Universidad Internacional de la Rioja (UNIR)

Escuela Superior de Ingeniería y Tecnología

Máster Universitario en Inteligencia Artificial

Detección de malware binario: comparación de algoritmos clásicos y redes neuronales con técnicas de interpretabilidad (XAI)

Trabajo Fin de Estudios

presentado por: Arturo Aguilera González

Dirigido por: Almudena Ruiz Iniesta

Ciudad: Málaga

Fecha: 17 de septiembre de 2025

Índice de Contenidos

Resumen vii

1. Introducción 1

1.1. Motivación . . . 1

1.2. Planteamiento del trabajo . . . 2

1.3. Estructura del trabajo . . . 3

2. Contexto y Estado del Arte 5

2.1. Contexto del problema . . . 5

2.2. Estado del arte . . . 6

2.2.1. Evolución de la detección de malware . . . 6

2.2.2. Detección basada en aprendizaje automático clásico . . . 8

2.2.3. Redes neuronales profundas aplicadas a malware . . . 10

2.2.4. Representaciones del malware (imágenes, bytes, secuencias) . . . 13

2.2.5. Técnicas de inteligencia artificial explicable (XAI) . . . 16

2.2.6. Aplicaciones de SHAP y LIME en ciberseguridad . . . 18

2.2.7. El dataset EMBER: referencia para la detección de malware . . . 21

2.3. Conclusiones . . . 22

3. Objetivos concretos y metodología de trabajo 23

3.1. Objetivo general . . . 23

3.2. Objetivos específicos . . . 23

3.3. Metodología del trabajo . . . 25

3.3.1. Fases del proyecto . . . 25

3.3.2. Herramientas y frameworks . . . 26

4. Desarrollo del trabajo 29

4.1. Descripción del dataset EMBER . . . 29

4.2. Preprocesamiento y vectorización . . . 31

4.2.1. Vectorización . . . 31

4.2.2. Filtrado, selección y normalización del conjunto de datos . . . 33

4.2.3. Exploración del dataset . . . 37

4.3. Métricas de evaluación 42

5. Resultados experimentales **46**

5.1. Modelos clásicos 46

5.1.1. Regresión Logística 47

5.1.2. Random Forest 49

5.1.3. LightGBM 52

5.1.4. SVM (RBF) 54

5.1.5. K-Nearest Neighbors (KNN) 56

5.1.6. Multi-Layer Perceptron (MLP) 58

5.2. Modelos de Deep Learning 60

5.2.1. Red neuronal MLP con Keras 61

5.2.2. Red neuronal convolucional 1D (CNN) con Keras 63

5.2.3. Red neuronal recurrente (LSTM) con Keras 66

5.3. Discusión y análisis de resultados 69

5.3.1. Comparativa de rendimiento entre modelos 69

5.3.2. Impacto del preprocesamiento y la vectorización 70

5.3.3. Conclusión del análisis 71

6. Explicabilidad **72**

6.1. Explicabilidad en modelos clásicos 73

6.2. Explicabilidad en modelos de Deep Learning 81

7. Conclusiones y trabajo futuro **89**

7.1. Conclusiones 89

7.2. Líneas de trabajo futuro 91

Referencias 93

Índice de Ilustraciones

2.1. Distribución de los artículos considerados en el periodo 2003–2022, según Abaoja y colaboradores en 2022. 7

2.2. Arquitectura del modelo AMDDL: conversión de binarios en imágenes para clasificación mediante CNN. 11

4.1. Distribución original de clases en el conjunto de datos EMBER. Las muestras no etiquetadas (etiqueta -1) suponen aproximadamente un 20 % del conjunto. 34

4.2. Representación vectorial sin normalizar de la primera muestra. Se observa una fuerte asimetría en las escalas de valores. 34

4.3. Representación vectorial de la primera muestra tras el escalado estándar. . 36

4.4. Distribución de clases en los subconjuntos de entrenamiento y prueba (50k y 16.6k muestras respectivamente). 37

4.5. Distribución del valor de entropía en secciones PE ejecutables (rango centrado en percentiles 1–99). 38

4.6. Distribución del tamaño del archivo PE (en bytes), recortada al rango de percentiles 1–99. 39

4.7. Distribución del número de cadenas ASCII extraídas por archivo, acotada al rango de percentiles 1–99. 40

4.8. Distribución del número de URLs detectadas en cadenas ASCII, acotada al rango de percentiles 1–99. 41

4.9. Distribución de la suma total del vector de funciones importadas (`imports`), recortada al rango de percentiles 1–99 y en escala logarítmica. 41

5.1. Matriz de confusión de la regresión logística en el conjunto de test (16.6k muestras balanceadas). 48

5.2. Matriz de confusión de Random Forest en el conjunto de test (16.6k muestras balanceadas). 51

5.3. Matriz de confusión de LightGBM en el conjunto de test (16.6k muestras balanceadas). 53

5.4. Matriz de confusión de SVM (RBF) en el conjunto de test (16.6k muestras balanceadas). 55
5.5. Matriz de confusión de KNN en el conjunto de test (16.6k muestras balanceadas). 57
5.6. Matriz de confusión de MLP en el conjunto de test (16.6k muestras balanceadas). 59
5.7. Matriz de confusión obtenida por el MLP en el conjunto de test (16.6k muestras balanceadas). 63
5.8. Matriz de confusión obtenida por el modelo CNN 1D en el conjunto de test (16.6k muestras). 65
5.9. Matriz de confusión obtenida por el modelo LSTM en el conjunto de test (16.6k muestras). 68

