           Fallecidos  Porcentaje (%)
GrupoEdad
0-20                0            0.00
20-40              12            8.82
40-60              28           20.59
60-80              70           51.47
80-100             26           19.12
```

Figura 19. Distribución de fallecidos por COVID-19 según edad

Si comparamos los resultados de la Figura 19 con los de la Figura

17 nos damos cuenta que el mayor porcentaje de fallecidos se encuentra en el grupo de 60-80 años (51.47%), pese a que solamente representa el 14.8% de los casos reportados de COVID-19. Es decir, la probabilidad de muerte para ese grupo de edad es significativamente más alta que para cualquiera de los otros grupos.

3.1.12. Casos de contagio por Localidad, Fallecidos por Localidad, y Tasa de muerte por Localidad

Vamos ahora a realizar EDA por localidad. De alguna manera podemos "replicar" lo que hicimos con el dataset completo, pero ahora localidad por localidad. Iniciaremos por calcular la cantidad de contagiados por localidad, con el siguiente código:

```
casos_por_localidad = data['Localidad de residencia'].
value_counts().sort_values(ascending=True)

plt.figure(figsize=(10,6))
plt.barh(
    casos_por_localidad.index,
    casos_por_localidad.values,
    color='#1aa6b7',
    edgecolor='#2e4756',
    alpha=0.85
)
plt.xlabel('Número de casos', fontsize=10)
plt.ylabel('Localidad', fontsize=10)
plt.title('Número de casos de COVID-19 por localidad', fontsize=12)

ax = plt.gca()
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)

plt.grid(axis='x', linestyle='--', alpha=0.3)
plt.tight_layout()
plt.show()
```

Cuando utilizamos **value_counts()**, podemos rápidamente realizar un

conteo para cada una de las opciones de la columna a la que estamos haciendo referencia. Este método es mucho más eficiente que realizar filtros, pues en ese caso tendríamos que escribir una línea de código por cada localidad.

Como resultado, obtenemos:

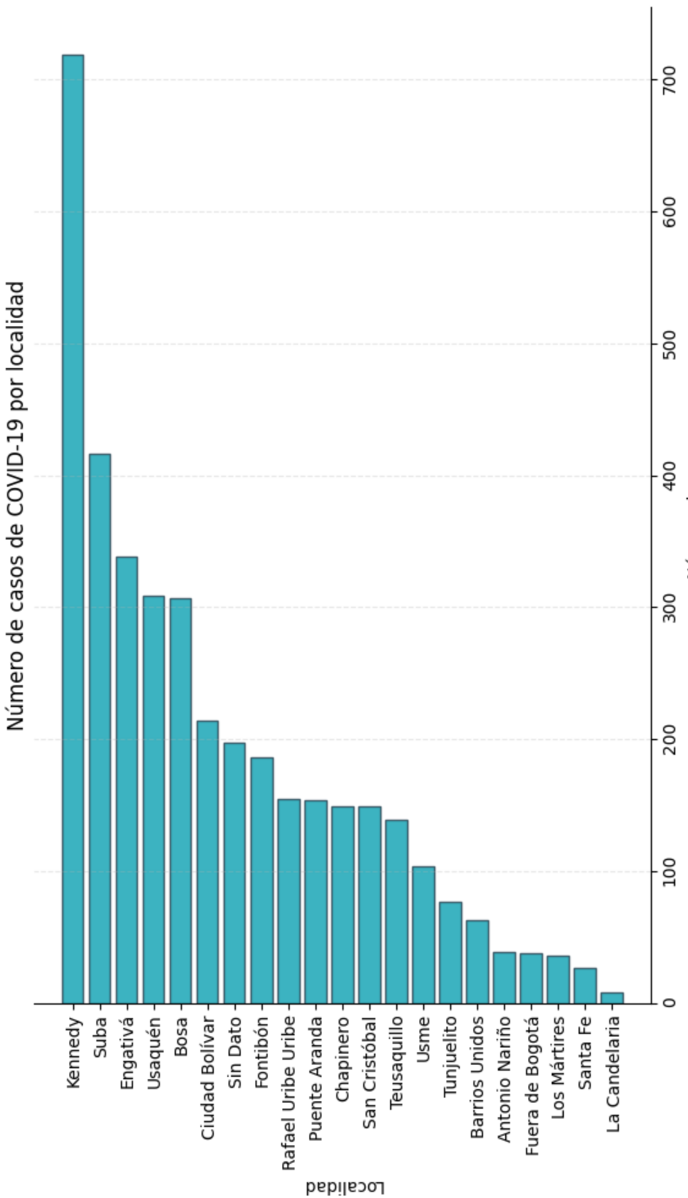


Figura 20. Casos de contagio por Localidad

Las localidades donde se concentra la mayor cantidad de contagios: **Kennedy**, seguida por **Suba** y **Engativá**. En el rango medio aparecen **Usaquén, Bosa, Ciudad Bolívar, Fontibón** y **San Cristóbal**, lo que evidencia una distribución amplia del virus en sectores con alta movilidad y actividad económica. Por su parte, localidades como **La Candelaria, Santa Fe, Los Mártires** y **Fuera de Bogotá** presentan un número significativamente menor de casos, lo cual puede estar relacionado con su menor densidad poblacional o con la naturaleza predominantemente administrativa o comercial de estas zonas.

Ahora, vamos a calcular cuántos fallecidos se tienen en cada una de las Localidades. Primero, realizamos un filtro para crear un subdataset correspondiente a “Fallecido” de la columna “Estado”. Posteriormente, con **value_counts()** calculamos los fallecidos por Localidad.

```
fallecidos = data[data['Estado'] == 'Fallecido'].copy()

# Contar fallecidos por localidad
fallecidos_por_localidad = fallecidos['Localidad de residencia'].value_
counts().sort_values(ascending=True)

# Crear gráfica
plt.figure(figsize=(10,6))
plt.barh(
    fallecidos_por_localidad.index,
    fallecidos_por_localidad.values,
    color='#1aa6b7',       # turquesa del libro
    edgecolor='#2e4756',   # gris oscuro
    alpha=0.85
)

plt.xlabel('Número de fallecidos', fontsize=10)
plt.ylabel('Localidad', fontsize=10)
plt.title('Número de fallecidos por COVID-19
según localidad', fontsize=12)

ax = plt.gca()
ax.spines['top'].set_visible(False)
```

```
    ax.spines['right'].set_visible(False)

    plt.grid(axis='x', linestyle='--', alpha=0.3)
    plt.tight_layout()
    plt.show()
```

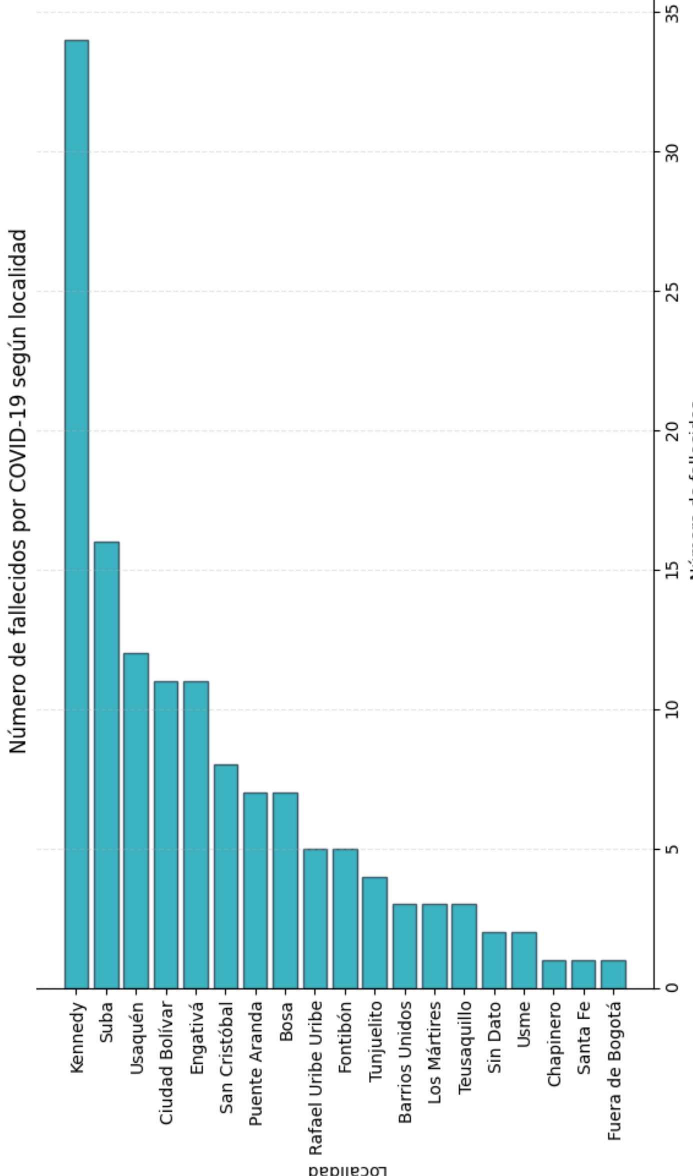


Figura 21. Número de fallecidos por COVID-19 según localidad

Localidades como Kennedy, Suba y Usaquén registran el mayor número de muertes, lo que puede relacionarse con su alta densidad poblacional, mayor volumen de movilidad diaria o características socioeconómicas que incrementan la exposición o dificultan el acceso oportuno a servicios de salud. En contraste, localidades como Santa Fe, Chapinero, Teusaquillo, Usme y Fuera de Bogotá presentan una mortalidad significativamente menor.

3.1.13. Tasa de muerte por Localidad

Finalizaremos nuestro EDA con la tasa de muerte por localidad. Esa tasa la calculamos de acuerdo con la siguiente ecuación matemática

$$Tasa\ de\ muerte\ por\ localidad = \frac{Número\ de\ fallecidos\ en\ la\ localidad}{Número\ total\ de\ contagios\ en\ la\ localidad} * 100$$

Utilizamos el siguiente código en Python:

```
# 1. Total de contagiados por localidad
contagiados_localidad = data['Localidad de residencia'].value_counts()

# 2. Total de fallecidos por localidad
fallecidos_localidad = data[data['Estado'] == 'Fallecido']['Localidad de residencia'].value_counts()

# 3. Unir ambas series en un solo dataframe
df_tasa = pd.DataFrame({
    'Contagiados': contagiados_localidad,
    'Fallecidos': fallecidos_localidad
})

# 4. Reemplazar NaN (localidades sin fallecidos) por 0
df_tasa = df_tasa.fillna(0)

# 5. Calcular la tasa de muerte
df_tasa['Tasa de muerte (%)'] = (df_tasa['Fallecidos']
/ df_tasa['Contagiados']) * 100
df_tasa['Tasa de muerte (%)'] = df_tasa['Tasa de muerte (%)'].round(2)
```

```
# 6. Ordenar de menor a mayor tasa
df_tasa = df_tasa.sort_values(by='Tasa de muerte (%)')

print(df_tasa)
```

Inicialmente, calculamos la **tasa de muerte por localidad**, combinando la información de contagiados y fallecidos reportados en el dataset. En primer lugar, se obtiene el total de contagiados por localidad utilizando value_counts(), que permite contar cuántos registros existen en cada categoría de la columna *Localidad de residencia*. Luego, se filtran los casos cuyo estado es "Fallecido" y se repite el proceso para obtener el número de fallecidos por localidad.

A continuación, ambas series se integran en un mismo *dataframe*, lo que facilita comparar el número total de contagiados con el número de fallecidos en cada zona. Dado que algunas localidades pueden no registrar fallecidos, se reemplazan los valores faltantes con cero mediante **fillna(0)** para evitar errores en los cálculos. Posteriormente, se calcula la tasa de muerte dividiendo el número de fallecidos entre el total de contagiados y multiplicando por 100, expresándola como porcentaje. Esta tasa se redondea a dos decimales para mejorar su presentación. Finalmente, el *dataframe* se ordena de menor a mayor tasa, lo que permite identificar rápidamente qué localidades presentan una mortalidad proporcional más alta.

Y obtenemos:

Localidad de residencia	Contagiados	Fallecidos	Tasa de muerte (%)
Antonio Nariño	39	0.0	0.00
La Candelaria	8	0.0	0.00
Chapinero	149	1.0	0.67
Sin Dato	197	2.0	1.02
Usme	104	2.0	1.92
Teusaquillo	139	3.0	2.16
Bosa	307	7.0	2.28
Fuera de Bogotá	38	1.0	2.63
Fontibón	186	5.0	2.69
Rafael Uribe Uribe	155	5.0	3.23
Engativá	338	11.0	3.25
Santa Fe	27	1.0	3.70
Suba	416	16.0	3.85
Usaquén	309	12.0	3.88
Puente Aranda	154	7.0	4.55
Kennedy	719	34.0	4.73
Barrios Unidos	63	3.0	4.76
Ciudad Bolívar	214	11.0	5.14
Tunjuelito	77	4.0	5.19
San Cristóbal	149	8.0	5.37
Los Mártires	36	3.0	8.33

Figura 22. Tasa de muerte por Localidad

¿Para qué sirve este reporte?

Este reporte permite identificar la **tasa de muerte por localidad**, un indicador mucho más informativo que observar únicamente el número de contagiados o de fallecidos por separado. Al relacionar ambos valores, es posible determinar en qué zonas de la ciudad el impacto del COVID-19 fue proporcionalmente más alto. Por ejemplo, localidades como **Los Mártires**, **San Cristóbal** y **Tunjuelito** presentan tasas de muerte más elevadas, a pesar de no ser las que registran más contagios. Esto sugiere posibles diferencias en acceso a servicios de salud, condiciones socioeconómicas, comorbilidades o tiempos de atención. En contraste, localidades con un alto número de contagios como **Kennedy**, **Suba** y **Engativá** muestran una tasa de mortalidad proporcionalmente menor. Este análisis permite a las autoridades orientar estrategias de intervención focalizada, asignación de recursos y políticas de prevención según la vulnerabilidad específica de cada territorio.

3.2. Caso de estudio 2: EDA y FE al Bank Churn dataset

Para este segundo caso de estudio, cambiaremos de dataset a una temática distinta. El dataset anterior correspondía al sector salud, y el de este caso al sector bancario. Aunque realizaremos algunas acciones de Análisis Exploratorio de Datos (EDA), el mayor énfasis de este caso estará en la Ingeniería de Características (Feature Engineering, FE).

La información de este dataset es:

- **RowNumber**: Número de fila dentro del dataset. No aporta información analítica; únicamente indica la posición del registro.
- **CustomerId**: Identificador único asignado a cada cliente. Sirve para distinguir un cliente de otro, pero no se utiliza como variable predictiva.
- **Surname**: Apellido del cliente. En la mayoría de modelos, este atributo no aporta información relevante y suele eliminarse para evitar sesgos.
- **CreditScore**: Puntaje crediticio del cliente. Un valor numérico que indica la solvencia o comportamiento esperado frente al crédito. Valores más altos representan mayor estabilidad financiera.
- **Geography**: País de residencia del cliente (por ejemplo: France, Spain, Germany). Es una variable categórica que permite analizar diferencias de comportamiento entre regiones.
- **Gender**: Género del cliente (Male / Female). Variable categórica usada en algunos análisis demográficos.
- **Age**: Edad del cliente. Es una de las variables más influyentes en el análisis de abandono bancario (churn).
- **Tenure**: Número de años que el cliente lleva vinculado al banco. Permite conocer la antigüedad y evaluar la fidelidad.
- **Balance**: Saldo actual de la cuenta del cliente. Un valor numérico que representa los fondos disponibles en el banco.
- **NumOfProducts**: Número de productos que el cliente tiene con el banco (por ejemplo, tarjetas, cuentas, créditos). Indica el nivel de relación comercial.

- **HasCrCard**: Indica si el cliente posee tarjeta de crédito.

  1 = Sí

  0 = No

- **IsActiveMember**: Indica si el cliente es considerado activo dentro del banco.

  1 = Activo

  0 = Inactivo

- **EstimatedSalary**: Salario estimado del cliente. Es una variable numérica que representa su nivel de ingresos aproximado.

- **Exited**: Variable objetivo del dataset. Indica si el cliente abandonó el banco (churn).

  1 = El cliente se fue

  0 = El cliente permaneció

Realizamos importe de librerías, y pre-visualización, como vimos en el capítulo 3.1. Nuestro dataset contiene 14 columnas y 10.002 filas.

	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exite
0	1	15634602	Hargrave	619	France	Female	42.0	2	0.00	1	1.0	1.0	101348.88	[illegible]
1	2	15647311	Hill	608	Spain	Female	41.0	1	83807.86	1	0.0	1.0	112542.58	[illegible]
2	3	15619304	Onio	502	France	Female	42.0	8	159660.80	3	1.0	0.0	113931.57	[illegible]
3	4	15701354	Boni	699	France	Female	39.0	1	0.00	2	0.0	0.0	93826.63	[illegible]
4	5	15737888	Mitchell	850	Spain	Female	43.0	2	125510.82	1	NaN	1.0	79084.10	[illegible]
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
9997	9998	15584532	Liu	709	France	Female	36.0	7	0.00	1	0.0	1.0	42085.58	[illegible]
9998	9999	15682355	Sabbatini	772	Germany	Male	42.0	3	75075.31	2	1.0	0.0	92888.52	[illegible]
9999	9999	15682355	Sabbatini	772	Germany	Male	42.0	3	75075.31	2	1.0	0.0	92888.52	[illegible]
10000	10000	15628319	Walker	792	France	Female	28.0	4	130142.79	1	1.0	0.0	38190.78	[illegible]
10001	10000	15628319	Walker	792	France	Female	28.0	4	130142.79	1	1.0	0.0	38190.78	[illegible]

10002 rows × 14 columns

Figura 23. Pre-visualización dataset Bank churn

3.2.1. Análisis Exploratorio de Datos Bank churn dataset

Iniciamos con el tipo de dato de cada columna del dataset, con la siguiente instrucción:

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10002 entries, 0 to 10001
Data columns (total 14 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   RowNumber        10002 non-null  int64
 1   CustomerId       10002 non-null  int64
 2   Surname          10002 non-null  object
 3   CreditScore      10002 non-null  int64
 4   Geography        10001 non-null  object
 5   Gender           10002 non-null  object
 6   Age              10001 non-null  float64
 7   Tenure           10002 non-null  int64
 8   Balance          10002 non-null  float64
 9   NumOfProducts    10002 non-null  int64
 10  HasCrCard        10001 non-null  float64
 11  IsActiveMember   10001 non-null  float64
 12  EstimatedSalary  10002 non-null  float64
 13  Exited           10002 non-null  int64
dtypes: float64(5), int64(6), object(3)
memory usage: 1.1+ MB
```

Figura 24. Tipo de datos Bank churn dataset

Es recomendable iniciar la tarea de EDA con este paso, dado que nos permite identificar el tipo de datos de cada columna, así como la cantidad de datos faltantes en cada una de ellas. Cuando nuestro alcance como científico de datos supera el EDA y queremos realizar modelamiento, necesitamos que todas las columnas sean numéricas o booleanas, por lo que debemos aplicar ingeniería de características en las que no lo sean. Para este caso de estudio, tenemos tres columnas que no son numéricas.