6.1. Regresión Logística: 12 coeficientes con mayor magnitud (signo: rojo → *malware*, azul → *benigno*). 74
6.2. Importancias globales SHAP para Random Forest (Top-20 características). 77
6.3. Summary plot de SHAP para Random Forest (Top-20 características). . . . 78
6.4. Summary plot de SHAP para LightGBM: importancia global de características. 79
6.5. Decision plot de SHAP para LightGBM: explicación local de una muestra. . 80
6.6. Explicación local con LIME para LightGBM: barras de impacto por característica. 80
6.7. MLP (Keras) — SHAP *summary plot*. 83
6.8. MLP (Keras) — LIME explicación local. 84
6.9. MLP (Keras) — SHAP *waterfall* de un ejemplo local. 85
6.10. CNN (Keras) — *Summary plot* de SHAP (Top-20). 86
6.11. CNN (Keras) — Gráficos de dependencia SHAP para `f637` (header) y `f930` (section). 87

Índice de Tablas

4.1. Desglose de bloques de características vectorizadas en el dataset EMBER . 33
4.2. Identificación de la característica con mayor valor absoluto en una muestra. 35
4.3. Estadísticos descriptivos de las columnas con valores extremos (616–626), que confirman la necesidad de estandarización. 35
4.4. Resumen de utilidad observada para las variables exploradas en la fase de análisis exploratorio. 42
4.5. Estructura de una matriz de confusión binaria. 43

5.1. Parámetros evaluados y mejor configuración seleccionada para la regresión logística. 47
5.2. Rendimiento de la regresión logística en el conjunto de test (16.6k muestras). 48
5.3. Informe de clasificación de la regresión logística en el conjunto de test (16.6k muestras). Las clases se codifican como 0 = Benigno y 1 = Malware. 49
5.4. Parámetros evaluados y mejor configuración seleccionada para Random Forest. 50
5.5. Rendimiento de Random Forest en el conjunto de test (16.6k muestras). . . 50
5.6. Informe de clasificación de Random Forest en el conjunto de test (16.6k muestras). 51
5.7. Parámetros evaluados y configuración seleccionada para LightGBM. 52
5.8. Rendimiento de LightGBM en el conjunto de test (16.6k muestras). 53
5.9. Informe de clasificación de LightGBM en el conjunto de test (16.6k muestras). 53
5.10. Rendimiento de SVM (RBF) en el conjunto de test (16.6k muestras). 55
5.11. Informe de clasificación de SVM (RBF) en el conjunto de test (16.6k muestras). 55
5.12. Parámetros evaluados y mejor configuración seleccionada para KNN. 56
5.13. Rendimiento de KNN en el conjunto de test (16.6k muestras). 57
5.14. Informe de clasificación de KNN en el conjunto de test (16.6k muestras). . . 58
5.15. Rendimiento de MLP en el conjunto de test (16.6k muestras). 59
5.16. Informe de clasificación de MLP en el conjunto de test (16.6k muestras). . . 59
5.17. Resumen de convergencia del MLP (Keras). Mejor *val_accuracy* en la época 6 (=0.965). 62

5.18. Informe de clasificación del MLP implementado en Keras en el conjunto de test (16.6k muestras). 62
5.19. Evolución del entrenamiento de la CNN 1D. Mejor *val_accuracy* en la época 7 (=0.965). 64
5.20. Informe de clasificación de la CNN 1D implementada en Keras en el conjunto de test (16.6k muestras). 65
5.21. Evolución del entrenamiento de la LSTM. El modelo no muestra mejoras en validación, manteniéndose en valores cercanos al azar. 67
5.22. Informe de clasificación de la LSTM implementada en Keras en el conjunto de test (16.6k muestras). 67

6.1. Top-12 coeficientes de la regresión logística: bloque EMBER, signo e interpretación. 75

Resumen

Este trabajo aborda la detección de malware en archivos ejecutables PE mediante técnicas de aprendizaje automático y explicabilidad (XAI). Se empleó el dataset EMBER 2018 para entrenar y evaluar modelos clásicos (Regresión Logística, Random Forest, LightGBM, SVM, KNN) y arquitecturas profundas (MLP y CNN), utilizando un conjunto balanceado de 50,000 muestras para entrenamiento y 16 600 para prueba. Además del rendimiento predictivo, se analizó la trazabilidad de las decisiones mediante herramientas post-hoc como SHAP y LIME. Los resultados mostraron que, aunque las métricas de precisión y F1 fueron similares entre modelos (72–82 %), las explicaciones ofrecidas por las redes neuronales resultaron más consistentes y semánticamente ricas que las de los algoritmos clásicos, al concentrarse en atributos clave como cabeceras, secciones e importaciones. La investigación concluye que la explicabilidad es el criterio diferenciador para seleccionar modelos en este dominio, siendo el *Deep Learning* combinado con XAI una alternativa prometedora para sistemas de seguridad confiables.

Palabras Clave: detección de malware, aprendizaje automático, explicabilidad, redes neuronales, SHAP

Abstract

This work addresses malware detection in PE executable files through machine learning and explainable artificial intelligence (XAI) techniques. The EMBER 2018 dataset was used to train and evaluate classical models (Logistic Regression, Random Forest, LightGBM, SVM, KNN) as well as deep learning architectures (MLP and CNN), with a balanced set of 50,000 samples for training and 16,600 for testing. In addition to predictive performance, decision traceability was analyzed using post-hoc tools such as SHAP and LIME. Results showed that, although accuracy and F1 scores were similar across models (72–82 %), neural networks provided more consistent and semantically meaningful explanations than classical algorithms, as they focused on key attributes such as headers, sections, and imports. The study concludes that explainability is the decisive criterion for selecting models in this domain, with deep learning combined with XAI emerging as a promising approach for building reliable security systems.

Keywords: malware detection, machine learning, explainability, neural networks, SHAP

1. Introducción

En el contexto actual de la digitalización, la seguridad informática se ha consolidado como un elemento esencial para proteger infraestructuras, datos y usuarios frente a amenazas cada vez más sofisticadas. Entre ellas, el malware —programas diseñados con fines maliciosos— continúa siendo uno de los principales vectores de ataque. Su capacidad para modificar su comportamiento y evadir los mecanismos de detección tradicionales plantea un desafío constante para investigadores y profesionales de la ciberseguridad.

En los últimos años, el aprendizaje profundo ha mostrado un gran potencial en este ámbito, ya que permite extraer de forma automática representaciones complejas de los archivos binarios, evitando en gran medida la necesidad de diseñar manualmente características específicas. Sin embargo, estos modelos suelen funcionar como "cajas negras", lo que dificulta comprender cómo llegan a una predicción y limita su uso en entornos críticos. Para afrontar este problema, en este trabajo se plantea la incorporación de enfoques de inteligencia artificial explicable (XAI), con el fin de construir un sistema de detección de malware que combine precisión predictiva con transparencia y fiabilidad en sus resultados.

1.1 Motivación

La detección de malware se ha convertido en un desafío estratégico, donde atacantes y defensores libran una batalla constante. El desarrollo y evolución del malware **puede compararse con una cebolla**: múltiples capas de técnicas de evasión, cifrado, ofuscación y empaquetado que dificultan enormemente su análisis. Cuando aparece un nuevo malware —el llamado día cero o *zero-day*—, este pasa inadvertido hasta que algún sistema lo detecta, generalmente por su comportamiento o por análisis manuales. A partir de ahí, se añaden firmas a antivirus o reglas a sandboxes, lo cual obliga a los atacantes a estudiar cómo fue detectado para **modificarlo y volver a evadir la defensa**. Este ciclo se repite continuamente, alimentando una carrera armamentística entre ambas partes.

En este contexto, las técnicas clásicas de detección basadas en firmas o reglas estáticas resultan cada vez más limitadas, especialmente frente a variantes polimórficas o metamórficas del malware, que modifican su código o estructura sin alterar su funcionalidad maliciosa, con el objetivo de evadir los mecanismos tradicionales de detección.

Por ello, se hace necesario un enfoque más flexible y adaptativo que permita identificar

nuevas amenazas antes de que sean formalmente registradas. Los modelos de **aprendizaje automático**, tanto en sus versiones más clásicas como en arquitecturas profundas, ofrecen una vía prometedora para abordar este problema, al permitir sistemas capaces de aprender patrones complejos directamente de los datos.

No obstante, la adopción de este tipo de técnicas plantea interrogantes importantes: ¿por qué se ha clasificado un archivo como malicioso?, ¿qué características han influido más en la decisión?, ¿es el modelo confiable en situaciones reales? Estas preguntas surgen independientemente de que el algoritmo utilizado sea un clasificador tradicional o una red neuronal profunda.

Aquí es donde entran en juego los enfoques de **inteligencia artificial explicable (XAI)**, que permiten abrir la "caja negra" de los modelos y comprender qué factores influyen en sus predicciones. Para ello existen distintas metodologías, tanto intrínsecas como post-hoc. En este trabajo se emplearán herramientas ampliamente utilizadas en la literatura, como **SHAP y LIME**. Estas librerías implementan métodos de XAI que permiten, respectivamente, descomponer la contribución de cada característica a una predicción (valores de Shapley) o aproximar localmente el comportamiento del modelo con un modelo interpretable. De este modo, no se trata de nuevas técnicas de aprendizaje en sí mismas, sino de recursos que complementan a los algoritmos ya entrenados y facilitan la interpretación de sus resultados.