Por otro lado, podemos deducir que existen columnas con datos faltantes, dado que no todas las columnas tienen los 10.002 valores, algunas tienen un dato faltante.

Como segundo paso de nuestro EDA vamos a conocer el comportamiento de la variable de salida, que corresponde a Exited. Queremos saber si nuestro dataset es balanceado o no, y cuantos casos corresponden a cada categoría. Este problema corresponde a una **clasificación binaria**.

Escribimos el siguiente código en Python.

```
from matplotlib import pyplot as plt
import seaborn as sns

# Agrupar y contar
counts = data.groupby('Exited').size()

# Crear gráfico
plt.figure(figsize=(6,4))
ax = counts.plot(kind='barh', color=sns.palettes.mpl_palette('Dark2'))
ax.spines[['top', 'right']].set_visible(False)

# Añadir las cantidades encima de cada barra
for i, value in enumerate(counts):
    plt.text(value + 1,        # posición x (ligeramente a la derecha del
valor)
             i,                # posición y (alineada con la barra)
             str(value),       # texto que se muestra
             va='center',      # alineación vertical
             fontsize=10)

plt.xlabel('Número de casos')
plt.ylabel('Exited')
plt.title('Distribución de clientes según Exited')

plt.tight_layout()
plt.show()
```

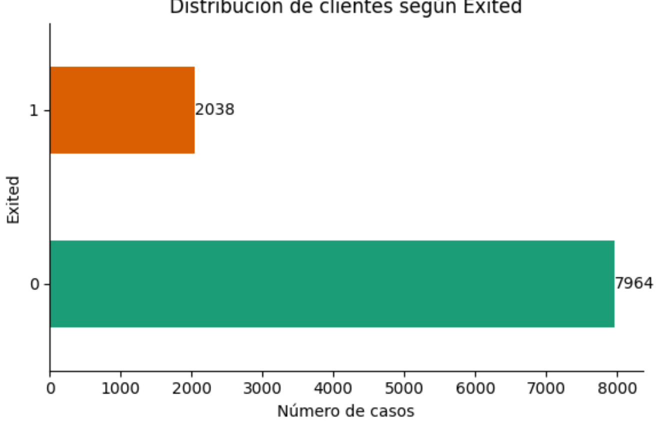


Figura 25. Distribución de casos Bank churn dataset

Y la proporción por clase

```
# PROPORCIÓN POR CLASE
data['Exited'].value_counts(normalize=True)
```

De acuerdo con la variable *Exited*, se observa que el 79,63 % de los clientes permanece activo, mientras que el 20,37 % ha cancelado su cuenta.

3.2.2. Transformación de columnas del dataset: de object a integer

Vamos a empezar con la columna "Gender". Inicialmente, verificamos la cantidad de opciones dentro de esta columna, así:

```
# CANTIDAD DE GENEROS DENTRO DEL DATASET
num_generos_distintos = data['Gender'].nunique()
print(f"Número de generos distintos: {num_generos_distintos}")
```

Y nos aparece que hay 2 géneros distintos. De la pre-visualización podemos observar que las dos opciones son: Female y Male.

Posteriormente, convertimos esta columna a entero realizando una asignación de "1" cuando es Male y de "0" cuando es Female. Adicionalmente contamos la cantidad de no nulos y lo visualizamos.

```
# CONVERTIR LA COLUMNA "GENDER" DE OBJECT A ENTERO
```

```
data['Gender'] = data['Gender'].apply(lambda x: 1 if x == 'Male' else 0)
no_nulos_genero = data['Gender'].count()
print(no_nulos_genero)
```

	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exited
0	1	15634602	Hargrave	619	France	0	42.0	2	0.00	1	1.0	1.0	101348.88	1
1	2	15647311	Hill	608	Spain	0	41.0	1	83807.86	1	0.0	1.0	112542.58	0
2	3	15619304	Onio	502	France	0	42.0	8	159660.80	3	1.0	0.0	113931.57	1
3	4	15701354	Boni	699	France	0	39.0	1	0.00	2	0.0	0.0	93826.63	0
4	5	15737888	Mitchell	850	Spain	0	43.0	2	125510.82	1	NaN	1.0	79084.10	0
5	6	15574012	Chu	645	Spain	1	44.0	8	113755.78	2	1.0	0.0	149756.71	1
6	7	15592531	Bartlett	822	NaN	1	50.0	7	0.00	2	1.0	1.0	10062.80	0
7	8	15656148	Obinna	376	Germany	0	29.0	4	115046.74	4	1.0	0.0	119346.88	1
8	9	15792365	He	501	France	1	44.0	4	142051.07	2	0.0	NaN	74940.50	0
9	10	15592389	H?	684	France	1	NaN	2	134603.88	1	1.0	1.0	71725.73	0

Figura 26. Columna “Gender” transformada, dataset Bank churn.

Ahora, procedemos con la columna "Geograhphy". De nuevo, lo primero que hacemos es revisar los valores al interior de la columna.

```
# CANTIDAD DE PAISES DENTRO DEL DATASET

num_paises_distintos = data['Geography'].nunique()
print(f"Número de paises distintos: {num_paises_distintos}")

# CANTIDAD DE VECES QUE APARECE CADA PAIS

vu_country_counts = data['Geography'].value_counts()
print(vu_country_counts)
```

Dado que no existe jerarquía entre los diferentes valores, la transformación adecuada para este caso corresponde a *one-hot ecoding*. Como son tres países diferentes, entonces se crearán tres nuevas columnas donde solamente "se enciende" una columna a la vez por registro.

La transformación y la pre-visualización la realizados con el siguiente código:

```
# CONVERTIR LA COLUMNA "GEOGRAPHY",
UTILIZANDO ONE-HOT ENCODING
data = pd.get_dummies(data, columns=['Geography'], prefix='Country')
# Mostrar el resultado
data.head(registro)
```

Geography
France
Spain
France
France
Spain
Spain
NaN
Germany
France
France

Country_France	Country_Germany	Country_Spain
True	False	False
False	False	True
True	False	False
True	False	False
False	False	True
False	False	True
False	False	False
False	True	False
True	False	False
True	False	False

Figura 27. Transformación columna "Geography", dataset Bank churn.

En la parte izquierda de la Figura 27 apreciamos la columna "Geography" la cual ha sido eliminada y reemplazada por tres columnas, como aparece en la parte derecha de la misma figura. Por ejemplo, para *France* se obtiene el vector [True, False, False]; mientras que, para *Spain* se tendrá el valor [False, False, True].

Finalmente, procedemos con la columna "Surname". En este caso, la cantidad de opciones diferentes es bastante significante:

```
# CANTIDAD DE APELLIDOS DIFERENTES DEL DATASET

num_apellidos_distintos = data['Surname'].nunique()
print(f"Número de apellidos distintos: {num_apellidos_distintos}")
```

Número de apellidos distintos: 2932

A su vez, cada apellido se repite un número de veces que no siempre es el mismo (como es de esperarse):

```
# CANTIDAD DE VECES QUE APARECE CADA APELLIDO

vu_surname_counts = data['Surname'].value_counts()
print(vu_surname_counts)
```

```
Surname
Smith      32
Martin     29
Walker     29
Scott      29
Brown      26
           ..
Hull        1
Sturdee     1
Flannagan   1
Dwyer       1
Corby       1
Name: count, Length: 2932, dtype: int64
```

Teniendo en cuenta lo anterior, en lugar de aplicar técnicas tradicionales como *one-hot encoding*, que generarían una elevada dimensionalidad, emplearemos una estrategia de codificación basada en la variable objetivo (target encoding). Para ello, calculamos la **probabilidad condicional** de abandono del cliente dada cada categoría del apellido, es decir,

$$P(Exited = 1 \mid Surname)$$

y este valor se asigna posteriormente a cada registro del dataset como una nueva variable numérica.

Esta transformación permite conservar información estadísticamente relevante sobre el comportamiento de los clientes, reduciendo al mismo tiempo la complejidad del espacio de características. Sin embargo, es importante advertir que esta técnica debe aplicarse cuidadosamente dentro del conjunto de entrenamiento para evitar problemas de *data leakage*.

```
# 1. Calcular la probabilidad condicional P(Exited=1 | Surname)
apellido_churn_prob = data.groupby('Surname')['Exited'].mean()

# 2. Mapear esa probabilidad a cada fila del dataframe original
data['Surname_Exited_Prob'] = data['Surname'].map(apellido_churn_
prob)

# 3.  Re-ubicar la columna "Surname_Exited_Prob" en la cuarta
 posición
col = data.pop('Surname_Exited_Prob')
data.insert(3, 'Surname_Exited_Prob', col)
```

Surname	Surname_Exited_Prob
Hargrave	1.000000
Hill	0.117647
Onio	0.250000
Boni	0.214286
Mitchell	0.100000
Chu	0.136364
Bartlett	0.250000
Obinna	0.500000
He	0.277778
H?	0.052632

Figura 28. Creación de nueva columna a partir de probabilidad condicional de Surname.

Posteriormente eliminamos la columna “Surname” (dado que ahora nos quedaremos con “Surname_Exited_Prob), así:

```
data = data.drop(columns=['Surname'])
```

3.2.3. Imputación de datos al interior de una columna del dataframe

Hasta este punto, ya hemos transformado las tres columnas de tipo *object* del dataset. Las demás columnas son numéricas, de tipo int64 o float64. No obstante, aún existen datos faltantes en algunas de ellas, por lo que es necesario realizar un proceso de imputación.

Para identificar en qué columnas faltan datos, utilizamos:

```
data.isnull().mean()
```

RowNumber	0.0000
CustomerId	0.0000
CreditScore	0.0000
Gender	0.0000
Age	0.0001
Tenure	0.0000
Balance	0.0000
NumOfProducts	0.0000
HasCrCard	0.0001
IsActiveMember	0.0001
EstimatedSalary	0.0000
Exited	0.0000
Country_France	0.0000
Country_Germany	0.0000
Country_Spain	0.0000
Surname_Exited_Prob	0.0000

Figura 29. Creación de nueva columna a partir de probabilidad condicional de Surname.

De acuerdo con el resultado anterior, tres columnas requieren proceso de imputación:
"Age", "HasCrCard" y "IsActiveMember".

Imputación en la columna "Age"

Para esta variable numérica utilizamos como estrategia el **promedio (media)** de los demás valores para reemplazar el dato faltante:

```
# IMPUTAR EN COLUMNA AGE
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy='mean')
```

```
# Aplicar imputación solo a 'col1'
data[['Age']] = imputer.fit_transform(data[['Age']])
```

Imputación en la columna "HasCrCard"

Esta columna solo posee dos valores posibles: **1** si el cliente tiene tarjeta de crédito, o **0** si no la tiene. Por lo tanto, no resulta adecuado utilizar una métrica como el promedio. En este caso se emplea como estrategia el **valor más frecuente (moda)**:

```
# Crear el imputador con estrategia de más frecuente
imputer = SimpleImputer(strategy='most_frequent')

# Aplicar imputación solo a 'col1'
data[['HasCrCard']] = imputer.fit_transform(data[['HasCrCard']])
```

Imputación en la columna "IsActiveMember"

Esta columna presenta un comportamiento similar a "HasCrCard", por lo que utilizamos la misma estrategia basada en la moda:

```
# Crear el imputador con estrategia de más frecuente
imputer = SimpleImputer(strategy='most_frequent')

# Aplicar imputación solo a 'col1'
data[['IsActiveMember']] = imputer.fit_
transform(data[['IsActiveMember']])
```

3.2.4. Eliminar columnas innecesarias del dataset

Ya estamos muy próximos a poder realizar el modelamiento de nuestro dataset. Sin embargo, aún debemos ejecutar un paso previo: eliminar columnas innecesarias. Aunque las columnas **"RowNumber"** y **"CustomerId"** son numéricas, estas no aportan información útil al proceso de modelado, dado que la primera corresponde únicamente a una secuencia, y la segunda es el identificador único de cada cliente, sin valor predictivo.

Para ello, utilizamos el siguiente código:

```
data = data.drop(columns=["RowNumber", "CustomerId"])
```

De forma opcional, procederemos a reordenar las columnas del dataset. Como buena práctica, la variable de salida suele ubicarse en la primera columna o en la última columna del daraset. En este caso, la ubicaremos en la primera columna.

```
col = data.pop('Exited')
data.insert(0, 'Exited', col)
data.head(10)
```

	Exited	Surname_Exited_Prob	CreditScore	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Country_France	Country_Germany	Country_Spain
0	1	1.000000	619	0	42.000000	2	0.00	1	1.0	1.0	101348.88	True	False	False
1	0	0.117647	608	0	41.000000	1	83807.86	1	0.0	1.0	112542.58	False	False	True
2	1	0.250000	502	0	42.000000	8	159660.80	3	1.0	0.0	113931.57	True	False	False
3	0	0.214286	699	0	39.000000	1	0.00	2	0.0	0.0	93826.63	True	False	False
4	0	0.100000	850	0	43.000000	2	125510.82	1	1.0	1.0	79084.10	False	False	True
5	1	0.136364	645	1	44.000000	8	113755.78	2	1.0	0.0	149756.71	False	False	True
6	0	0.250000	822	1	50.000000	7	0.00	2	1.0	1.0	10062.80	False	False	False
7	1	0.500000	376	0	29.000000	4	115046.74	4	1.0	0.0	119346.88	False	True	False
8	0	0.277778	501	1	44.000000	4	142051.07	2	0.0	1.0	74940.50	True	False	False
9	0	0.052632	684	1	38.922311	2	134603.88	1	1.0	1.0	71725.73	True	False	False

Figura 30. Dataset Bank churn con ingeniería de características.

Por último, vamos a verificar que todas las columnas sean numéricas y/o booleanas, y que no existan datos faltantes, es decir que cada columna contenga 10.002 datos no-nulos:

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10002 entries, 0 to 10001
Data columns (total 14 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   Exited              10002 non-null  int64
 1   Surname_Exited_Prob 10002 non-null  float64
 2   CreditScore         10002 non-null  int64
 3   Gender              10002 non-null  int64
 4   Age                 10002 non-null  float64
 5   Tenure              10002 non-null  int64
 6   Balance             10002 non-null  float64
 7   NumOfProducts       10002 non-null  int64
 8   HasCrCard           10002 non-null  float64
 9   IsActiveMember      10002 non-null  float64
 10  EstimatedSalary     10002 non-null  float64
 11  Country_France      10002 non-null  bool
 12  Country_Germany     10002 non-null  bool
 13  Country_Spain       10002 non-null  bool
dtypes: bool(3), float64(6), int64(5)
memory usage: 889.0 KB
```

Figura 31. Información del dataset posterior a FE.

¡Ahora sí, nuestro dataset está listo para realizar modelamiento!

3.2.5. Modelamiento

Entramos a la fase final. Nuestro dataset de Banck Churn no está inicialmente listo para realizar un modelamiento. Tenía columnas de tipo objeto, datos faltantes en columnas numéricas, y algunas columnas que no aportaban información para el modelo. Hemos realizado tareas de limpieza de datos (ej. imputación) y conversión o transformación de variables. Nuestro dataset ya cumple con las condiciones para que puede ingresarse a un modelo de clasificación binario. Para nuestro caso, vamos a seleccionar un árbol de decisión, el cual el mismo identifica los "patrones" para la toma de decisión en relación a si el cliente seguirá activo y se retirará.

Inicialmente importamos las librerías de trabajo:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, confusion_matrix,
 classification_report
import matplotlib.pyplot as plt
from sklearn.tree import plot_tree
```

Posteriormente, separamos los *features* y la salida, así:

```
# ◈ 1. Separar features (X) y variable objetivo (y)
X = data.drop(columns=["Exited"])  # "Diagnostico" es la columna que
indica si es COVID o H1N1
y = data["Exited"]
```

A continuación, separamos el *dataset* en dos particiones: entrenamiento y validación. Típicamente, se utilizan del 70% - 80% para entrenamiento, y el resto para validación (es decir, del 20% al 30%).

```
# ◈ 2. Dividir en entrenamiento (80%) y validación (20%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
 random_state=42)
```

Como tercer paso, creamos y entrenamos el árbol de decisión. Para este caso hemos definido un árbol de dos niveles

```
# ◈ 3. Crear y entrenar el Árbol de Decisión
model = DecisionTreeClassifier(criterion="entropy", max_depth=2,
 random_state=42)
model.fit(X_train, y_train)
```

Como cuarto paso, realizamos las predicciones con los datos de validación:

```
# ◈ 4. Hacer predicciones
y_pred = model.predict(X_test)
```

Como quinto paso, evaluamos el modelo, mostramos la matriz de confusión e imprimimos el reporte

```
# ◈ 5. Evaluar el modelo, mostrar matriz de confusión e imprimir reporte
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")

print("\nMatriz de Confusión:")
print(confusion_matrix(y_test, y_pred))

print("\nReporte de Clasificación:")
print(classification_report(y_test, y_pred))
```

Finalmente, visualizamos el árbol de decisión generado

```
# ◈ 6. Visualizar el árbol de decisión
plt.figure(figsize=(12, 8))
plot_tree(model, feature_names=X.columns, class_names = ['Cliente Activo', 'Cliente Retirado'], filled=True)
plt.show()
```

```
Accuracy: 0.81

Matriz de Confusión:
[[1352  247]
 [ 137  265]]

Reporte de Clasificación:
              precision    recall  f1-score   support

           0       0.91      0.85      0.88      1599
           1       0.52      0.66      0.58       402

    accuracy                           0.81      2001
   macro avg       0.71      0.75      0.73      2001
weighted avg       0.83      0.81      0.82      2001
```

Figura 32. Matriz de confusión del Modelo 1: dos niveles de profundidad.

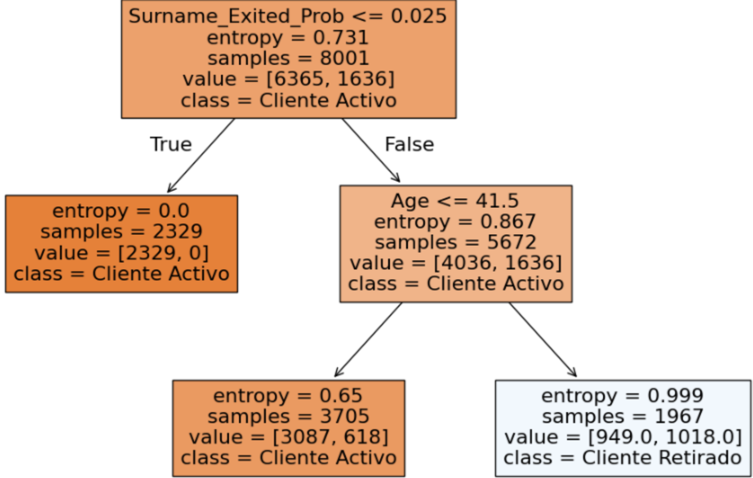


Figura 33. Árbol de decisión del Modelo 1: dos niveles de profundidad.

El árbol de decisión comienza analizando la variable Surname_Exited_Prob, que representa la probabilidad de abandono asociada al apellido del cliente.