1.2 Planteamiento del trabajo

La detección de malware binario constituye uno de los desafíos centrales dentro de la ciberseguridad. Los enfoques tradicionales han recurrido principalmente a algoritmos de aprendizaje automático clásico, como árboles de decisión, máquinas de vectores soporte (SVM) o bosques aleatorios. Estos métodos, basados en la extracción manual de características a partir de los ejecutables, han permitido construir sistemas de detección efectivos e incluso relativamente interpretables en determinados casos. No obstante, su capacidad de adaptación frente a nuevas variantes o técnicas avanzadas de ofuscación resulta limitada.

De manera paralela, las redes neuronales profundas han adquirido protagonismo en los últimos años. Su principal ventaja es la posibilidad de aprender representaciones complejas directamente a partir de los datos, reduciendo la necesidad de ingeniería manual.

Aunque suelen alcanzar un rendimiento superior en entornos con grandes volúmenes de información y escenarios cambiantes, presentan una desventaja importante: su opacidad. La dificultad para comprender cómo alcanzan sus predicciones genera desconfianza, especialmente cuando se aplican en ámbitos críticos de seguridad.

En este contexto, la inteligencia artificial explicable (XAI) cobra especial importancia. Su propósito es dotar de transparencia a los modelos, ofreciendo información acerca de los factores que han motivado una decisión. En este trabajo se emplearán librerías como *SHAP* y *LIME* para complementar los experimentos, de forma que los resultados obtenidos puedan ser auditados y comprendidos por analistas humanos.

1.3 Estructura del trabajo

Este trabajo se organiza en siete capítulos que cubren desde los fundamentos teóricos hasta las conclusiones y líneas futuras:

En el **Capítulo 1** se introduce el contexto del proyecto, exponiendo la motivación, el planteamiento del trabajo y la estructura del documento.

En el **Capítulo 2** se presenta el contexto y el estado del arte, abordando la evolución de la detección de malware, los enfoques basados en aprendizaje automático clásico y en redes neuronales profundas, las distintas formas de representación del malware, y los fundamentos de la inteligencia artificial explicable (XAI), con especial atención a herramientas como SHAP y LIME. También se describe el dataset EMBER, utilizado como referencia en este trabajo.

En el **Capítulo 3** se formulan los objetivos generales y específicos, junto con la metodología empleada, que incluye las fases del proyecto y las herramientas y frameworks utilizados.

El **Capítulo 4** desarrolla el proceso experimental, comenzando con la descripción y preprocesamiento del dataset EMBER, la vectorización de datos, la selección y normalización de características, y las métricas de evaluación aplicadas.

En el **Capítulo 5** se presentan los resultados obtenidos. Se analizan de forma diferenciada los modelos clásicos y los modelos de aprendizaje profundo, discutiendo su rendimiento, el impacto del preprocesamiento, la relación entre interpretabilidad y precisión, así como las limitaciones del enfoque propuesto.

El **Capítulo 6** está dedicado íntegramente a la explicabilidad de los modelos, introdu-

ciendo las técnicas de XAI y aplicándolas tanto a los algoritmos clásicos como a las redes neuronales profundas.

Finalmente, en el **Capítulo 7** se recogen las conclusiones principales del trabajo y se plantean posibles líneas de investigación futura.

2. Contexto y Estado del Arte

2.1 Contexto del problema

Hoy en día, el malware continúa siendo una de las amenazas más persistentes en el ámbito de la ciberseguridad. Su capacidad para mutar, camuflarse y evadir los mecanismos de detección tradicionales ha obligado a replantear las estrategias defensivas. Durante mucho tiempo, las soluciones antivirus se basaron en firmas y reglas heurísticas, eficaces frente a muestras conocidas. Sin embargo, frente a variantes inéditas o ataques de tipo *zero-day*, estas aproximaciones evidencian limitaciones significativas.

En este contexto, la inteligencia artificial —y en particular el aprendizaje automático— ha comenzado a desempeñar un papel cada vez más relevante. Estos modelos son capaces de aprender patrones complejos a partir de grandes volúmenes de datos, lo que les permite reconocer comportamientos sospechosos incluso en archivos no catalogados previamente. Dentro de este campo, las técnicas de aprendizaje profundo han mostrado un potencial destacado, ya sea trabajando directamente con secuencias de bytes o con características previamente extraídas.

No obstante, su aplicación práctica plantea retos adicionales. Tanto los modelos clásicos como las redes neuronales profundas suelen percibirse como una "caja negra": generan predicciones, pero ofrecen poca claridad sobre las razones detrás de ellas. Esta falta de transparencia dificulta el trabajo de los analistas, que necesitan validar las decisiones del sistema e identificar los factores que influyen en cada clasificación.

Para abordar esta carencia, han cobrado protagonismo los métodos de **inteligencia artificial explicable** (XAI, por sus siglas en inglés). Herramientas como *SHAP* y *LIME* permiten interpretar las salidas de los modelos, destacando qué variables o patrones han tenido mayor peso en una decisión concreta. De este modo, el reto actual no consiste únicamente en entrenar modelos más precisos para la detección de malware, sino también en dotarlos de mecanismos que hagan su funcionamiento comprensible y justificable para los profesionales de ciberseguridad.

2.2 Estado del arte

La investigación en detección de malware ha experimentado un notable progreso en los últimos años, evolucionando desde enfoques clásicos hacia el uso de técnicas de aprendizaje automático y, más recientemente, de aprendizaje profundo. Paralelamente, ha aumentado el interés por la interpretabilidad de los modelos, aspecto clave para su adopción en entornos críticos. En las secciones siguientes se presentan los avances más representativos publicados entre 2020 y 2025, organizados en diferentes apartados temáticos. Cada uno de ellos recoge trabajos relevantes, poniendo énfasis en sus aportaciones técnicas, los resultados obtenidos y las principales fortalezas y limitaciones identificadas.

2.2.1 Evolución de la detección de malware

A lo largo de las últimas décadas, las estrategias para detectar malware han cambiado de manera significativa. En sus inicios, los sistemas antivirus se apoyaban casi exclusivamente en firmas, es decir, patrones de bytes previamente conocidos que se comparaban de forma directa con los ejecutables. Este método resultaba eficaz frente a muestras repetitivas, pero se volvía insuficiente ante variantes desconocidas o técnicas de ofuscación. Con el tiempo, se introdujeron enfoques heurísticos y de análisis de comportamiento —basados en la ejecución controlada de los programas— que ampliaron las capacidades de detección. Posteriormente, el campo avanzó hacia la aplicación de modelos de aprendizaje automático.

En la última década, el incremento en la disponibilidad de datos y la aparición de hardware especializado (como GPU o TPU) han favorecido la adopción del aprendizaje profundo en este ámbito. Tal y como señalan (Song y cols., 2025), desde los primeros trabajos de principios de 2010 la comunidad científica ha ido orientando su atención hacia estas técnicas, observándose entre 2015 y 2020 un fuerte crecimiento en el uso de modelos profundos, que han logrado resultados muy competitivos.

Actualmente, la literatura suele distinguir tres grandes categorías de detección: *estática* (basada en firmas o estructuras del binario), *dinámica* (centrada en el comportamiento en ejecución) e *híbrida* (combinación de ambas). Los métodos modernos tienden a integrar estas aproximaciones con análisis de características más elaboradas —como trazas de llamadas al sistema o grafos de programa— junto con algoritmos de inteligencia artificial. En una revisión reciente, (Gaber, Ahmed, y Janicke, 2024) destacan que el rendimiento

depende en gran medida de la calidad de las características empleadas y del tipo de análisis realizado: el enfoque estático es más rápido, pero sensible a la ofuscación; el dinámico es más robusto, aunque conlleva un mayor coste computacional y es vulnerable a técnicas anti-*debugging*. A ello se suman retos de fondo como el desbalance de clases, la falta de datasets públicos actualizados, el *concept drift* (evolución constante del malware) o los ataques adversarios.

Entre los estudios con perspectiva histórica, (Aboaoja y cols., 2022) proponen una taxonomía que clasifica los enfoques en estáticos, dinámicos, híbridos y basados en aprendizaje automático o profundo, resaltando las fortalezas y debilidades de cada uno. Asimismo, muestran cómo la incorporación de técnicas de inteligencia artificial ha permitido mejorar de forma notable las tasas de detección en comparación con las aproximaciones basadas únicamente en firmas. De hecho, los autores ilustran esta evolución mediante una figura que recopila artículos publicados entre 2003 y 2022, donde se aprecia un incremento pronunciado a partir de 2018, coincidiendo con la adopción masiva de modelos profundos (véase la Figura 2.1).

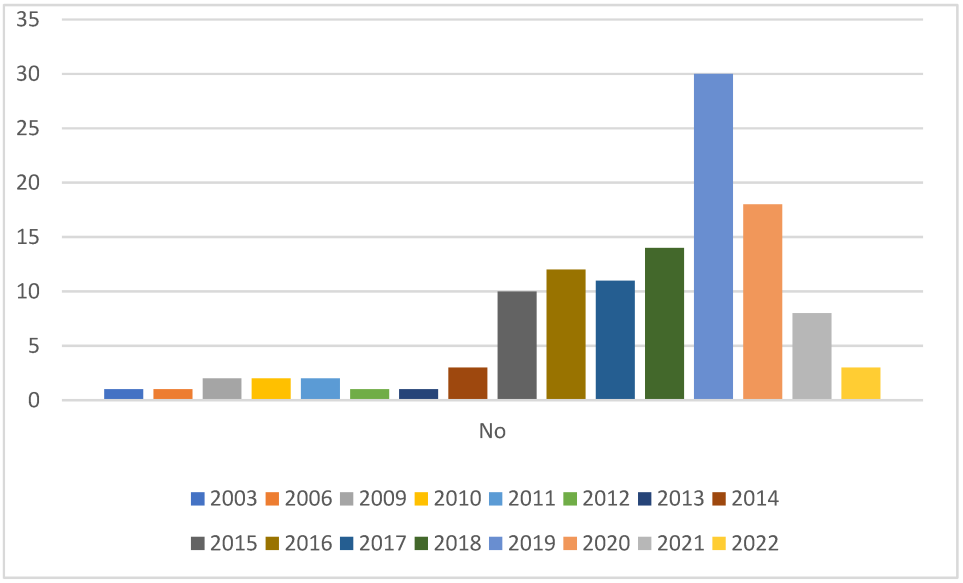


Figura 2.1: Distribución de los artículos considerados en el periodo 2003–2022, según Abaoja y colaboradores en 2022.