- **Primera pregunta del árbol:**

  Si el valor de Surname_Exited_Prob **es menor o igual a 0.025**, el modelo concluye directamente que el cliente es un **Cliente Activo**. Esto significa que, históricamente, los clientes con esta característica casi nunca abandonan el banco.

- **Si** Surname_Exited_Prob **es mayor a 0.025**, el modelo realiza una segunda pregunta utilizando la variable Age:

  - Si la **edad del cliente es menor o igual a 41.5 años**, el modelo clasifica nuevamente al cliente como **Cliente Activo**.

  - Si la **edad es mayor a 41.5 años**, el modelo clasifica al cliente como **Cliente Retirado**.

Con estos dos niveles de profundidad, obtuvimos un *accuracy* del 81%. Vemos en la matriz de confusión que, de los 402 casos de retiro, el modelo identifica correctamente 265. Aún tenemos un margen de mejora importante, por lo que aumentaremos el nivel de profundidad.

```
Accuracy: 0.87

Matriz de Confusión:
[[1504   95]
 [ 156  246]]

Reporte de Clasificación:
              precision    recall  f1-score   support

           0       0.91      0.94      0.92      1599
           1       0.72      0.61      0.66       402

    accuracy                           0.87      2001
   macro avg       0.81      0.78      0.79      2001
weighted avg       0.87      0.87      0.87      2001
```

Figura 34. Matriz de confusión del Modelo 2: cuatro niveles de profundidad.

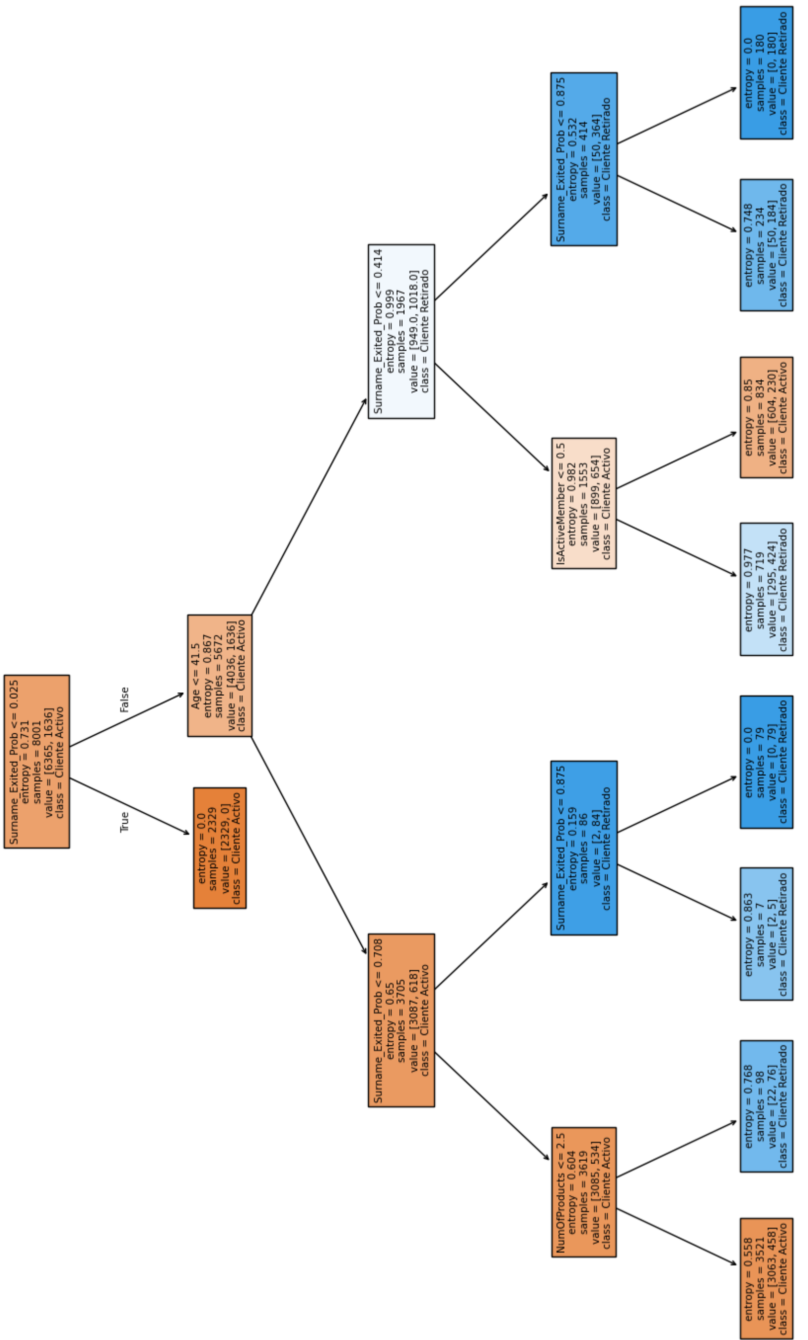


Figura 35. Árbol de decisión del Modelo 2: cuatro niveles de profundidad.

Con cuatro niveles de profundidad, pasamos de un

acc = 0.81 a un acc = 0.87. No obstante, de los casos de retiro, ahora identifica correctamente sólo 246 casos vs los 265 del modelo inicial.

Entonces, ¿cómo sabemos cuál modelo seleccionar? Esto depende de cuál sea la prioridad del banco.

- Si el objetivo es identificar a los clientes que están próximos a retirarse para intentar convencerlos de no hacerlo, entonces la métrica más importante a comparar es el Recall de la clase 1 (clientes retirados). En el primer modelo se obtuvo un *Recall* de 0.66, mientras que en el segundo modelo se obtuvo un valor de 0.61. Si yo fuese el gerente del banco, me quedaría con el **Modelo 1**, de acuerdo con este criterio, aun cuando esto implique contactar también a algunos clientes que en realidad sí desean continuar.
- Pero si, la prioridad del banco no es contactar a todos los posibles clientes que podrían retirarse, sino evitar al máximo contactar a clientes que en realidad no se van a ir, ya que cada contacto implica un costo económico (llamadas, correos, personal, campañas, etc.), entonces la mejor opción es **Modelo 2**, dado que ahora debemos enfocarnos en *Precision* de la clase 1, que en el Modelo 1 fue de 0.52, mientras que la del Modelo 2 fue de 0.72.

SERIES DE TIEMPO

En este capítulo trabajaremos con datos estructurados, pero que tienen una condición especial: la dependencia de sus registros en relación con la línea de tiempo. ¿Qué significa esto? Que lo que sucede en un momento específico está fuertemente relacionado con lo que aconteció en tiempos anteriores. En este tipo de datasets existe una columna particular que contiene la información de la fecha (e incluso la hora) del registro, y esta información se convierte en la base para la construcción de nuevas características y la aplicación de técnicas de ingeniería de características.

Una serie de tiempo es, formalmente, un conjunto de observaciones registradas en instantes sucesivos, donde el orden temporal es esencial para comprender el comportamiento de la variable estudiada. A diferencia de los datasets estructurados tradicionales (en los que las filas pueden reordenarse sin alterar su significado), en una serie de tiempo cada valor depende del momento en que ocurrió y, en muchos casos, de los valores que le precedieron.

En términos simples, una serie de tiempo nos permite observar cómo evoluciona una variable a través del tiempo. Ejemplos comunes incluyen:

- El número diario de casos confirmados de COVID-19.
- La temperatura registrada por hora en una ciudad.
- El precio del dólar minuto a minuto.
- El número de ventas mensuales de un producto.

- La frecuencia cardíaca registrada segundo a segundo por un dispositivo wearable.

Este tipo de información es fundamental para tareas como analizar patrones, identificar tendencias, detectar anomalías y predecir valores futuros mediante técnicas de pronóstico (*forecasting*).

Las series de tiempo poseen características particulares que las diferencian de otros tipos de datos:

- Secuencia temporal obligatoria: no se puede alterar el orden de los registros sin perder información esencial.
- Dependencia entre observaciones: un valor puede influir sobre los siguientes.
- Contexto histórico: comprender el pasado es clave para interpretar o predecir el futuro.

Por estas razones, el análisis de series de tiempo requiere métodos específicos que respeten la naturaleza temporal del fenómeno. Cada decisión, desde la visualización inicial hasta el modelamiento, debe apoyarse en esta estructura para obtener resultados coherentes y útiles.

Nota aclaratoria sobre el dataset de COVID-19. Aunque el daraset de casos de COVID-19 de Bogotá fue abordado en el Capítulo 3, constituye, estrictamente hablando, una serie de tiempo. La razón de trabajarlo como una dataset estructurado básico es que nos permitió introducir de manera gradual los conceptos fundamentales de limpieza de datos y EDA, en un contexto real y familiar para el lector.

4.1. Conceptos Básicos de Series de Tiempo

Antes de iniciar nuestros ejemplos con Series de Tiempo, es necesario conocer algunos concentos básicos, los cuales se presentan a continuación.

4.1.1. Componentes fundamentales de una Serie de Tiempo

Una serie de tiempo puede analizarse como la combinación de varios

componentes que describen distintos tipos de comportamiento temporal. Identificar estos componentes permite comprender mejor la dinámica de los datos y constituye un paso clave antes de cualquier proceso de modelamiento o pronóstico.

De forma general, una serie de tiempo puede descomponerse en los siguientes elementos:

- **Tendencia**: representa el comportamiento global de largo plazo de la serie. Indica si la variable presenta una dirección creciente, decreciente o estable a lo largo del tiempo. Como ejemplo, se puede mencionar el aumento progresivo del precio de venta de una criptomoneda.

- **Estacionalidad**: corresponde a patrones que se repiten cada cierto periodo de tiempo, por ejemplo, días, semanas o meses. Un ejemplo es el incremento de ventas en el mes de diciembre, que se repite cada año.

- **Ciclo**: a diferencia de la estacionalidad, en este caso las fluctuaciones que se repiten en el tiempo no tienen una duración constante, y su inicio y fin no son perfectamente predecibles. Un ejemplo son los ciclos de “burbuja” y recesión en un país.

- **Ruido o componente irregular**: corresponde a variaciones aleatorias, errores de medición o perturbaciones imprevistas que no pueden explicarse por la tendencia, la estacionalidad ni el ciclo.

Nota aclaratoria 1: en este capítulo se trabaja bajo el supuesto de una descomposición aditiva de la serie de tiempo, una aproximación común en el análisis exploratorio; sin embargo, en aplicaciones reales pueden existir descomposiciones multiplicativas o mixtas.

Nota aclaratoria 2: en este capítulo, el término estacionariedad se utiliza en el sentido de estacionariedad débil o de segundo orden, en la cual la media y la varianza de la serie permanecen constantes en el tiempo y la covarianza depende únicamente del retardo temporal.

4.1.2. Clasificación de Series de Tiempo

Las series de tiempo pueden clasificarse de distintas formas según la cantidad de variables involucradas, la naturaleza de los datos, su comportamiento estadístico y la regularidad con la que son registradas. Esta clasificación es importante porque condiciona las técnicas de análisis y los modelos que pueden aplicarse (Ver Figura 36).

Figura 36. Clasificación Series de Tiempo.

A continuación, se presentan los tipos más comunes de series de tiempo.

Series de tiempo de una y varias variables

- Serie de tiempo de una sola variable: está conformada por una sola variable que se observa a lo largo del tiempo. Por ejemplo, el precio diario del dólar, la temperatura por hora o la cantidad de ventas mensuales.

- Serie de tiempo multivariada: está compuesta por dos o más variables observadas simultáneamente en el tiempo. Por ejemplo, un dataset de criptomoneda con las siguientes variables: precio de apertura, precio de cierre, cantidad de transacciones, precio más alto. Las series multivariadas permiten analizar relaciones entre variables, pero también presentan una mayor complejidad en el modelamiento.

Series de tiempo continuas y discretas

- Series continuas: toman valores en "cualquier instante" de tiempo.

- Series discretas: la mayoría de los datasets utilizados en ciencia de datos corresponden a este tipo. Registra valores específicos del fenómeno observado, con espaciamientos, por ejemplo, de minutos, horas, o días.

Series de tiempo estacionarias y no estacionarias

- Serie estacionaria: presenta media constante, varianza constante y una estructura de correlación que no cambia en el tiempo. Este tipo de series es especialmente importante porque muchos modelos estadísticos clásicos asumen estacionariedad.

- Serie no estacionaria: presenta cambios en su media, varianza o estructura de dependencia a lo largo del tiempo. La mayoría de las series reales pertenecen a esta categoría y requieren transformaciones antes de su modelamiento.

Series regulares e irregulares

- Series regulares: el espaciamiento en tiempo entre registros es fijo. Por ejemplo, existe un registro cada minuto.

- Series irregulares: los datos no siguen un patrón fijo de tiempo entre observaciones, lo cual dificulta su análisis directo y, en muchos casos, exige procesos previos de interpolación o re-muestreo.

4.1.3. Autocorrelación y memoria temporal (en series financieras no estacionarias)

En una serie de tiempo, los valores no suelen ser independientes entre sí: lo que ocurre en un instante determinado está, en mayor o menor medida, influenciado por lo que ocurrió en momentos anteriores. A este fenómeno se le conoce como autocorrelación y está directamente relacionado con el concepto de memoria temporal.

En el contexto de este capítulo, donde se trabaja principalmente con series financieras no estacionarias (como precios de acciones y criptomonedas), la autocorrelación juega un papel clave para entender la dinámica del mercado y para la construcción de variables predictoras basadas en el pasado.

- **Autocorrelación**

Mide el grado de relación entre el valor actual de una serie y sus valores pasados desplazados en el tiempo, conocidos como retardos o *lags*. Por ejemplo, permite responder preguntas como:

¿El precio de hoy está relacionado con el precio de ayer?

¿Existe relación entre el valor actual y el observado hace un mes?

En series financieras es frecuente encontrar autocorrelaciones débiles, inestables y variables en el tiempo, debido a la alta volatilidad y a la influencia de múltiples factores externos. Aun así, estos patrones temporales son fundamentales para el diseño de modelos predictivos.

- **Memoria temporal**

Hace referencia al horizonte de tiempo durante el cual los valores pasados influyen sobre el presente. Dependiendo del fenómeno, una serie puede presentar:

Memoria corta: solo los valores más recientes tienen impacto significativo.

Memoria larga: valores antiguos siguen influyendo en el comportamiento actual.

En datos financieros suele predominar una memoria corta, donde los últimos días u horas contienen la mayor parte de la información relevante para la predicción.

Nota aclaratoria sobre las implicaciones prácticas para este libro. En los ejemplos desarrollados en este capítulo, la autocorrelación y la memoria temporal se utilizarán principalmente para definir retardos temporales (*lags*) como variables de entrada, construir ventanas deslizantes (*sliding windows*), calcular promedios móviles como mecanismos de suavizado y alimentar modelos de

aprendizaje automático (*machine learning*) que aprendan directamente de estas dependencias temporales.

4.1.4. Diferencias entre el análisis exploratorio clásico y el EDA en series de tiempo

El análisis exploratorio de datos (EDA) en series de tiempo requiere un enfoque diferente al utilizado en datasets estructurados tradicionales. Aunque ambos comparten objetivos como comprender la distribución, detectar patrones y encontrar anomalías, la dimensión temporal introduce retos y técnicas específicas. En este capítulo, además, trabajamos con datos financieros no estacionarios, lo cual refuerza la necesidad de un EDA adaptado a este tipo de señales.

A continuación, se presentan las diferencias más relevantes.

- **El orden de los datos sí importa**

En un EDA clásico, las filas pueden reorganizarse sin afectar la interpretación de los resultados. En series de tiempo, reordenar los registros destruye la estructura temporal, impidiendo analizar tendencias, patrones o rupturas.

Por esta razón, el primer paso del EDA temporal es asegurar que los datos estén correctamente ordenados por fecha y hora.

- **Las visualizaciones se basan en el tiempo**

En EDA tradicional predominan histogramas, diagramas de caja y gráficos de barras. En series de tiempo, la herramienta más importante es la gráfica temporal, que permite visualizar:

- Tendencias
- Cambios abruptos
- Periodos de alta volatilidad
- Comportamientos inusuales

Para datos financieros (ej. como los precios de acciones o criptomonedas), estas curvas son esenciales para identificar patrones reales del mercado.

- **Análisis de estabilidad temporal**

Mientras que en EDA clásico se evalúan distribuciones completas, en series de tiempo se revisan cómo cambian esas distribuciones a lo largo del tiempo.

Por ejemplo:

- ❖ ¿Aumenta la varianza en ciertos periodos?
- ❖ ¿La volatilidad del precio cambia según el año o el mes?
- ❖ ¿Existen rupturas estructurales evidentes?

Este análisis es especialmente relevante en datos no estacionarios.

- **Uso de estadísticas móviles**

En vez de trabajar con una única media o desviación estándar global, en series de tiempo se emplean *promedios móviles*, *desviaciones móviles* y *ventanas deslizantes*, que permiten estudiar cómo evolucionan estos indicadores en el tiempo.

Esto es particularmente útil para series financieras que presentan:

- ❖ Cambios de volatilidad
- ❖ Tendencias marcadas
- ❖ Fluctuaciones cortas y abruptas

- **Identificación de dependencias temporales**

En el EDA clásico no se consideran relaciones temporales entre registros. En series de tiempo, es indispensable evaluar cómo los valores pasados influyen en los presentes, a través de:

- Retardos (*lags*)
- Ventanas temporales
- Transformaciones basadas en memoria corta

En este capítulo, esta idea se utilizará para la creación de features y no para construir modelos estadísticos clásicos como ARIMA.

- **Tratamiento de valores faltantes**

En datasets tradicionales se puede imputar con media, mediana o moda sin mayores consecuencias en el orden de los datos. En series de tiempo, la imputación debe preservar el comportamiento temporal, por ejemplo, mediante:

- Relleno hacia adelante (*forward fill*)
- Promedios móviles

En datos financieros (ej. donde no se generan observaciones en fines de semana o festivos), es frecuente realizar procesos de re-muestreo en vez de imputación directa.

- **Enfoque orientado al comportamiento y no a la distribución**

El EDA clásico se centra en cómo están distribuidos los valores. El EDA temporal se centra en cómo cambian (evolucionan) esos valores a lo largo del tiempo, lo cual es esencial para comprender fenómenos dinámicos, volátiles y no estacionarios.

Realizar un EDA adecuado en series de tiempo permite construir *features* robustas, detectar patrones clave y preparar la señal para modelos que aprovechan su estructura temporal. Esta preparación es indispensable antes de avanzar hacia los ejemplos prácticos del capítulo.

4.2. Ingeniería de características en Series de Tiempo

A continuación, se presentarán las técnicas de construcción de características más comúnes en series de tiempo.

4.2.1. Características con retardos (Lag Features)

Las características con retardo, o *lag features*, consisten en utilizar valores pasados de una variable como nuevas variables de entrada del modelo. En el contexto de series de tiempo, un retardo de orden "*k*" corresponde al valor de la serie observado "*k*" instantes antes.

Por ejemplo, un retardo de un día en el precio de cierre representa el valor del precio del día anterior, mientras que un retardo de cinco días corresponde al precio observado cinco días atrás.

Este tipo de características permite al modelo capturar **dependencias temporales directas**, bajo el supuesto de que el comportamiento pasado de la serie puede aportar información sobre su evolución futura.