Aun con estos avances, los modelos actuales presentan nuevos retos. Se necesitan datasets más amplios y actualizados, técnicas más sólidas frente a malware altamente ofuscado y, sobre todo, modelos más interpretables. La complejidad añadida del aprendizaje pro-

fundo ha hecho que la falta de explicabilidad se convierta en un problema central, ya que limita la confianza en entornos donde las decisiones deben ser comprensibles y justificables.

2.2.2 Detección basada en aprendizaje automático clásico

El aprendizaje automático clásico —esto es, modelos no "profundos" como árboles de decisión, máquinas de vectores soporte (SVM) o bosques aleatorios— ha tenido un papel muy relevante en la detección de malware. Normalmente se ha trabajado extrayendo de manera manual características estáticas o dinámicas de los binarios, que luego sirven de entrada a los clasificadores. Entre 2020 y 2025 se han publicado numerosos estudios que comparan estos algoritmos, a menudo en combinación con distintas técnicas de ingeniería de características.

Un ejemplo representativo es el trabajo de (Ferdous, Islam, Mahboubi, y Islam, 2025), donde se revisa de manera sistemática la literatura reciente sobre machine learning aplicado a la detección de malware en diversas plataformas: desde ordenadores personales hasta dispositivos móviles, entornos IoT o servicios en la nube. Frente a revisiones más centradas en un único ámbito, este estudio ofrece una visión global que permite identificar similitudes y diferencias según el contexto. La revisión clasifica los enfoques en función del algoritmo utilizado (árboles, SVM, k-NN, etc.), del tipo de características (permisos, llamadas API, opcodes, tráfico de red...) y del escenario de aplicación (detección en tiempo real, prevención, clasificación multicategoría). Entre sus conclusiones, destacan que los algoritmos clásicos siguen siendo muy competitivos en entornos con recursos limitados, como IoT, donde pesan su bajo coste computacional y facilidad de interpretación. En cambio, en escenarios más cambiantes como el cloud computing se requiere complementar estos modelos con mecanismos de actualización continua. También señalan problemas aún abiertos, como la escasez de datasets bien equilibrados para cada plataforma y la necesidad de algoritmos capaces de combinar eficiencia con capacidad de generalización.

De forma complementaria, (Wu, Zhuang, Jia, y Zhang, 2025) realizan una revisión centrada en la evolución del machine learning para la detección de malware. Su principal aportación es una comparación sistemática entre algoritmos clásicos y modelos profundos, analizando tanto sus ventajas como sus limitaciones prácticas. Evalúan métodos como regresión logística, árboles de decisión, random forests o SVM en escenarios estáticos (basados en características de los binarios) y dinámicos (trazas de ejecución, llamadas al sistema). Según el estudio, los enfoques tradicionales presentan limitaciones de escala-

bilidad y abstracción frente a las redes profundas, lo que explica en parte la transición hacia estas últimas. Sin embargo, Wu subraya que los algoritmos clásicos siguen siendo preferibles cuando prima la interpretabilidad o la rapidez, como en la detección temprana en endpoints o en situaciones de respuesta rápida. El artículo además anticipa un futuro híbrido: modelos clásicos, arquitecturas profundas y LLM conviviendo en sistemas que busquen un equilibrio entre explicabilidad, robustez y capacidad de adaptación.

En el ámbito Android, (Wajahat, Murtaza, Bakhshi, y cols., 2024) combinaron una exhaustiva extracción de características (permisos, llamadas API, intents, etc., obtenidas con herramientas como Dex2Jar) con algoritmos clásicos. Tras aplicar técnicas de selección (gain ratio, chi-square), lograron que un Random Forest y una SVM superaran el 98 % de precisión en datasets estándar como Drebin y TUANDROMD. Este caso demuestra que, con un buen conjunto de características, los modelos clásicos pueden alcanzar resultados muy competitivos.

Otra línea destacada son los sistemas ligeros e interpretables. En (Madamidola, Ngobigha, y Ez-zizi, 2025), por ejemplo, se propone un detector basado en random forests entrenado únicamente con las cinco características más relevantes, capaz de identificar variantes de malware ofuscado no vistas en el entrenamiento. A pesar de entrenarse con muestras de una sola familia de spyware (Transponder) y validarse en 15 subtipos diferentes, el modelo alcanzó más del 99,8 % de precisión, incluso detectando familias nuevas. El enfoque se complementa con el uso de herramientas de explicabilidad (SHAP) para mostrar qué rasgos justifican cada clasificación, reforzando su valor práctico para analistas.