De forma conceptual:

- close_lag_1: precio de cierre del día anterior
- close_lag_5: precio de cierre de cinco días atrás

4.2.2. Ventanas deslizantes (Rolling Window Features)

Las características basadas en ventanas deslizantes, o *rolling features*, resumen el comportamiento de la serie en un intervalo de tiempo reciente mediante estadísticas simples, como el promedio o la desviación estándar.

Una ventana deslizante de tamaño "*w*" se define como el conjunto de los últimos "*w*" valores observados en la serie. A partir de esta ventana, se pueden calcular distintas métricas que describen el comportamiento local de la serie.

Ejemplos comunes incluyen:

- promedio móvil (*rolling mean*), asociado a tendencias locales,
- desviación estándar móvil (*rolling std*), asociada a volatilidad.

A diferencia de los retardos individuales, las ventanas deslizantes permiten capturar **patrones agregados**, reduciendo la sensibilidad al ruido puntual.

4.2.3. Ventanas expansivas (Expanding Window Features)

Las ventanas expansivas consideran todos los valores históricos de la serie desde su inicio hasta el instante actual. A medida que avanza el tiempo, la ventana se va ampliando, incorporando progresivamente más información.

Este tipo de características permite calcular estadísticas acumuladas, como:

- promedio histórico acumulado,
- desviación estándar histórica acumulada.

Las ventanas expansivas son útiles para capturar el **comportamiento global de largo plazo** de la serie y proporcionar al modelo un contexto histórico más amplio.

4.3. Caso de Estudio 1 de Series de Tiempo: clima

Para este primer caso de estudio se utiliza un dataset que contiene registros climáticos horarios obtenidos a partir de diferentes sensores atmosféricos.

```
import pandas as pd
clima_df = pd.read_csv("/content/clima.csv")
clima_df.head()
```

	tiempo_hora_local	temperatura	humedad	punto_rocio	direccion_viento	velocidad_viento	rafagas_viento	presion	indice_UV	ozono	intensidad_precipitacion	estado_clima
0	01-10-2019 00:00	26.0	1.0	22.01	101.0	3.08	6.42	1008.45	0.0	268.6	0.0	partly-cloudy-night
1	01-10-2019 01:00	25.0	1.0	22.06	105.0	2.94	6.28	1008.24	0.0	269.1	0.0	partly-cloudy-night
2	01-10-2019 02:00	25.0	1.0	22.06	106.0	2.82	6.08	1007.90	0.0	269.7	0.0	partly-cloudy-night
3	01-10-2019 03:00	24.0	1.0	21.87	106.0	2.63	5.74	1007.61	0.0	270.4	0.0	partly-cloudy-night
4	01-10-2019 04:00	23.0	1.0	21.53	100.0	2.39	5.30	1007.51	0.0	271.3	0.0	partly-cloudy-night

Figura 37. Pre-visualización del dataset: Clima.

La variable de salida está explícita y corresponde a “estado_clima”, mientras que las demás columnas del dataframe son los *features* iniciales (excepto tipo_hora_local al cual debemos realizarle un tratamiento especial).

Antes de empezar a trabajar con el dataset es necesario conocer el tipo de datos de cada columna,

```
clima_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 648 entries, 0 to 647
Data columns (total 12 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   tiempo_hora_local         499 non-null    object
 1   temperatura               499 non-null    float64
 2   humedad                   499 non-null    float64
 3   punto_rocio               499 non-null    float64
 4   direccion_viento          499 non-null    float64
 5   velocidad_viento          499 non-null    float64
 6   rafagas_viento            499 non-null    float64
 7   presion                   499 non-null    float64
 8   indice_UV                 499 non-null    float64
 9   ozono                     499 non-null    float64
 10  intensidad_precipitacion  499 non-null    float64
 11  estado_clima              499 non-null    object
dtypes: float64(10), object(2)
memory usage: 60.9+ KB
```

Figura 38. Información del dataset: Clima.

Observaciones iniciales:

- Aunque el dataset tiene **648 registros**, solo **499 cuentan con valores no nulos**, lo que indica la presencia de **datos faltantes** que deberán ser tratados en etapas posteriores. No obstante, en este caso, los registros faltantes corresponden a intervalos temporales completos sin medición, y no a valores aislados dentro de una observación.

- La mayoría de las variables son **numéricas continuas**, lo cual es consistente con la naturaleza de las mediciones climáticas.

- Existen **dos variables categóricas** (tiempo_hora_local y estado_clima) que requieren transformación antes de ser utilizadas en un modelo de aprendizaje automático.

- La variable tiempo_hora_local, aunque aparece como tipo object, representa información temporal y deberá convertirse a un **tipo datetime** para permitir análisis y transformaciones propias de series de tiempo.

4.3.1. EDA en el dataset Clima

La primera acción a realizar en este dataset consiste en la conversión de la columna “tiempo_hora_local” al índice del dataframe. Inicialmente debemos hacer la transformación del tipo de formato de object a *datetime* y posteriormente llevar esta columna al índice.

```
clima_df['tiempo_hora_local'] = pd.to_datetime(clima_df['tiempo_hora_local'], format='%d-%m-%Y %H:%M')

# Asignación de la columna temporal como índice del dataframe
clima_df = clima_df.set_index('tiempo_hora_local')

# Ordenar por índice temporal (buena práctica en series de tiempo)
clima_df = clima_df.sort_index()
clima_df.head()
```

	temperatura	humedad	punto_rocio	direccion_viento	velocidad_viento	rafagas_viento	presion	indice_UV	ozono	intensidad_precipitacion	estado_clima
tiempo_hora_local											
2019-10-01 00:00:00	26.0	1.0	22.01	101.0	3.08	6.42	1008.45	0.0	268.6	0.0	partly-cloudy-night
2019-10-01 01:00:00	25.0	1.0	22.06	105.0	2.94	6.28	1008.24	0.0	269.1	0.0	partly-cloudy-night
2019-10-01 02:00:00	25.0	1.0	22.06	106.0	2.82	6.08	1007.90	0.0	269.7	0.0	partly-cloudy-night
2019-10-01 03:00:00	24.0	1.0	21.87	106.0	2.63	5.74	1007.61	0.0	270.4	0.0	partly-cloudy-night
2019-10-01 04:00:00	23.0	1.0	21.53	100.0	2.39	5.30	1007.51	0.0	271.3	0.0	partly-cloudy-night

Figura 39. Dataset Clima: conversión de la columna tiempo_hora_local.

La segunda tarea a realizar consiste en responder al siguiente interrogante:

¿qué hacer con los datos faltantes?

En este caso, la respuesta difiere ligeramente de la estrategia empleada para los datos estructurados analizados en el capítulo anterior. Lo primero que debe tenerse en cuenta es que, en este tipo de datasets, los registros mantienen una relación temporal, por lo que, a priori, no resulta una buena estrategia romper dicha dependencia eliminando filas con valores faltantes de forma indiscriminada. Sin embargo, el análisis cambia cuando se trata de la variable de salida, ya que no es válido "suponer" o imputar su valor. Hacerlo introduciría etiquetas artificiales en el proceso de aprendizaje y daría lugar a un problema de *label leakage*, afectando negativamente la validez del modelo.

Por lo cual, vamos a eliminar los registros en los cuales existe dato faltante en la salida a predecir, que en nuestro caso es "estado_clima":

```
clima_df = clima_df.dropna(subset=['estado_clima'])
```

Y revisamos qué ocurrió en nuestro dataset:

```
clima_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 499 entries, 2019-10-01 00:00:00 to 2019-10-21 18:00:00
Data columns (total 11 columns):
 #   Column                     Non-Null Count  Dtype
---  ------                     --------------  -----
 0   temperatura                499 non-null    float64
 1   humedad                    499 non-null    float64
 2   punto_rocio                499 non-null    float64
 3   direccion_viento           499 non-null    float64
 4   velocidad_viento           499 non-null    float64
 5   rafagas_viento             499 non-null    float64
 6   presion                    499 non-null    float64
 7   indice_UV                  499 non-null    float64
 8   ozono                      499 non-null    float64
 9   intensidad_precipitacion   499 non-null    float64
 10  estado_clima               499 non-null    object
dtypes: float64(10), object(1)
memory usage: 46.8+ KB
```

Figura 40. Dataset Clima: conversión de la columna tiempo_hora_local.

En este caso particular, se observa que los registros con valores faltantes en la variable objetivo coinciden con aquellos que también presentan datos faltantes en las variables de entrada. Esto sugiere la ausencia completa de observaciones en determinados intervalos temporales. Por esta razón, al eliminar los registros sin valor en la variable de salida, se obtiene simultáneamente un daraset sin valores faltantes, sin necesidad de aplicar técnicas adicionales de imputación.

Como tercera tarea, se procede a **transformar la variable de salida**. El primer paso consiste en identificar cuántas categorías diferentes presenta la columna "*estado_clima*". Para ello, se ejecuta el siguiente comando:

```
unicos_clima = clima_df['estado_clima'].unique()
print(unicos_clima)
```

Y obtenemos:

['partly-cloudy-night' 'partly-cloudy-day' 'clear-night' 'clear-day'

'cloudy']

A partir de esta información, se opta por definir un **diccionario de codificación manual**, en el cual se asigna un valor entero a cada categoría de la variable *estado_clima*. Esta codificación permite representar la variable de salida de forma compacta y adecuada para su uso en modelos de clasificación supervisada.

Es importante destacar que en este escenario **no se aplica one-hot encoding**, ya que esta técnica está diseñada para variables de entrada (*features*) y no para la variable de salida. En problemas de clasificación multiclase, la salida debe representarse como una única columna numérica que identifique la clase asociada a cada registro. El uso de one-hot encoding en la variable objetivo rompería la formulación estándar del problema y dificultaría la interpretación y evaluación del modelo.

```
# Diccionario de codificación del estado del clima
clima_dict = {
    'clear-day': 1,
    'clear-night': 2,
    'partly-cloudy-day': 3,
    'partly-cloudy-night': 4,
    'cloudy': 5
}
```

Y continuamos con la codificación de la salida:

```
# Codificación de la variable de salida
clima_df['estado_clima_enc'] = clima_df['estado_clima'].map(clima_
dict)

# Verificar que no existan valores sin codificar
print("Valores no codificados:",
      clima_df['estado_clima_enc'].isna().sum())

# Eliminar la columna "estado_clima"
del clima_df['estado_clima']

clima_df.head()
```

	temperatura	humedad	punto_rocio	direccion_viento	velocidad_viento	rafagas_viento	presion	indice_UV	ozono	intensidad_precipitacion	estado_clima_enc
tiempo_hora_local											
2019-10-01 00:00:00	26.0	1.0	22.01	101.0	3.08	6.42	1008.45	0.0	268.6	0.0	4
2019-10-01 01:00:00	25.0	1.0	22.06	105.0	2.94	6.28	1008.24	0.0	269.1	0.0	4
2019-10-01 02:00:00	25.0	1.0	22.06	106.0	2.82	6.08	1007.90	0.0	269.7	0.0	4
2019-10-01 03:00:00	24.0	1.0	21.87	106.0	2.63	5.74	1007.61	0.0	270.4	0.0	4
2019-10-01 04:00:00	23.0	1.0	21.53	100.0	2.39	5.30	1007.51	0.0	271.3	0.0	4

Figura 41. Codificación de la salida en el dataset Clima.

4.3.2. Modelamiento del dataset Clima (baseline)

Ya tenemos el dataset listo para realizar modelamiento. Sin embargo, es recomendado conocer si el problema de clasificación (que en este caso es multi-clase) es balanceado o no. Para ello, vamos a utilizar el siguiente código en Python:

```
import matplotlib.pyplot as plt

# Diccionario inverso (número → nombre)
inv_clima_dict = {v: k for k, v in clima_dict.items()}

# Conteo por clase
counts = (
    clima_df[“estado_clima_enc”]
    .value_counts()
    .sort_index()
    .rename(index=inv_clima_dict)
)

# Gráfica
plt.figure(figsize=(8,5))
bars = plt.bar(
    counts.index,
    counts.values
)

# Etiquetas encima de cada barra
for bar in bars:
    height = bar.get_height()
    plt.text(
        bar.get_x() + bar.get_width() / 2,
        height,
        f”{int(height)}”,
        ha=”center”,
        va=”bottom”
    )
```

```
plt.xlabel("Estado del clima")
plt.ylabel("Número de observaciones")
plt.title("Distribución de clases del dataset de Clima")
plt.xticks(rotation=30)
plt.tight_layout()
plt.show()
```

Y obtenemos:

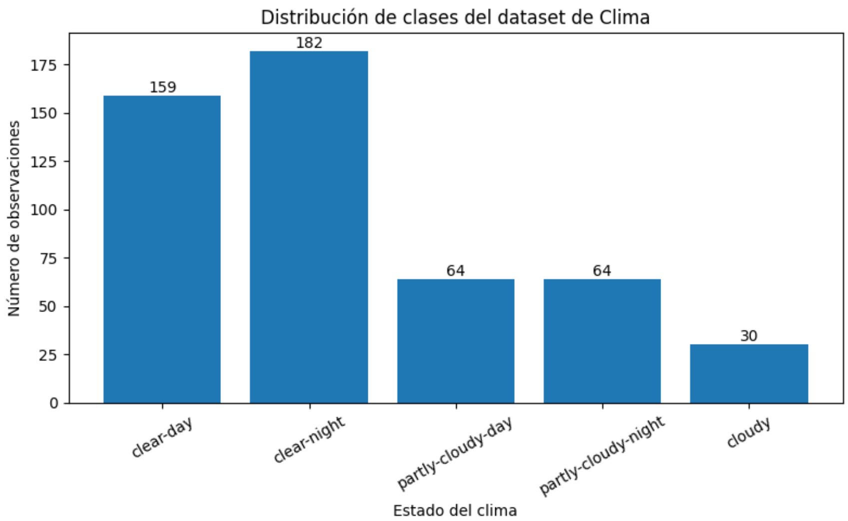


Figura 42. Distribución de datos por clase, dataset Clima.

De acuerdo con los resultados, el dataset no es balanceado, existiendo una diferencia significativa entre la clase mayoritaria (182 casos) y la minoritaria (30 casos).

Cuando realicemos el modelamiento, entonces debemos utilizar la opción "weighted" en el cálculo de las métricas precisión, recall y F1. Utilizamos split temporal, para garantizar que datos consecutivos queden en entrenamiento y otro grupo de datos consecutivos queden en test, dada la dependencia temporal entre los registros. Esta es una diferencia importante en términos de Split con datos estructurados que no son series de tiempo. Y como modelo utilizamos RandomForestClassifier

```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    confusion_matrix, classification_report
)
# ==========================
# 1) Definir X e y
# ==========================
X = clima_df.drop(columns=["estado_clima_enc"], errors="ignore")
y = clima_df["estado_clima_enc"]  # objetivo multiclase (1..5)
# ==========================
# 2) Split temporal (70%-30%)
# ==========================
split = int(len(clima_df) * 0.7)
X_train, X_test = X.iloc[:split], X.iloc[split:]
y_train, y_test = y.iloc[:split], y.iloc[split:]
# ==========================
# 3) Entrenar modelo (baseline sin FE)
# ==========================
clf = RandomForestClassifier(
    n_estimators=100,
    random_state=42,
    class_weight="balanced"
)
clf.fit(X_train, y_train)
# ==========================
# 4) Predicción
# ==========================
y_pred = clf.predict(X_test)
# ==========================
# 5) Métricas
# ==========================
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average="weighted", zero_
```

```
division=0)
recall = recall_score(y_test, y_pred, average="weighted", zero_
division=0)
f1 = f1_score(y_test, y_pred, average="weighted", zero_division=0)
print(f"Accuracy:  {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall:    {recall:.4f}")
print(f"F1-score:  {f1:.4f}")
print("\nClassification Report:\n", classification_report(y_test, y_pred,
 zero_division=0))
# ==========================
# 6) Matriz de confusión
# ==========================
labels_num = [1, 2, 3, 4, 5]
labels_txt = ['clear-day', 'clear-night', 'partly-cloudy-day', 'partly-cloudy-
night', 'cloudy']
conf_matrix = confusion_matrix(y_test, y_pred, labels=labels_num)
plt.figure(figsize=(7, 5))
sns.heatmap(
    conf_matrix,
    annot=True,
    fmt="d",
    cmap="Blues",
    xticklabels=labels_txt,
    yticklabels=labels_txt
)
plt.xlabel("Predicción")
plt.ylabel("Real")
plt.title("Matriz de Confusión (Baseline, sin FE)")
plt.show()
```

```
Accuracy:  0.6867
Precision: 0.6549
Recall:    0.6867
F1-score:  0.6589

Classification Report:
              precision    recall  f1-score   support

           1       0.93      0.78      0.85        68
           2       0.62      0.89      0.74        56
           3       0.00      0.00      0.00         3
           4       0.00      0.00      0.00        23
           5       0.00      0.00      0.00         0

    accuracy                           0.69       150
   macro avg       0.31      0.33      0.32       150
weighted avg       0.65      0.69      0.66       150
```

Figura 43. Resultados de modelamiento del dataset de Clima, sin FE.

Las clases 3, 4 y 5 (que son minoritarias), presentan un desempeño muy bajo, con ausencia de aciertos en el conjunto de prueba, dado que, no tienen ningún acierto.

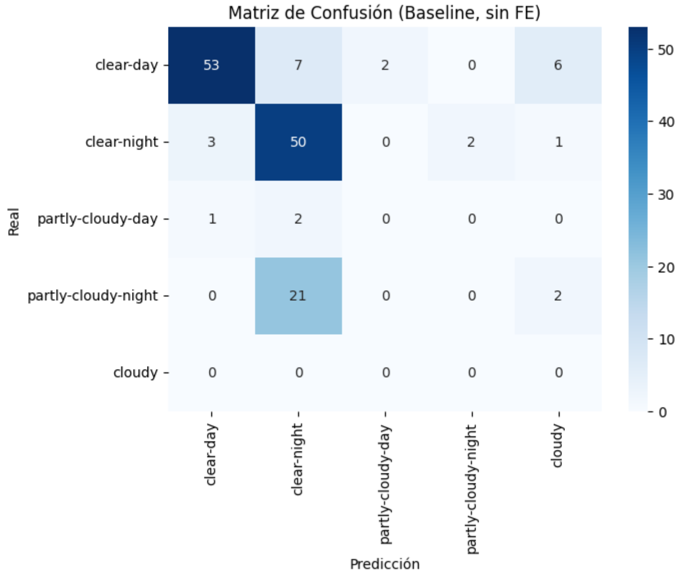


Figura 44. Matriz de confusión del dataset Clima, sin FE.

4.3.3. Ingeniería de Características aplicada al dataset de Clima

Matriz de correlación

En esta sección se analiza la **correlación entre las variables de entrada**, con el objetivo de apoyar de forma sistemática el proceso de **ingeniería de características**. La matriz de correlación permite identificar relaciones lineales entre los diferentes atributos climáticos, lo cual resulta clave para la toma de decisiones en la construcción de nuevos *features*.

Para ello, se calcula la matriz de correlación considerando únicamente las variables numéricas del dataset, excluyendo la variable objetivo codificada.

```
import seaborn as sns
import matplotlib.pyplot as plt

# Seleccionar solo variables numéricas
corr_matrix = clima_df.drop(columns=['estado_clima_enc']).corr()

# Graficar heatmap de correlación
plt.figure(figsize=(10, 6))
sns.heatmap(
    corr_matrix,
    annot=True,
    cmap=”coolwarm”,
    fmt=”.2f”,
    square=True
)
plt.title(“Matriz de correlación entre variables climáticas”)
plt.show()
```

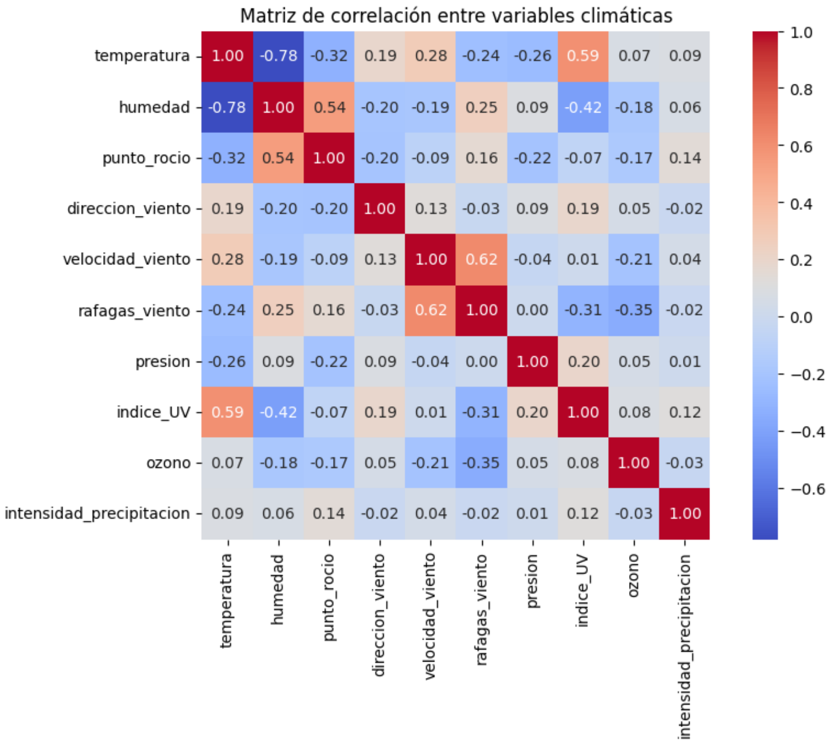


Figura 45. Matriz de correlación entre las variables del dataset de Clima.

Dado que la temperatura es una variable central del sistema climático y presenta correlaciones relevantes con otras variables del dataset, se utiliza como referencia inicial para orientar el proceso de ingeniería de características. Con el fin de evitar la generación indiscriminada de características y reducir redundancia, se seleccionan únicamente las variables con mayor correlación con la temperatura para aplicar transformaciones temporales.

Nos enfocaremos en la primera columna de la matriz de correlación y vemos que, en su orden, los tres primeros features más correlados con temperatura son: humedad (0.78), índice_UV (0.59) y punto_rocio (0.32). Entonces, inicialmente realizaremos FE con estos cuatro features, aplicando lags, Rolling y expanding a cada uno de ellos, así:

```
import pandas as pd

# Asegurar orden temporal (muy importante para lags/rolling/expanding)
clima_df = clima_df.sort_index()

# Copia para crear features sin tocar el original
clima_fe = clima_df.copy()

# -----------------------------
# 1) Lag features (retardos)
# -----------------------------
fe_cols = ["temperatura", "humedad", "indice_UV", "punto_rocio"]
sizes = [1, 2, 3]  # horas

for col in fe_cols:
    for s in sizes:
        clima_fe[f"{col}_lag{s}"] = clima_fe[col].shift(s)

# -----------------------------
# 2) Rolling features (ventanas móviles)
# -----------------------------
windows = [3, 6]  # horas

for col in fe_cols:
    for w in windows:
      clima_fe[f"{col}_roll_mean_{w}h"] = clima_fe[col].rolling(window=w).
mean()
        clima_fe[f"{col}_roll_std_{w}h"]  = clima_fe[col].rolling(window=w).
std()
       clima_fe[f"{col}_roll_min_{w}h"]  = clima_fe[col].rolling(window=w).
min()
      clima_fe[f"{col}_roll_max_{w}h"]  = clima_fe[col].rolling(window=w).
max()

# -----------------------------
# 3) Expanding features (acumuladas)
```

```
# ----------------------------
for col in fe_cols:
    clima_fe[f"{col}_exp_mean"] = clima_fe[col].expanding().mean()
    clima_fe[f"{col}_exp_max"]  = clima_fe[col].expanding().max()

# ----------------------------
# 4) Limpieza por NaNs creados por lags/rolling
# ----------------------------
clima_fe = clima_fe.dropna().copy()

# ----------------------------
# 5) Separar X e y (listo para modelamiento con FE)
# ----------------------------
X_fe = clima_fe.drop(columns=["estado_clima_enc"], errors="ignore")
y_fe = clima_fe["estado_clima_enc"]

print("Shape original:", clima_df.shape)
print("Shape con FE:", clima_fe.shape)
print("Nuevas columnas creadas:", X_fe.shape[1] - (clima_df.shape[1]
- 1))
```

Y obtenemos:

Shape original: (499, 11)
Shape con FE: (494, 63)
Nuevas columnas creadas: 52

Entrenamos el nuevo dataframe, así

```
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    confusion_matrix, classification_report
```

```
)

# ==========================
# 1) Definir X e y (con FE)
# ==========================
X = X_fe.copy()
y = y_fe.copy()   # objetivo multiclase (1..5)

# ==========================
# 2) Split temporal (70%-30%)
# ==========================
split = int(len(X) * 0.7)

X_train, X_test = X.iloc[:split], X.iloc[split:]
y_train, y_test = y.iloc[:split], y.iloc[split:]

# ==========================
# 3) Entrenar modelo (con FE)
# ==========================
clf_fe = RandomForestClassifier(
    n_estimators=200,          # un poco más robusto que el baseline
    random_state=42,
    class_weight="balanced",
    n_jobs=-1
)

clf_fe.fit(X_train, y_train)

# ==========================
# 4) Predicción
# ==========================
y_pred = clf_fe.predict(X_test)

# ==========================
# 5) Métricas
# ==========================
```

```
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average="weighted", zero_
division=0)
recall = recall_score(y_test, y_pred, average="weighted", zero_
division=0)
f1 = f1_score(y_test, y_pred, average="weighted", zero_division=0)

print("Resultados del modelo con Feature Engineering\n")
print(f"Accuracy:  {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall:    {recall:.4f}")
print(f"F1-score:  {f1:.4f}")

print("\nClassification Report:\n")
print(classification_report(y_test, y_pred, zero_division=0))

# ==========================
# 6) Matriz de confusión
# ==========================
labels_num = [1, 2, 3, 4, 5]
labels_txt = [
    "clear-day",
    "clear-night",
    "partly-cloudy-day",
    "partly-cloudy-night",
    "cloudy"
]
conf_matrix = confusion_matrix(y_test, y_pred, labels=labels_num)

plt.figure(figsize=(7, 5))
sns.heatmap(
    conf_matrix,
    annot=True,
    fmt="d",
    cmap="Blues",
    xticklabels=labels_txt,
```

```
    yticklabels=labels_txt
)
plt.xlabel("Predicción")
plt.ylabel("Real")
plt.title("Matriz de Confusión (Modelo con Feature Engineering)")
plt.tight_layout()
plt.show()
```

Y obtenemos la siguiente matriz de confusión:

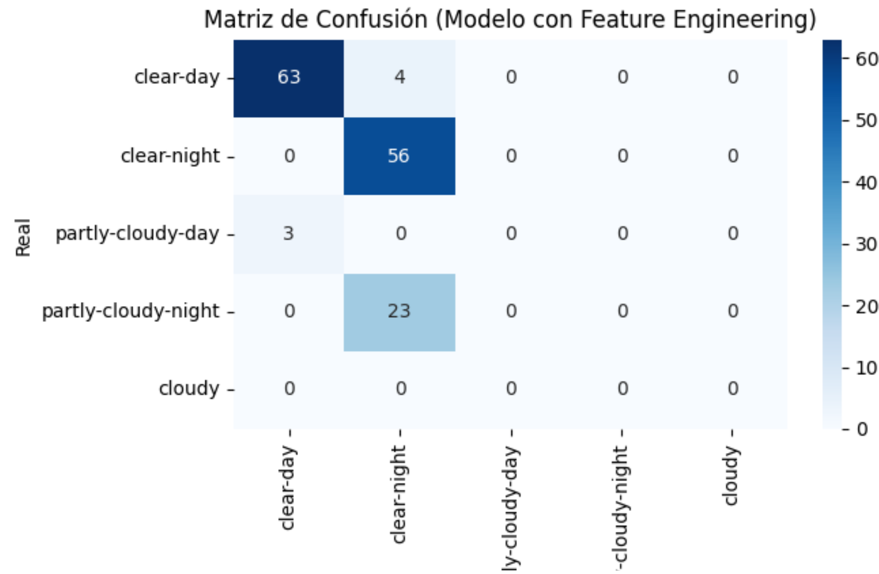


Figura 46. Matriz de confusión del dataset Clima, con FE (opción 1).

Al comparar la matriz de confusión de la Figura 46 con la presentada en la Figura 44, se observa un incremento en el número de casos correctamente clasificados para las clases *clear-day* y *clear-night*. En el escenario sin ingeniería de características, estas clases registraban 53 y 50 aciertos, respectivamente, mientras que al incorporar las características derivadas (opción 1) los aciertos aumentan a 63 y 56. El modelo con ingeniería de características alcanza una exactitud (*accuracy*) de 0.7987.

Si bien la mejora se concentra principalmente en las clases mayoritarias, este resultado evidencia que la ingeniería de características permite incorporar información adicional relevante al modelo, mejorando su desempeño sin necesidad de modificar la arquitectura del clasificador.

Finalmente, visualizaremos de los features originales y los constuidos, la correlación de ellos con la salida, así:

```
import pandas as pd
import matplotlib.pyplot as plt

# Obtener importancias
importances = clf_fe.feature_importances_

# Crear DataFrame
feature_importance_df = pd.DataFrame({
   "feature": X_fe.columns,
   "importance": importances
}).sort_values(by="importance", ascending=False)

# Mostrar las 15 más importantes
feature_importance_df.head(15)
```

El atributo feature_importances_ forma parte interna del objeto RandomForestClassifier y se genera automáticamente durante el entrenamiento del modelo.

Y para visualización:

```
plt.figure(figsize=(8,5))
top_n = 15
plt.barh(
   feature_importance_df["feature"].head(top_n)[::-1],
   feature_importance_df["importance"].head(top_n)[::-1]
)
plt.xlabel("Importancia")
plt.title("Importancia de las características (Random Forest)")
plt.tight_layout()
plt.show()
```

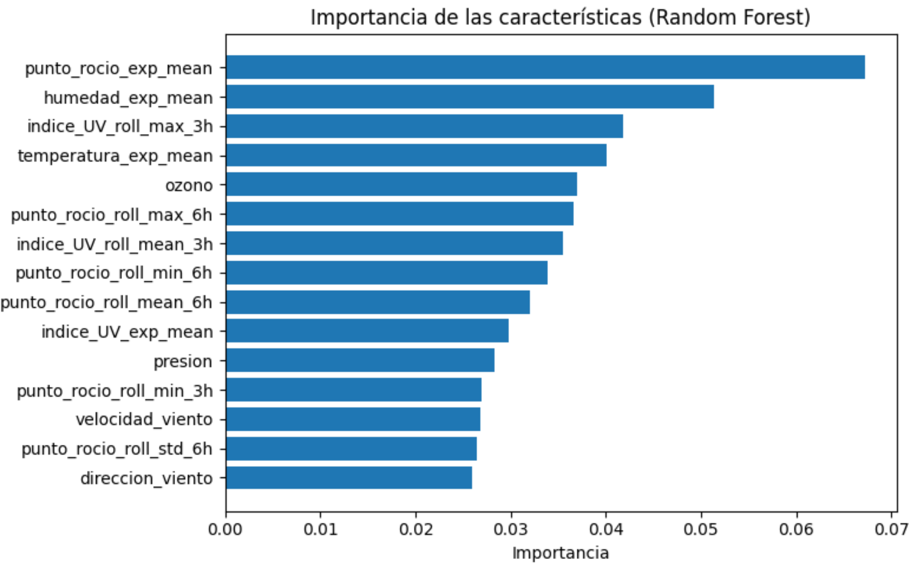


Figura 47. Importancia de las características (Random Forest) para el dataset Clima con FE.

Quiero resaltar que de los 5-top features de mayor importancia para el modelo de clasificación de estado del clima, los cuatro primeros corresponden a features construidos con FE, lo cual permitió que mejoráramos el desempeño del modelo *baseline*. Para este caso de estudio, expanding fue el método que generó features con mayor importancia.

4.4. Caso de Estudio 2 de Series de Tiempo: Twlo prices

A continuación, se trabajará con el dataset twlo_prices.csv, el cual contiene precios intradía de la acción de Twilio (TWLO) con una resolución de un minuto. El daraset incluye el precio de cierre (close), el volumen negociado (volume) y la marca temporal (date). Este tipo de series de tiempo de alta frecuencia es representativo de datos financieros reales, caracterizados por alta variabilidad, ausencia de estacionariedad y una fuerte dependencia temporal.

4.4.1. Análisis Exploratorio y construcción de la variable de salida

Iniciamos con la lectura del dataset y su pre-visualización:

```
import pandas as pd
price_df = pd.read_csv("/content/twlo_prices.csv")
price_df.head(10)
```

	close	volume	date
0	99.9800	93417.0	2020-01-02 14:30:00+00:00
1	99.7800	16685.0	2020-01-02 14:31:00+00:00
2	100.1400	21998.0	2020-01-02 14:32:00+00:00
3	100.3500	18348.0	2020-01-02 14:33:00+00:00
4	100.5500	22181.0	2020-01-02 14:34:00+00:00
5	100.6100	14573.0	2020-01-02 14:35:00+00:00
6	100.9500	13960.0	2020-01-02 14:36:00+00:00
7	100.9875	20219.0	2020-01-02 14:37:00+00:00
8	101.0500	10588.0	2020-01-02 14:38:00+00:00
9	101.1400	14118.0	2020-01-02 14:39:00+00:00

Figura 48. Pre-visualización de dataset twlo_prices.csv.

Adicionalmente, podemos conocer el tamaño del dataset, así:

```
price_df.shape
(146502, 3)
```

Es decir, la cantidad de registros es de 146,502, con las columnas: *close*, *volume* y *date*.

Tratamiento de la variable temporal

Antes de aplicar cualquier técnica de ingeniería de características (FE) en series de tiempo, es indispensable que la variable temporal sea tratada

explícitamente como tal. En el dataset twlo_prices.csv, la columna date contiene la información temporal de cada observación y debe convertirse en un índice de tipo datetime.

Este paso permite:

- preservar el orden cronológico de la serie,
- habilitar operaciones temporales (ventanas móviles, rezagos, resampling),
- y evitar errores conceptuales al tratar la fecha como una variable categórica o numérica común.

```
# Conversión de la columna date a formato datetime
price_df['date'] = pd.to_datetime(price_df['date'])
# Definir la fecha como índice temporal
price_df = price_df.set_index('date')
# Verificar estructura final
price_df.head()
```

date	close	volume
2020-01-02 14:30:00+00:00	99.9800	93417.0
2020-01-02 14:31:00+00:00	99.7800	16685.0
2020-01-02 14:32:00+00:00	100.1400	21998.0
2020-01-02 14:33:00+00:00	100.3500	18348.0
2020-01-02 14:34:00+00:00	100.5500	22181.0
2020-01-02 14:35:00+00:00	100.6100	14573.0
2020-01-02 14:36:00+00:00	100.9500	13960.0
2020-01-02 14:37:00+00:00	100.9875	20219.0
2020-01-02 14:38:00+00:00	101.0500	10588.0
2020-01-02 14:39:00+00:00	101.1400	14118.0

Figura 49. Pre-visualización de twlo_prices.csv, posterior a conversión *datetime*.

Al establecer la columna date como índice temporal, por medio de set_index('date'), esta deja de aparecer como una variable del dataset. No es necesario eliminarla explícitamente, ya que su información se conserva en el índice y será utilizada como referencia temporal para el análisis y la ingeniería de características.

De forma opcional, se puede asegurar que el dataset esté ordenado cronológicamente:

```
price_df.sort_index(inplace=True)
print(price_df.index)
```

Visualización de la serie temporal

Gracias a que el índice del *DataFrame* ahora contiene información temporal completa (*año, mes, día, hora y minuto*), es posible generar visualizaciones donde la dimensión temporal corresponde directamente al eje horizontal.

Para visualizar simultáneamente el precio de cierre y el volumen negociado, se puede utilizar el siguiente código:

```
import matplotlib.pyplot as plt

# Crear subgráficos
fig, ax1 = plt.subplots(figsize=(10, 6))

# Graficar 'close' en el primer eje
ax1.plot(price_df.index, price_df['close'], color='tab:blue', label='Close',
linewidth=2)
ax1.set_xlabel('Fecha')
ax1.set_ylabel('Precio de Cierre', color='tab:blue')
ax1.tick_params(axis='y', labelcolor='tab:blue')

# Crear un segundo eje para 'volume'
ax2 = ax1.twinx()
ax2.plot(price_df.index, price_df['volume'], color='tab:orange',
 label='Volume', linewidth=2)
ax2.set_ylabel('Volumen', color='tab:orange')
```

```
ax2.tick_params(axis='y', labelcolor='tab:orange')

# Añadir título
plt.title('Precio de Cierre y Volumen de TWLO (Intradía)', fontsize=10)

# Mostrar gráfico
plt.tight_layout()
plt.show()
```

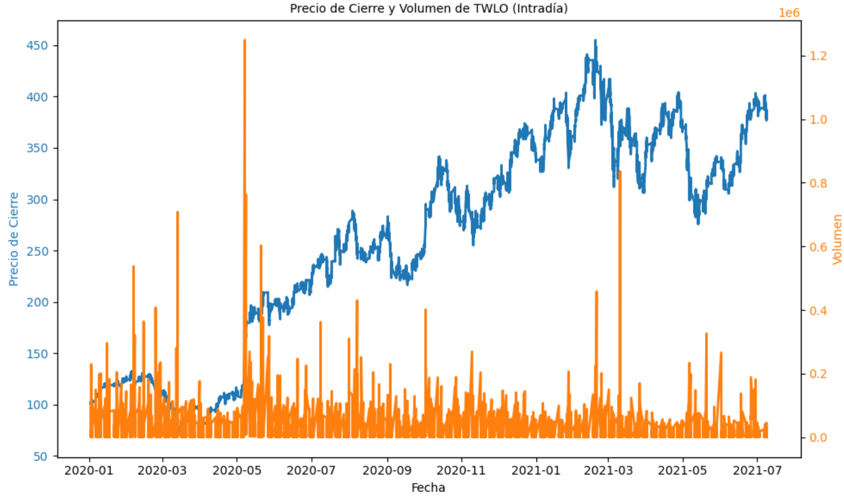


Figura 50. Gráfica de “volumen” y “close” de twlo_prices.csv

Alternativamente, las variables pueden visualizarse por separado:

```
price_df['close'].plot()
price_df['volume'].plot()
```

Construcción de la variable de salida

En problemas de series de tiempo financieras, la construcción de la variable objetivo es una de las decisiones más críticas del proceso de modelamiento. A diferencia de otros dominios, no basta con utilizar el valor actual de la serie, ya que esto conduciría a un problema trivial o a un escenario de *data*

leakage. En este caso, el objetivo es predecir si el precio de cierre diario del día siguiente presenta un incremento significativo, utilizando únicamente información disponible hasta el instante actual.