En general, los algoritmos clásicos destacan por su simplicidad y rapidez, además de ofrecer modelos que suelen ser más fáciles de interpretar. Con técnicas adecuadas de selección de atributos y balanceo de datos, métodos como Random Forest pueden ofrecer resultados comparables a redes profundas, pero con un coste computacional mucho menor. (Galli, La Gatta, Moscato, Postiglione, y Sperlì, 2024), por ejemplo, observaron que un clasificador basado en árboles de decisión se entrena y explica más rápido que una red neuronal recurrente. También es común que los métodos basados en ensambles (bagging, boosting) muestren gran robustez frente a datos ruidosos.

No obstante, estos enfoques arrastran limitaciones. Requieren ingeniería manual de características, dependiente del conocimiento experto, y tienen dificultades para manejar datos en bruto como secuencias de bytes muy largas. Su rendimiento también se degrada cuando aumenta la dimensionalidad o cuando se enfrentan a malware con técnicas de

evasión novedosas, escenarios en los que las redes profundas suelen manejar mejor la complejidad. De ahí que, pese a sus fortalezas, los modelos clásicos convivan cada vez más con arquitecturas neuronales en soluciones modernas de detección.

2.2.3 Redes neuronales profundas aplicadas a malware

El uso de redes neuronales profundas ha impulsado el estado del arte en detección de malware, permitiendo aprender patrones automáticamente de los datos sin requerir una definición manual exhaustiva de features. Entre 2020 y 2025 se han propuesto numerosos modelos profundos para distintos tipos de representaciones (ver apartado 2.2.4), tales como CNNs (redes convolucionales) para entradas en forma de imagen o byte, RNNs/LSTMs para secuencias de instrucciones o API, y más recientemente transformers y GNNs (redes neuronales gráficas).

En Android, (Aamir, Ali, Ishmanov, y cols., 2024) introdujeron el modelo AMDDL, una arquitectura basada en CNN diseñada para detectar malware en smartphones. Con 215 características estáticas del dataset Drebin como entrada, su CNN logró más del 99 % de precisión, superando métodos previos. Esta arquitectura profunda demostró ser capaz de distinguir con alta exactitud entre aplicaciones benignas y malware en Android, indicando la eficacia de las CNN en capturar patrones locales de bytes/opcodes. La Figura 2.2 ilustra el proceso seguido por AMDDL, donde los binarios son transformados en imágenes que se usan como entrada para una red convolucional profunda.

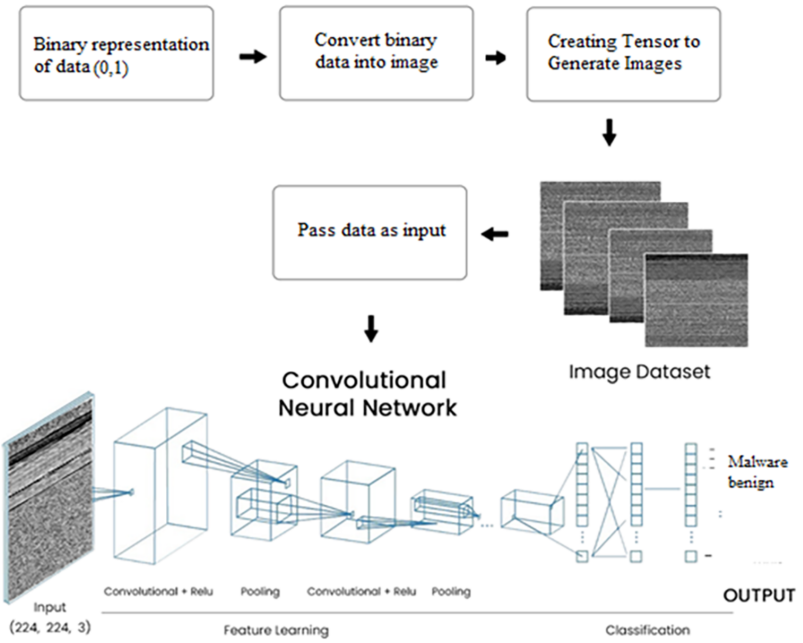


Figura 2.2: Arquitectura del modelo AMDDL: conversión de binarios en imágenes para clasificación mediante CNN.

Por su parte, (Lu, Zhao, Zhu, y Liò, 2024) enfocan la detección desde la perspectiva de las llamadas a funciones en Android: propusieron SNDGCN, que combina Graph Convolutional Networks con técnicas de desruido (denoising) para analizar el grafo de llamadas de función (FCG) de una app. Su modelo alcanzó las mejores métricas de accuracy, recall y F1-score en su evaluación, mostrando robustez incluso frente a ruido o código añadido para evadir detección. La fortaleza de este enfoque radica en captar la estructura de llamadas y flujo de programa del malware mediante una red profunda sobre grafos, lo que mejora la generalización a variantes con comportamiento similar.

Un desarrollo muy reciente es la incorporación de modelos transformer y técnicas de lenguaje natural a la detección de malware. (Rahali y Akhloufi, 2023) presentan MalBERTv2, un modelo basado en la arquitectura BERT entrenado sobre código fuente de aplicaciones, para clasificación de malware Android. Su modelo alcanzó puntuaciones F1 de hasta 99 % en ciertos conjuntos de datos, demostrando que los transformers pre-entrenados pueden capturar representaciones muy ricas del comportamiento de una aplicación (por ejemplo, relaciones semánticas entre secuencias de código) y detectar malware con alta precisión.