Dado que el dataset contiene precios intradía con resolución de un minuto, es necesario definir explícitamente el horizonte temporal de la predicción. En este estudio de caso, se plantea el siguiente problema:

Determinar si el precio de cierre diario del día siguiente presenta un incremento significativo respecto al día actual.

Esta formulación permite:

- evitar el uso de información futura,
- reducir el ruido intradía,
- y trabajar con una definición de salida alineada con decisiones reales de análisis financiero.

Para lo cual, el primer paso consiste en obtener el precio de cierre diario a partir de los datos intradía. Para ello, se selecciona el último valor disponible de la variable close para cada día:

```
daily_close = price_df['close'].resample('1D').last().dropna()
daily_close.head(10)
```

Obteniendo:

date	close
2020-01-02 00:00:00+00:00	103.15
2020-01-03 00:00:00+00:00	103.52
2020-01-06 00:00:00+00:00	107.46
2020-01-07 00:00:00+00:00	108.09
2020-01-08 00:00:00+00:00	109.38
2020-01-09 00:00:00+00:00	113.02
2020-01-10 00:00:00+00:00	115.75
2020-01-13 00:00:00+00:00	120.32
2020-01-14 00:00:00+00:00	119.02
2020-01-15 00:00:00+00:00	119.88

Figura 51. Precio de cierre diario, dataset twlo_prices.csv.

Este proceso transforma la serie intradía en una serie diaria, conservando la información relevante para la definición del evento a predecir.

A partir del precio de cierre diario, se calcula el retorno porcentual entre días consecutivos:

```
#Retorno diario
daily_ret = daily_close.pct_change()
```

donde *pct_change* calcula el cambio porcentual entre un valor y el valor anterior en la serie. La fórmula que se aplica internamente, es:

$$pct_change_t = \frac{P_t - P_{t-1}}{P_{t-1}}$$

Es decir, al precio del día actual se le resta el precio del día anterior, y el resultado se divide por el precio del día anterior. El valor obtenido representa el retorno relativo asociado al día actual. Por ejemplo, al precio de cierre del 3 de enero de 2020 se le resta el precio de cierre del 2 de enero de 2020 y el resultado se divide por el precio de cierre del 2 de enero de 2020, obteniendo (103.52-103.15)/103.15 = 0.003587.

```
#Retorno diario
daily_ret = daily_close.pct_change()
print(daily_ret)
```

```
date
2020-01-02 00:00:00+00:00         NaN
2020-01-03 00:00:00+00:00    0.003587
2020-01-06 00:00:00+00:00    0.038060
2020-01-07 00:00:00+00:00    0.005863
2020-01-08 00:00:00+00:00    0.011934
                               ...
2021-07-01 00:00:00+00:00   -0.018971
2021-07-02 00:00:00+00:00    0.004886
2021-07-06 00:00:00+00:00    0.014639
2021-07-07 00:00:00+00:00   -0.014985
2021-07-08 00:00:00+00:00   -0.009885
```

Figura 52. Precio de retorno, dataset twlo_prices.csv.

El primer valor del retorno diario es NaN, ya que no existe un día previo con el cual comparar el precio de cierre. El uso de retornos, en lugar de precios absolutos, permite trabajar con una variable más estable y comparable a lo largo del tiempo.

Posteriormente, creamos una variable objetivo, así:

```
# Umbral mínimo de variación (0.2%)
threshold = 0.002

# Variable objetivo: ¿sube el precio al día siguiente?
daily_target = (daily_ret.shift(-1) > threshold).astype(int)

print(daily_target)

daily_target.value_counts()
```

Este umbral (*threshold = 0.002*, equivalente a una variación del 0.2 %) se define con el fin de filtrar fluctuaciones pequeñas asociadas al ruido propio del mercado, y concentrar el análisis en movimientos de precio con mayor relevancia práctica.

La expresión daily_ret.shift(-1) desplaza la serie de retornos una posición hacia atrás, de modo que a cada registro se le asocia el retorno observado en el día siguiente. De esta forma, el valor del primer registro corresponde al retorno del segundo día, y así sucesivamente.

Posteriormente, se evalúa si dicho retorno supera el umbral definido. En caso afirmativo, se asigna el valor entero **1** a la variable objetivo daily_target, indicando un incremento relevante del precio al día siguiente; en caso contrario, se asigna el valor entero **0**.

```
date
2020-01-02 00:00:00+00:00    1
2020-01-03 00:00:00+00:00    1
2020-01-06 00:00:00+00:00    1
2020-01-07 00:00:00+00:00    1
2020-01-08 00:00:00+00:00    1
                            ..
2021-07-01 00:00:00+00:00    1
2021-07-02 00:00:00+00:00    1
2021-07-06 00:00:00+00:00    0
2021-07-07 00:00:00+00:00    0
2021-07-08 00:00:00+00:00    0
Name: close, Length: 382, dtype: int64
        count

 close
```

Figura 53. Valor daily_target del dataset twlo_prices.csv.

Finalmente, la variable objetivo diaria se asigna a cada observación intradía correspondiente al mismo día. De esta forma, cada registro intradía queda asociado con el evento que ocurrirá al cierre del día siguiente.

```
# Crear columna auxiliar con el día
price_df['day'] = price_df.index.floor('D')
```

Con price_df.index.floor a cada uno de los registros del dataset, se les obtendrá la información únicamente del día en formato (año-mes-día) y dejando en 00:00:00 la información de la hora y minuto del registro.

```
# Crear columna auxiliar con el día
price_df['day'] = price_df.index.floor('D')

price_df.head(10)
```

date	close	volume	day
2020-01-02 14:30:00+00:00	99.9800	93417.0	2020-01-02 00:00:00+00:00
2020-01-02 14:31:00+00:00	99.7800	16685.0	2020-01-02 00:00:00+00:00
2020-01-02 14:32:00+00:00	100.1400	21998.0	2020-01-02 00:00:00+00:00
2020-01-02 14:33:00+00:00	100.3500	18348.0	2020-01-02 00:00:00+00:00
2020-01-02 14:34:00+00:00	100.5500	22181.0	2020-01-02 00:00:00+00:00
2020-01-02 14:35:00+00:00	100.6100	14573.0	2020-01-02 00:00:00+00:00
2020-01-02 14:36:00+00:00	100.9500	13960.0	2020-01-02 00:00:00+00:00
2020-01-02 14:37:00+00:00	100.9875	20219.0	2020-01-02 00:00:00+00:00
2020-01-02 14:38:00+00:00	101.0500	10588.0	2020-01-02 00:00:00+00:00
2020-01-02 14:39:00+00:00	101.1400	14118.0	2020-01-02 00:00:00+00:00

Figura 54. Creación de la columna "day" del dataset twlo_prices.csv.

Posteriormente, con price_df.merge mezclamos el *dataframe* original con *daily_target* (que hemos renombrado como "target") a partir de la columna day.

```
# Unir la variable objetivo diaria al dataset intradía
price_df = price_df.merge(
    daily_target.rename('target'),
    left_on='day',
    right_index=True,
    how='inner'
)
# Eliminar columna auxiliar
price_df.drop(columns=['day'], inplace=True)

price_df[['close', 'target']].head()
```

date	close	volume	target
2020-01-02 14:30:00+00:00	99.98	93417.0	1
2020-01-02 14:31:00+00:00	99.78	16685.0	1
2020-01-02 14:32:00+00:00	100.14	21998.0	1
2020-01-02 14:33:00+00:00	100.35	18348.0	1
2020-01-02 14:34:00+00:00	100.55	22181.0	1

Figura 55. Dataset twlo_prices.csv con columna de salida denominada "target".

A partir de este punto, el dataset contiene explícitamente la variable target, que será utilizada en los experimentos de modelamiento.

Con info podemos verificar que el dataset contiene 146.502 registros, con dos columnas de entrada numéricas de tipo float64 correspondientes a "close" y "volumen" y una columna de salida de tipo int64 correspondiente a "target". Todas las celdas están diligenciadas.

4.2.2. Modelamiento en twlo_prices dataset antes de FE

Ahora podemos proceder con el modelamiento. Es importante aclarar que el modelo obtenido en este punto corresponde al **modelo base (baseline)**, el cual será utilizado posteriormente como referencia para compararlo con el modelo resultante de aplicar técnicas de **ingeniería de características (Feature Engineering, FE)** al dataset.

Paso 1: Importe de librerías y definición de la entrada del modelo (X) y de la salida (y)

```
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_
score, f1_score, confusion_matrix, classification_report
```

```
import seaborn as sns

X = price_df.drop(columns=["target"])
y = price_df["target"]
```

Paso 2: Identificación de tipo de dataset (balanceado/desbalanceado)

```
y.value_counts(normalize=True)
```

target	proportion
1	0.524273
0	0.475727

dtype: float64

Figura 56. Distribución del dataset twlo_prices.csv.

Como se observa, las clases se encuentran prácticamente balanceadas, con proporciones cercanas al 50 % para ambas categorías. Esto permite interpretar métricas globales como *acc* sin el sesgo que normalmente introducen *datasets* altamente desbalanceados.

Paso 3: Split temporal

A diferencia del *split* que hacíamos en el capítulo anterior de datos estructurados, en este caso debemos tener en cuenta que los datos tienen un **orden cronológico** y no pueden mezclarse arbitrariamente. El *split* temporal divide el dataset en dos subconjuntos **respetando ese orden**: uno para entrenamiento (pasado) y otro para prueba (futuro).

```
# Split temporal (70%-30%)
split = int(len(price_df) * 0.7)
X_train, X_test = X.iloc[:split], X.iloc[split:]
y_train, y_test = y.iloc[:split], y.iloc[split:]
```

De esta forma, los datos comprendidos desde el inicio del dataset hasta el índice definido por *split* se utilizan para el **entrenamiento**, mientras que los

datos posteriores a dicho índice y hasta el final de la serie se emplean para la **evaluación del modelo**.

Paso 4: Entrenamiento del modelo

Teniendo en cuenta el tipo de salida binaria, el modelo lo trabajaremos de clasificación binaria. Utilizaremos en este caso un DecisionTreeClassifier, así:

```
# Modelamiento

from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report
import matplotlib.pyplot as plt

# Modelo base: Árbol de decisión
model = DecisionTreeClassifier(criterion="entropy", max_depth=2,
random_state=42)
model.fit(X_train, y_train)

# Predicción
y_pred = model.predict(X_test)

# Métricas
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")

print("\nMatriz de Confusión:")
print(confusion_matrix(y_test, y_pred))

print("\nReporte de Clasificación:")
print(classification_report(y_test, y_pred, target_names=["No subió",
"Subió"]))

# Visualización del árbol
plt.figure(figsize=(12, 8))
plot_tree(
```

```
        model,
        feature_names=X.columns,
        class_names=["No subió", "Subió"],
        filled=True,
        rounded=True
    )
    plt.show()
```

Este código es muy similar al utilizado en el Capítulo anterior. En este caso las etiquetas son: "No subió", "Subió".

Y obtenemos el siguiente resultado:

```
Accuracy: 0.56

Matriz de Confusión:
[[ 8207 15185]
 [ 4078 16481]]

Reporte de Clasificación:
              precision    recall  f1-score   support

    No subió       0.67      0.35      0.46     23392
       Subió       0.52      0.80      0.63     20559

    accuracy                           0.56     43951
   macro avg       0.59      0.58      0.55     43951
weighted avg       0.60      0.56      0.54     43951
```

Figura 57. Información del dataset twlo_prices.csv, después del proceso de EDA.

Dado que la clase mayoritaria corresponde aproximadamente al 52% de las observaciones, el accuracy de referencia (que ocurre cuando el clasificador siempre predice la clase mayoritaria) es cercano al 52 %. El árbol de decisión entrenado sin ingeniería de características obtiene un *accuracy* de 56%, apenas 4% por encima del *baseline nulo*. Esto indica que, en términos globales, el modelo apenas supera una predicción aleatoria informada por la distribución de clases, lo cual evidencia una capacidad predictiva muy limitada.

4.4.3. Ingeniería de características con selección estadística

A partir de este punto, el objetivo es incorporar **contexto temporal** al dataset *twlo_prices*, de modo que el modelo pueda capturar patrones dinámicos que no están presentes en los valores instantáneos de las variables *close* y *volume*.

Si bien las técnicas de ingeniería de características empleadas en este capítulo son ampliamente utilizadas en el análisis de series de tiempo, la selección de sus parámetros (ej. el número de retardos o el tamaño de las ventanas temporales) no es trivial. Elegir estos valores de forma arbitraria puede:

- introducir ruido innecesario en los datos,
- aumentar la dimensionalidad del problema, y
- degradar el desempeño del modelo.

Por esta razón, en este estudio se emplean criterios estadísticos para guiar la selección de retardos y tamaños de ventana, con el objetivo de reducir el costo computacional y mejorar la interpretabilidad del proceso de ingeniería de características.

El procedimiento que se presenta a continuación corresponde a un enfoque avanzado de ingeniería de características guiado por criterios estadísticos. No se espera que el lector memorice el código, sino que comprenda la lógica general que sustenta la selección de retardos y ventanas temporales.

```
import numpy as np
import pandas as pd
from statsmodels.tsa.stattools import acf
from sklearn.feature_selection import mutual_info_classif

def add_statistical_fe(
    price_df: pd.DataFrame,
    max_lag: int = 60,
    top_k_lags: int = 8,
    acf_nlags: int = 200,
    acf_threshold: float = 0.05,
```

```
    use_expanding: bool = True,
):
    df = price_df.copy()

    # --- 0) Orden y verificación ---
    df = df.sort_index()
    required = {“close”, “volume”, “target”}
    missing = required - set(df.columns)
    if missing:
        raise ValueError(f”Faltan columnas requeridas: {missing}”)

    # --- 1) Selección de ventanas rolling usando ACF -> tiempo de
decorrelación (tau) ---
    r = df[“close”].dropna()
    if len(r) < 50:
        raise ValueError(“Muy pocos datos para estimar ACF de forma
 estable.”)

    acf_vals = acf(r, nlags=min(acf_nlags, len(r) - 1), fft=True)
    # Primer lag donde |ACF| cae por debajo del umbral
    tau = next((i for i in range(1, len(acf_vals)) if abs(acf_vals[i]) < acf_
threshold), 20)

   # Ventanas alrededor de tau (redondeadas y con mínimos razonables)
   windows = sorted({max(3, int(round(tau / 2))), max(3, int(round(tau))),
max(5, int(round(2 * tau)))})
    # Si tau es muy pequeño, añade una ventana un poco más larga
para estabilidad
    if windows[-1] < 10:
        windows = sorted(set(windows + [10, 20]))

    # --- 2) Selección de lags usando Mutual Information (MI) ---
    # Creamos candidatos de lags sobre retornos (ret)
    X_lags = pd.concat({f”close_lag_{k}”: df[“close”].shift(k) for k in
range(1, max_lag + 1)}, axis=1)
    tmp_mi = pd.concat([X_lags, df[“target”]], axis=1).dropna()
```

```
    if len(tmp_mi) < 200:
        # Si hay pocos datos (por NaNs), baja max_lag automáticamente
        new_max_lag = max(10, min(max_lag, len(df) // 20))
        X_lags = pd.concat({f"close_lag_{k}": df["close"].shift(k) for k in
range(1, new_max_lag + 1)}, axis=1)
        tmp_mi = pd.concat([X_lags, df["target"]], axis=1).dropna()

    mi = mutual_info_classif(
        tmp_mi.drop(columns=["target"]),
        tmp_mi["target"],
        random_state=42
    )
    mi_rank = (
        pd.Series(mi, index=tmp_mi.drop(columns=["target"]).columns)
        .sort_values(ascending=False)
    )

    # Top-k lags (extrae el número del lag desde el nombre "ret_lag_k")
    selected_lags = [int(name.split("_")[-1]) for name in mi_rank.
head(top_k_lags).index]
    selected_lags = sorted(set(selected_lags))

    # --- 3) Construcción de features con lags seleccionados ---
    # Lags de Close
    for k in selected_lags:
        df[f"close_lag_{k}"] = df["close"].shift(k)

    # (Opcional) Lags de volumen en los mismos k (a veces ayuda, a
veces no; útil para docencia)
    for k in selected_lags:
        df[f"vol_lag_{k}"] = df["volume"].shift(k)

    # --- 4) Rolling features con ventanas elegidas (sobre Close y volume)
---

    for w in windows:
        # retornos
```

```
        df[f"close_roll_mean_{w}"] = df["close"].rolling(window=w).mean()
        df[f"close_roll_std_{w}"] = df["close"].rolling(window=w).std()

        # volumen (se suele beneficiar de suavizado)
        df[f"vol_roll_mean_{w}"] = df["volume"].rolling(window=w).mean()
        df[f"vol_roll_std_{w}"] = df["volume"].rolling(window=w).std()

    # --- 5) Expanding (acumulado histórico sobre Close y volume) ---
    if use_expanding:
        df["close_exp_mean"] = df["close"].expanding().mean()
        df["close_exp_std"]  = df["close"].expanding().std()
        df["vol_exp_mean"] = df["volume"].expanding().mean()
        df["vol_exp_std"]  = df["volume"].expanding().std()

    # --- 6) Dataset final ---
    df_fe = df.dropna().copy()

    # X / y listos para modelar (sin leakage de target)
    X = df_fe.drop(columns=["target"])
    y = df_fe["target"]

    # Parámetros seleccionados
    meta = {
        "tau_acf": tau,
        "acf_threshold": acf_threshold,
        "rolling_windows": windows,
        "selected_lags_mi": selected_lags,
        "top_k_lags": top_k_lags,
        "max_lag_considered": max_lag,
    }
    return X, y, df_fe, meta
```

Este código realiza tres tareas principales:

(1) selecciona lags de forma informada usando *Mutual Information (MI)*,

(2) selecciona ventanas rolling a partir del tiempo de decorrelación estimado mediante la *función de autocorrelación (ACF)*, y

(3) genera *features* del tipo lags + rolling mean/std + expanding.

1. **Inicialmente se importan las librerías de trabajo**: *NumPy* y *Pandas* se utilizan para la manipulación de datos y *dataframes*; *acf* se emplea para estimar cuánta **memoria temporal** presenta la serie; y *mutual_info_classif* permite medir qué tanto una variable explica la salida (*target* en este caso), incluso cuando la relación no es lineal.

2. **Como segundo paso se definen los parámetros de diseño**, entre ellos: el número máximo de lags candidatos (*max_lag*), la cantidad de lags finales seleccionados (*top_k_lags*), el número máximo de retardos utilizados para estimar la ACF (*acf_nlags*), el umbral a partir del cual se considera que la serie se ha decorrelacionado (*acf_threshold*), y si se agregan o no métricas acumuladas históricas (*use_expanding*).

3. **Sección 0 – Orden y verificaciones**: el *dataframe* se ordena cronológicamente para respetar la naturaleza temporal de la serie y se verifica la existencia de las columnas requeridas.

4. **Sección 1 – Selección de ventanas rolling usando ACF → tiempo de decorrelación (τ)**: se calcula la función de autocorrelación de la serie close y se identifica el primer *lag* para el cual el valor absoluto de la ACF cae por debajo del umbral definido. Este valor se interpreta como el **tiempo de decorrelación** de la serie. A partir de τ se definen las ventanas *rolling* principales: τ/2, τ y 2τ, incorporando mínimos razonables para garantizar estabilidad numérica.

5. **Sección 2 – Selección de lags usando Mutual Information (MI)**: se generan múltiples *lags* candidatos de la variable close y se evalúa qué tan informativo resulta cada uno para predecir la variable objetivo. Los *lags* se ordenan de mayor a menor según su puntaje de MI y se seleccionan los *top_k_lags* más relevantes.

6. **Sección 3 – Construcción de *features* con lags seleccionados**: se crean nuevos *features* de tipo *lag* para las columnas close y volume, utilizando únicamente los retardos seleccionados en la sección anterior.

7. **Sección 4 – *Rolling features* con ventanas elegidas**: para cada ventana definida en la Sección 1 se calculan nuevas variables de **promedio móvil** y **desviación estándar móvil**, tanto para close como para volume, capturando tendencia y volatilidad local.

8. **Sección 5 – *Expanding* (acumulado histórico)**: se generan *features* acumuladas desde el inicio de la serie hasta el instante actual, calculando promedio y desviación estándar de forma *expanding*. En este caso, el tamaño efectivo de la ventana crece progresivamente a medida que se avanza en el *dataframe*.

9. **Sección 6 – Dataset final (limpieza, X/y y metadatos)**: Se eliminan los registros iniciales que contienen valores NaN generados por las operaciones de *lags*, *rolling* y *expanding*. Posteriormente, se separan las variables de entrada (X) y la variable objetivo (y). Adicionalmente, se construye un diccionario de **metadatos** (meta) que almacena información clave del proceso, como el valor de τ, las ventanas *rolling* seleccionadas y los *lags* finales utilizados, facilitando la trazabilidad y reproducibilidad del experimento.

Nota aclaratoria: En este caso se calcula la ACF sobre la serie de precios (*close*) con el fin de estimar la memoria temporal del nivel de precios, y no de los retornos, dado que el objetivo es capturar estructuras de dependencia de mediano plazo.

Ahora, aplicamos la función y obtenemos:

```
X_fe, y_fe, price_df_fe, fe_meta = add_statistical_fe(price_df)

print("Selección FE (estadística):")
print(fe_meta)
print("Shape con FE:", X_fe.shape, y_fe.shape)
```

Donde:

- X_fe: es el conjunto de features resultante de aplicar ingeniería de características. Incluye nuevas características tipo: retardo (*lags*), estadísticas móviles (*rolling*), estadísticas acumuladas (*expanding*).
- y_fe: corresponde a la variable objetivo, que en este caso es target, alineada con X_fe. Ambas tienen la misma cantidad de registros y el mismo índice temporal.
- price_df_fe: es el dataframe completo que contiene las variables originales, las nuevas variables obtenidas con FE, y la salida. Es útil para fines de inspección, EDA, o visualización de las características construidas.
- fe_meta: es un diccionario con los parámetros seleccionados automáticamente durante el proceso de FE, como los retardos elegidos mediante *mutual information* y los tamaños de ventana derivados de ACF. Este objeto permite documentar y reproducir las decisiones tomadas durante la ingeniería de características.

Obtenemos como resultado:

Selección FE (estadística):
{'tau_acf': 20, 'acf_threshold': 0.05, 'rolling_windows': [10, 20, 40], 'selected_lags_mi': [1, 2, 3, 4, 6, 9, 10, 12], 'top_k_lags': 8, 'max_lag_considered': 60}

Shape con FE: (146463, 34) (146463,)

Es decir, que nuestro dataset contiene ahora 34 columnas, el valor de tau encontrado fue de 20, por lo que el tamaño de ventanas para Rolling es de 10, 20 y 40 (por la regla explicada anteriormente). Se trabajaron con ochos valores de lags comprendidos entre 1 y 12.

Después de haber aplicada FE es importante verificar que la distribución del dataset no haya sido drásticamente modificada, para ello:

```
y_fe.value_counts(normalize=True)
```

Y obtenemos que la clase "1" tiene el 52% y la clase "0" tiene el 48% de los datos, al igual que el dataset original.

Finalmente, podemos realizar algunas gráficas de los nuevos features. Por ejemplo, de tipo Rolling para la variable close, en relación con el feature original, así:

```
cols = ['close', 'close_roll_mean_10', 'close_roll_mean_20']

# Seleccionar un tramo continuo y eliminar NaNs
plot_df = price_df_fe[cols].dropna().iloc[1200:1500]

plot_df.plot(figsize=(10, 4))
plt.title("Ventanas rolling del feature Close")
plt.xlabel("Tiempo")
plt.ylabel("Valor")
plt.tight_layout()
plt.show()
```

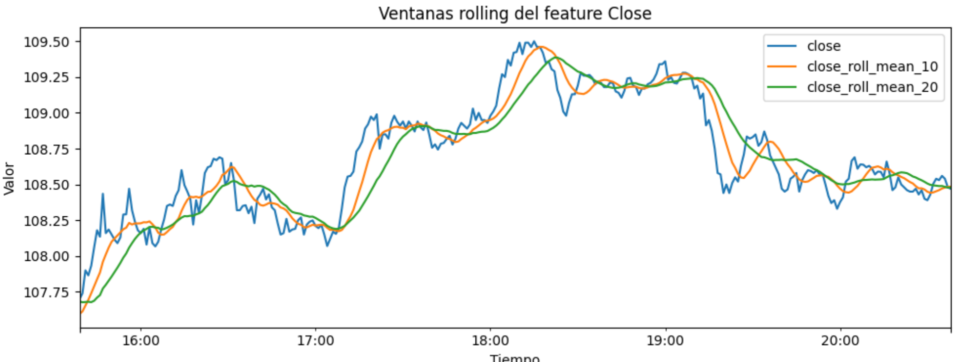


Figura 58. Ejemplo de nuevos *features* tipo Rolling, a partir de Close.

La información del nuevo dataset se presenta a continuación:

```
price_df_fe.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 146463 entries, 2020-01-02 15:09:00+00:00 to 2021-07-08 19:59:00+00:00
Data columns (total 35 columns):
 #   Column              Non-Null Count   Dtype
---  ------              --------------   -----
 0   close               146463 non-null  float64
 1   volume              146463 non-null  float64
 2   target              146463 non-null  int64
 3   close_lag_1         146463 non-null  float64
 4   close_lag_2         146463 non-null  float64
 5   close_lag_3         146463 non-null  float64
 6   close_lag_4         146463 non-null  float64
 7   close_lag_6         146463 non-null  float64
 8   close_lag_9         146463 non-null  float64
 9   close_lag_10        146463 non-null  float64
 10  close_lag_12        146463 non-null  float64
 11  vol_lag_1           146463 non-null  float64
 12  vol_lag_2           146463 non-null  float64
 13  vol_lag_3           146463 non-null  float64
 14  vol_lag_4           146463 non-null  float64
 15  vol_lag_6           146463 non-null  float64
 16  vol_lag_9           146463 non-null  float64
 17  vol_lag_10          146463 non-null  float64
 18  vol_lag_12          146463 non-null  float64
 19  close_roll_mean_10  146463 non-null  float64
 20  close_roll_std_10   146463 non-null  float64
 21  vol_roll_mean_10    146463 non-null  float64
 22  vol_roll_std_10     146463 non-null  float64
 23  close_roll_mean_20  146463 non-null  float64
 24  close_roll_std_20   146463 non-null  float64
 25  vol_roll_mean_20    146463 non-null  float64
 26  vol_roll_std_20     146463 non-null  float64
 27  close_roll_mean_40  146463 non-null  float64
 28  close_roll_std_40   146463 non-null  float64
 29  vol_roll_mean_40    146463 non-null  float64
 30  vol_roll_std_40     146463 non-null  float64
 31  close_exp_mean      146463 non-null  float64
 32  close_exp_std       146463 non-null  float64
 33  vol_exp_mean        146463 non-null  float64
 34  vol_exp_std         146463 non-null  float64
```

Figura 59. Información del dataset twlo_prices.csv, posterior a FE.

4.4.4. Entrenamiento del modelo y predicción con FE

Vamos a utilizar el mismo tipo de modelo utilizado antes de aplicar FE. El único cambio que hacemos es que nuestra entrada ahora es X_fe.

```
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report
import matplotlib.pyplot as plt

# 1) Split temporal (70/30)
split = int(len(X_fe) * 0.7)
X_train, X_test = X_fe.iloc[:split], X_fe.iloc[split:]
y_train, y_test = y_fe.iloc[:split], y_fe.iloc[split:]

# 2) Entrenar y predecir
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

# 3) Métricas
accuracy = accuracy_score(y_test, y_pred)
print(f”Accuracy: {accuracy:.2f}”)

print(“\nMatriz de Confusión:”)
cm = confusion_matrix(y_test, y_pred)
print(cm)

print(“\nReporte de Clasificación:”)
print(classification_report(y_test, y_pred, target_names=[“No subió”,
“Subió”]))

# 4) Heatmap simple (sin seaborn)
plt.figure(figsize=(6, 5))
plt.imshow(cm)
plt.title(“Matriz de Confusión”)
plt.xlabel(“Predicción”)
plt.ylabel(“Real”)
plt.xticks([0, 1], [“No subió”, “Subió”])
plt.yticks([0, 1], [“No subió”, “Subió”])
```

```
# Valores dentro de la matriz
for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        plt.text(j, i, cm[i, j], ha="center", va="center")

plt.colorbar()
plt.tight_layout()
plt.show()
```

Y obtenemos como resultado:

```
Accuracy: 0.53

Matriz de Confusión:
[[23380     0]
 [20559     0]]

Reporte de Clasificación:
              precision    recall  f1-score   support

    No subió       0.53      1.00      0.69     23380
       Subió       0.00      0.00      0.00     20559

    accuracy                           0.53     43939
   macro avg       0.27      0.50      0.35     43939
weighted avg       0.28      0.53      0.37     43939
```

Figura 60. Resultado del modelo posterior a la ingeniería de características, utilizando un árbol de decisión con dos niveles de profundidad.

El modelo resultante presenta un comportamiento claramente **sesgado hacia la clase "No subió"**, clasificando todos los registros dentro de esta categoría. Sin embargo, este resultado no debe interpretarse como un error metodológico, sino como una **limitación del modelo utilizado**.

Tras la etapa de ingeniería de características, el nuevo *dataset* contiene un número significativamente mayor de *features* en comparación con el daraset original. Un árbol de decisión con únicamente **dos niveles de profundidad** no tiene la capacidad suficiente para explotar esta mayor complejidad, ya que

dispone de muy pocas divisiones (preguntas) para separar adecuadamente el espacio de características.

Como una mejora sencilla desde un enfoque **model-centric**, se incrementa la profundidad máxima del árbol de decisión, por ejemplo, a **cuatro niveles**, permitiendo al modelo capturar relaciones más complejas entre las variables de entrada. Para ello, se define el siguiente modelo:

```
model = DecisionTreeClassifier(
    criterion="entropy",
    max_depth=4,
    min_samples_leaf=50,
    random_state=42
)
```

Posteriormente, se ejecuta nuevamente el proceso de modelamiento utilizando el daraset con ingeniería de características (X_fe), obteniendo los resultados mostrados en la Figura 50.

```
Accuracy: 0.56

Matriz de Confusión:
[[ 8188 15192]
 [ 4043 16516]]

Reporte de Clasificación:
              precision    recall  f1-score   support

    No subió       0.67      0.35      0.46     23380
       Subió       0.52      0.80      0.63     20559

    accuracy                           0.56     43939
   macro avg       0.60      0.58      0.55     43939
weighted avg       0.60      0.56      0.54     43939
```

Figura 61. Resultado del modelo posterior a la ingeniería de características, utilizando un árbol de decisión con cuatro niveles de profundidad.

Al comparar el Modelo 1 (sin ingeniería de características) con **el Modelo 3 (con ingeniería de características y árbol de cuatro niveles)**, se observan los siguientes resultados:

- El *accuracy* de ambos modelos es de **0.56**.
- Las métricas de **precisión (P)**, **recall (R)** y **F1-score** son iguales en ambos casos.
- El número de aciertos para la clase **“Subió”** en el Modelo 3 es de **16 516**, mientras que en el Modelo 1 es de **16 481**.
- El número de aciertos para la clase **“No subió”** en el Modelo 3 es de **8 188**, frente a **8 207** en el Modelo 1.
- La cantidad total de aciertos del Modelo 3 es de **24 704**, mientras que en el Modelo 1 es de **24 688**, lo que representa una diferencia de **19 registros**.

Aunque la diferencia entre ambos modelos es pequeña en términos absolutos, su interpretación depende del contexto del problema. Por ejemplo, si cada registro representara una transacción o un usuario, estos 19 casos adicionales correctamente clasificados podrían traducirse en decisiones más acertadas en un entorno real, lo cual ***no resulta despreciable en aplicaciones prácticas***.

4.5. Conclusiones de cierre del capitulo

En este capítulo se abordó el análisis de series de tiempo a través de dos casos de estudio con características y niveles de complejidad distintos. En el primer caso, basado en un dataset climático, se mostró cómo la incorporación sistemática de ingeniería de características temporales puede mejorar de forma significativa el desempeño de un modelo de clasificación, incluso sin modificar su arquitectura. Este ejemplo permitió introducir conceptos fundamentales como dependencia temporal, autocorrelación y construcción de características basadas en retardos, ventanas móviles y estadísticas acumuladas, enfatizando la importancia de una representación adecuada de los datos.

En contraste, el segundo caso de estudio, basado en datos financieros de alta frecuencia, evidenció que la ingeniería de características no garantiza necesariamente una mejora automática en el desempeño del modelo. A pesar de emplear criterios estadísticos para la selección de retardos y ventanas, y de aplicar un proceso cuidadoso para evitar *data leakage*, los resultados muestran que la naturaleza altamente ruidosa y no estacionaria de las series financieras impone limitaciones inherentes al proceso de modelamiento. Este contraste refuerza una idea central del capítulo: en ciencia de datos, las decisiones metodológicas deben estar guiadas tanto por el análisis de los datos como por el contexto del problema, y no todas las técnicas producen los mismos beneficios en todos los escenarios.

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.

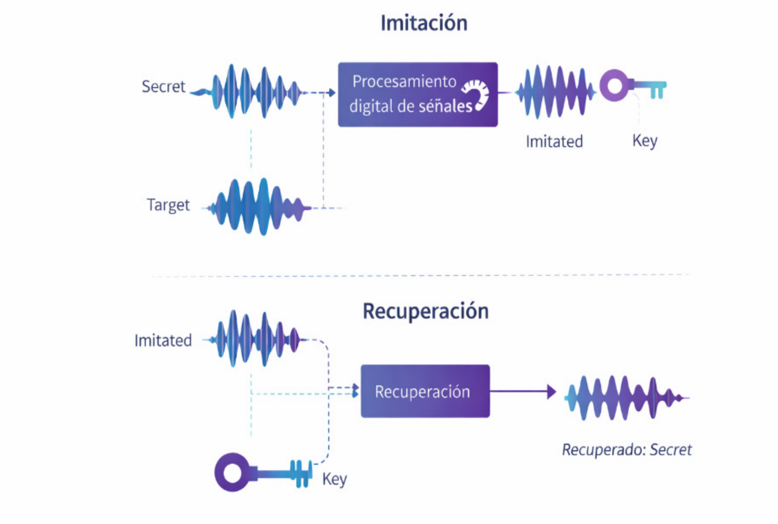


Figura 63. Diagrama del proceso de imitación y recuperación de voz.

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).

doramariaballesteros Rename reverse (1).ipynb to reverse.ipynb f31ec0a · 2 years ago 27 Co

Imitation	Imitation	2 yea
LICENSE	Initial commit	4 yea
README.md	README.md	2 yea
Reverse	Reverse	2 yea
imitated.wav	Add files via upload	2 yea
imitation.ipynb	Add files via upload	2 yea
key.csv	Add files via upload	2 yea
reverse.ipynb	Rename reverse (1).ipynb to reverse.ipynb	2 yea
secret.wav	Add files via upload	2 yea
target.wav	Add files via upload	2 yea

README MIT license

Voice Conversion using Signal Processing

It consists of two parts: Imitation and Reverse. Imitation allows to obtain an imitated message and a key from a sec message and another target. The imitated message sounds very similar to the target. In the case of Reverse, the secret message is obtained from the imitated message and the key. The main application of the code is covert communication of audio signals.

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.

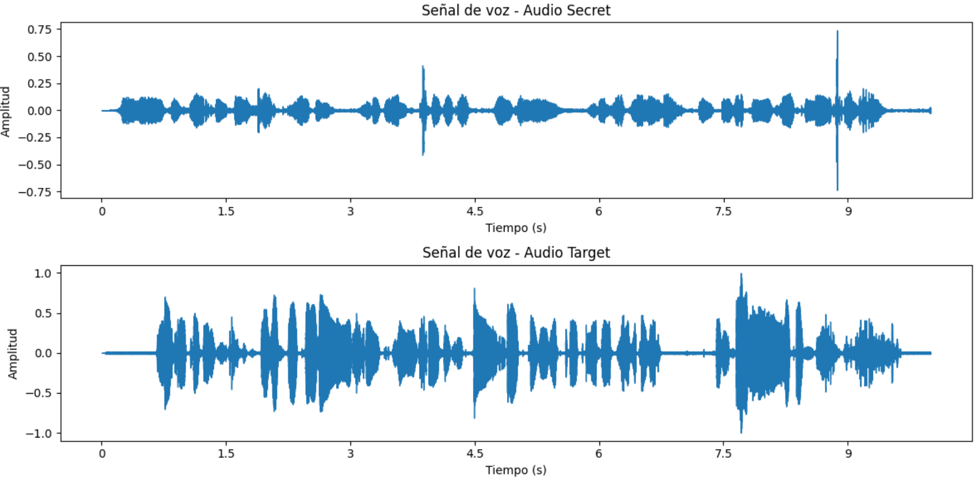


Figura 65. Visualización en el dominio del tiempo audio “secret” y “target”.

```
# 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.

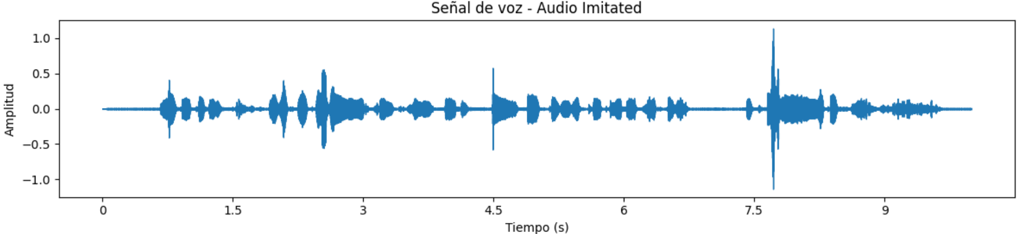


Figura 66. Visualización en el dominio del tiempo audio “imitated”.

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.

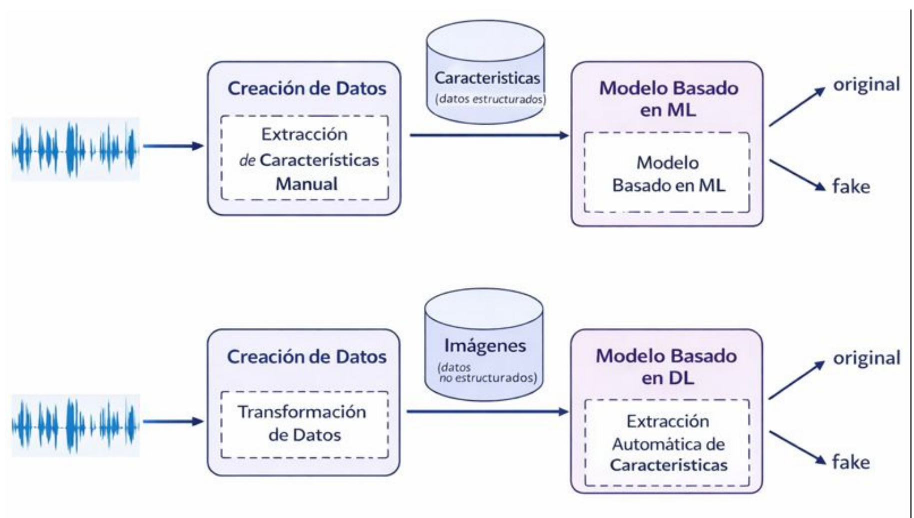


Figura 67. Enfoques dentro del proyecto Deep4SNet.

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.

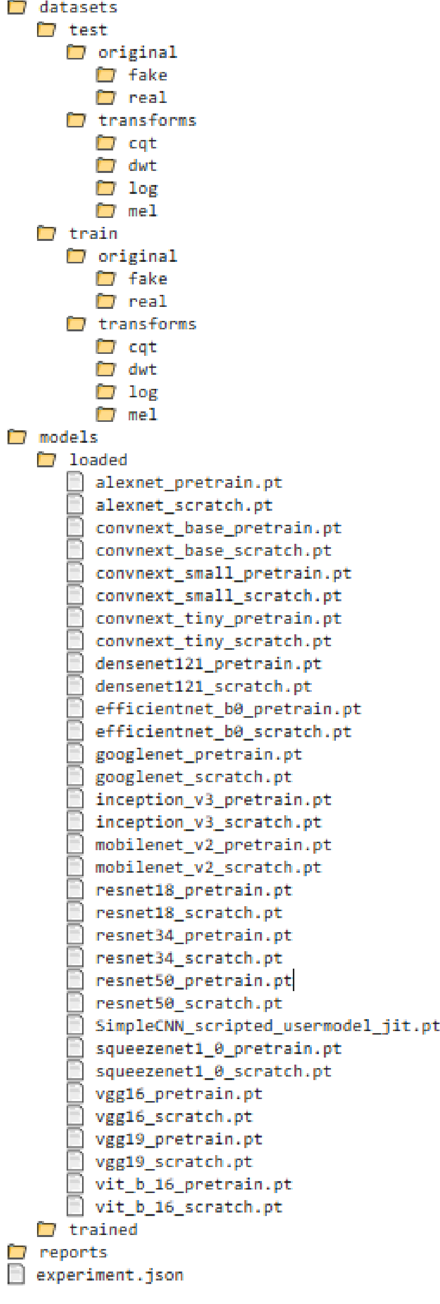


Figura 69. Ejemplo de resultado de la Sección 7 de FakeVoiceFinder.

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:

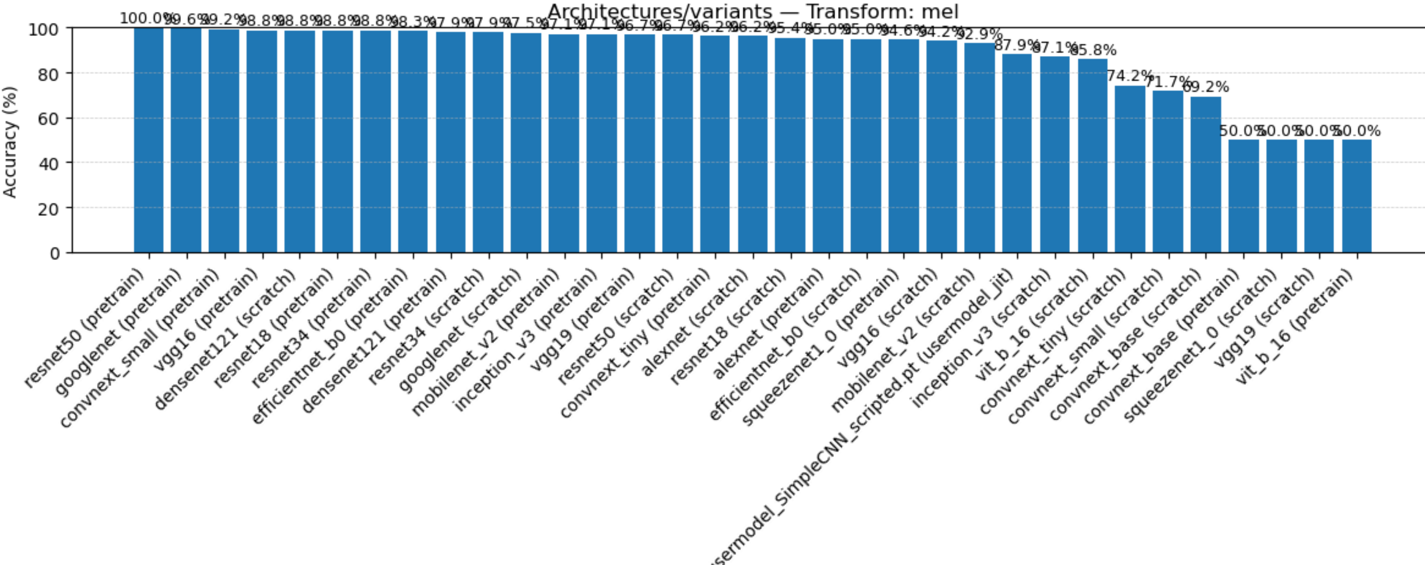


Figura 72. Ejemplo de gráfica model-center de FakeVoiceFinder: comparación de 33 modelos con el tipo de representación espectral espectrograma en escala Mel.

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.

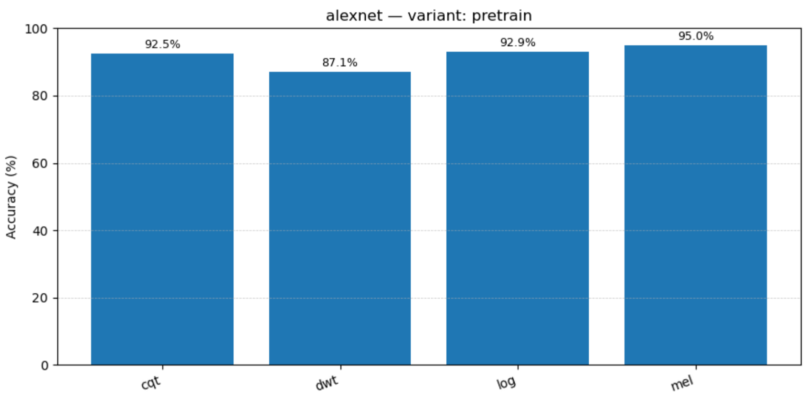
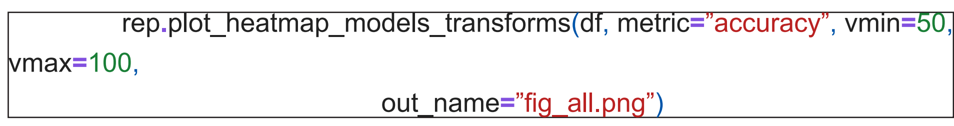


Figura 73. Ejemplo de gráfica data-center de FakeVoiceFinder: comparación de 4 modelos con diferente tipo de representación espectral, para un mismo tipo de arquitectura.

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”)
```

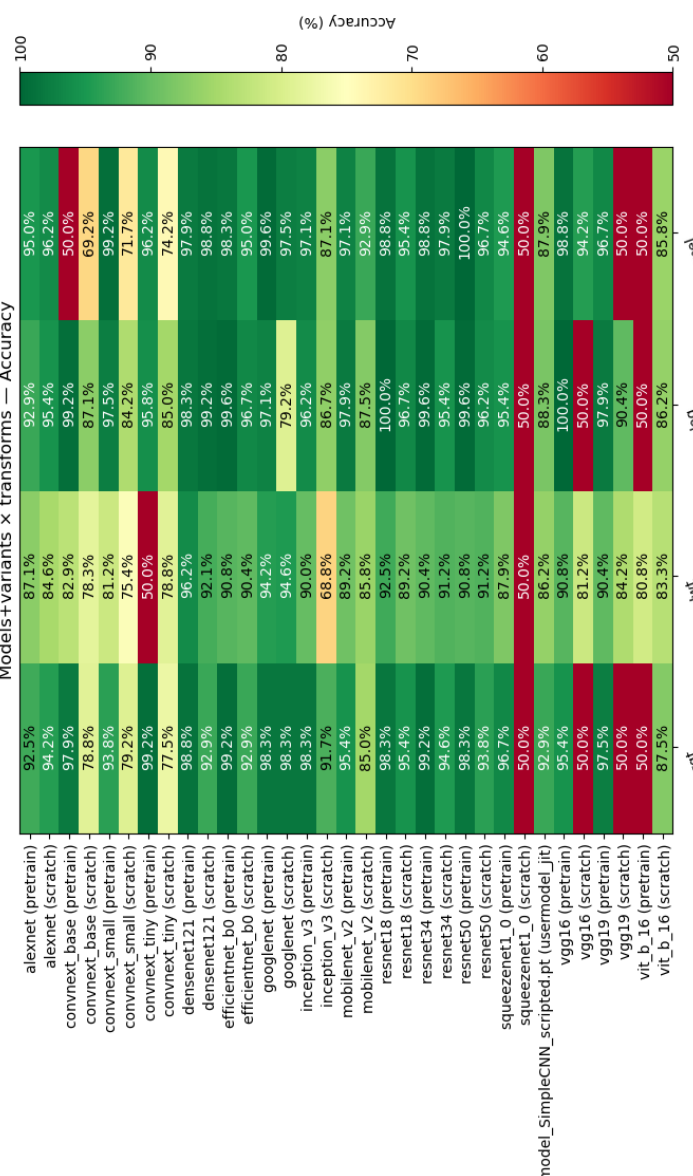


Figura 74. Ejemplo de gráfica bajo un enfoque híbrido en FakeVoiceFinder: comparación de 33 modelos, cada uno evaluado con un tipo de representación espectral (total de 132 configuraciones). La métrica de accuracy se utiliza con fines comparativos entre configuraciones.

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.

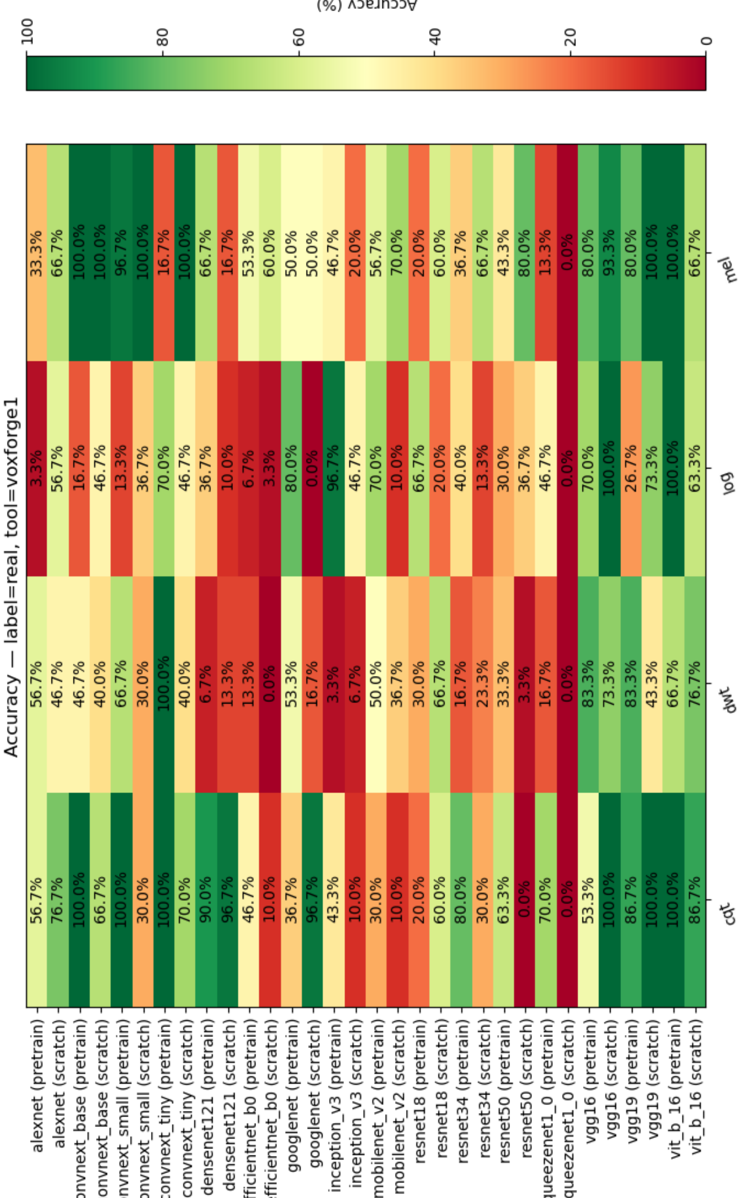


Figura 75. Ejemplo de Inferencia de FakeVoiceFinder: herramienta de gemneración de audio sintético Voxforge1.

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.

CONCLUSIONES Y REFLEXIONES DE CIERRE

A lo largo de este libro hemos recorrido un camino que va más allá del entrenamiento de modelos o del uso de librerías específicas. Hemos aprendido que la ciencia de datos comienza mucho antes del modelado y que su verdadero valor reside en la comprensión profunda del dato: su origen, su estructura, su contexto y las decisiones que lo transforman en conocimiento.

Desde los datos estructurados y las series de tiempo, hasta el audio sintético y la detección de deepfakes de voz, cada capítulo ha mostrado que no existen soluciones universales ni recetas únicas. Existen, en cambio, procesos rigurosos, preguntas bien formuladas y una responsabilidad técnica y ética frente a los datos que utilizamos y a los sistemas que construimos.

Este recorrido también deja una idea central: un modelo es tan confiable como lo son los datos que lo alimentan y las decisiones que lo preceden. La ingeniería de datos, el análisis exploratorio y la ingeniería de características no son etapas accesorias, sino el núcleo donde se define la calidad, la interpretabilidad y el impacto real de cualquier solución basada en inteligencia artificial.

En un contexto donde los sistemas automáticos tienen cada vez mayor influencia sobre la sociedad, la ciencia de datos exige algo más que habilidades técnicas. Exige criterio, conciencia y una mirada crítica capaz de cuestionar resultados, reconocer sesgos y comprender las implicaciones del uso de la tecnología. Formar ingenieros y científicos de datos no consiste únicamente en enseñar a programar modelos, sino en cultivar una forma de pensar con datos.

Este libro cierra como comenzó: con una invitación. A explorar con curiosidad, a construir con rigor y a investigar con responsabilidad. Que cada lector continúe este camino entendiendo que los datos no son un fin en sí mismos, sino un medio para comprender mejor el mundo y aportar soluciones que sean, al mismo tiempo, técnicamente sólidas y humanamente conscientes.

REFERENCIAS

1. Ballesteros, D.; Moreno, J.
Highly transparent steganography model of speech signals using Efficient Wavelet Masking.
Expert Systems with Applications, vol. 39, **2012**.
Disponible en: https://doi.org/10.1016/j.eswa.2012.02.066

2. Ballesteros, D.; Moreno, J.
On the ability of adaptation of speech signals and data hiding.
Expert Systems with Applications, vol. 39, **2012**.
Disponible en: https://doi.org/10.1016/j.eswa.2012.05.027

3. Ballesteros, D.
COVID-19 Bogotá: código en Python para análisis de datos.
Video de YouTube, **2020**.
Disponible en: https://youtu.be/kezqpilbKyA (último acceso: 19 diciembre 2025).

4. Ballesteros L., D. M.; Rodríguez, Y. P.; Renza, D.
H-Voice: Fake voice histograms (Imitation + DeepVoice).
Mendeley Data, V4, **2020**.
Disponible en: https://data.mendeley.com/datasets/k47yd3m28w/4 (último acceso: 16 enero 2025).

5. Ballesteros, D. M.; Rodríguez, Y.; Renza, D.
A dataset of histograms of original and fake voice recordings (H-Voice).
Data in Brief, vol. 29, 105331, **2020**.
Disponible en: https://www.sciencedirect.com/science/article/pii/S2352340920302250

6. Ballesteros, D.; Rodríguez, Y.; Renza, D.; Arce, G.
Deep4SNet: deep learning for fake speech classification.
Expert Systems with Applications, vol. 184, **2021**.
Disponible en: https://doi.org/10.1016/j.eswa.2021.115465

7. Ballesteros, D.
Imitation-using-Signal-Processing.
Repositorio de GitHub, **2023**.
Disponible en: https://github.com/doramariaballesteros/Imitation-using-Signal-Processing (último acceso: 19 diciembre 2025).

8. Beltrán, M.; Ballesteros L., D. M.
Fake Audio Dataset (ElevenLabs & Respeecher).
Mendeley Data, V1, **2025**.
Disponible en: https://data.mendeley.com/datasets/79g59sp69z/1
(último acceso: 16 enero 2025).

9. González, B.; Ballesteros L., D. M.
TTS/V2V Audio Deepfake Dataset.
Mendeley Data, V2, **2025**.
Disponible en: https://data.mendeley.com/datasets/h4zbs27tkr/2
(último acceso: 16 enero 2025).

10. Pachón. C.; Ballesteros, D.
FakeVoiceFinder
Repositorio de GitHub, **2025**.
Disponible en:https://github.com/DEEP-CGPS/FakeVoiceFinder
(último acceso: 16 enero 2026).

11. Pachón, C.; Ballesteros, D.
FakeVoiceFinder: An Open-Source Framework for Synthetic and Deepfake Audio Detection.
Big Data and Cognitive Computing, vol. 10, no. 1, 2026.
Disponible en: https://doi.org/10.3390/bdcc10010025