En esta misma línea, (Gupta y cols., 2025) proponen un enfoque innovador que combina técnicas de procesamiento de lenguaje natural (NLP) con redes neuronales profundas

para la clasificación de malware. La idea central de su modelo consiste en extraer información semántica de las llamadas a funciones de kernel realizadas por el software, representándolas como secuencias que posteriormente son procesadas mediante operaciones de convolución. Además, incorporan una etapa de transformación de segmentos de código en representaciones visuales (imágenes en escala de grises), que luego son tratadas por capas convolucionales para captar patrones espaciales. Finalmente, la red aplica una función sigmoide para la clasificación binaria (malicioso vs. benigno). Para validar su propuesta, los autores emplearon un conjunto de datos basado en llamadas API de kernel, evaluando la eficacia del modelo con métricas de precisión y exactitud. Los resultados muestran que esta aproximación supera las limitaciones de los métodos tradicionales basados en firmas, alcanzando altos niveles de rendimiento en la detección de variantes complejas de malware. La principal aportación del estudio radica en demostrar que la combinación de **NLP y deep learning** permite capturar características semánticas y estructurales del malware, lo que incrementa la capacidad de generalización y la adaptabilidad frente a nuevas amenazas.

De forma complementaria, (Johny y cols., 2024) exploran un paradigma multimodal en el que se fusionan distintas representaciones visuales del malware para alimentar un sistema de deep learning. En concreto, los autores convierten los binarios en múltiples vistas gráficas: imágenes en escala de grises, grafos de entropía y representaciones SimHash. Estas vistas son integradas en una arquitectura de red neuronal profunda que aprovecha la complementariedad entre fuentes de información, logrando capturar tanto patrones globales como locales del malware. Uno de los aspectos más destacados de este trabajo es el uso de técnicas de explicabilidad basadas en Grad-CAM, que permiten identificar regiones críticas de las imágenes que influyen en la decisión del modelo, facilitando así la validación por parte de analistas humanos. Los experimentos, realizados sobre un dataset amplio y heterogéneo, reportan una tasa de detección cercana al 100 % en pruebas controladas, lo que muestra el potencial de la fusión de características visuales para robustecer los modelos de detección. Aunque los autores reconocen que este rendimiento tan elevado puede estar influenciado por las condiciones experimentales, su investigación abre la puerta al uso de enfoques híbridos de visión por computador y deep learning para mejorar la detección y explicabilidad del malware.

Las redes profundas pueden aprender características complejas directamente de los datos en crudo (bytes, secuencias, gráficas), eliminando la necesidad de definir manual-

mente indicadores. Esto les permite descubrir patrones sutiles o de alto nivel que podrían escapar a métodos clásicos. Los resultados citados (precisiones cercanas o superiores al 99 % en varios casos) reflejan el alto rendimiento de estos modelos cuando se entrenan con suficientes datos. Además, la diversidad de arquitecturas disponibles (CNN, RNN, GCN, Transformers) brinda flexibilidad para modelar malware bajo distintas representaciones. Sin embargo, su principal inconveniente radica en la opacidad y complejidad: son modelos de caja negra con millones de parámetros, difíciles de interpretar. También requieren grandes volúmenes de datos balanceados para entrenar adecuadamente, y pueden ser vulnerables a técnicas de evasión adversaria (por ejemplo, pequeños cambios en el input pueden engañar a la red). Algunos estudios (Sun y cols., 2025) evidencian que un ataque adversarial puede reducir la exactitud de MalConv (un CNN de bytes) de $> 90\,\%$ a $< 2\,\%$, lo cual alerta sobre la fragilidad potencial de los modelos profundos ante malware especialmente diseñado para evadirlos. En suma, las redes neuronales profundas ofrecen un salto en capacidad de detección, pero a costa de nuevos desafíos en robustez e interpretabilidad.

2.2.4 Representaciones del malware (imágenes, bytes, secuencias)

Un tema muy investigado hoy en día es cómo representar los binarios de malware para que puedan ser entendidos por los modelos de IA. Además de las formas clásicas (como mirar las APIs usadas, permisos o los opcodes más comunes), se han creado otras formas de representación que hacen que el malware se convierta en un formato más fácil de manejar con machine learning. Entre las más usadas están: transformar el binario en una imagen, trabajar con la secuencia cruda de bytes o de opcodes, o construir secuencias de llamadas a APIs o trazas de comportamiento. Estas formas de representación, sobre todo si se combinan con técnicas de extracción de características, han demostrado mejorar la detección e incluso ayudar a descubrir nuevas familias de malware.

La idea de visualizar malware como imagen apareció alrededor de 2011 (Nataraj, Karthikeyan, Jacob, y Manjunath, 2011), y en los últimos años se ha refinado bastante. La lógica es tomar los bytes de un ejecutable y verlos como valores de píxeles en escala de grises (u otros formatos), para luego aplicar métodos de visión por computador. Más adelante, (Deng, Guo, Shen, Cui, y Ping, 2023) propusieron MCTVD, un método para visualizar malware usando tres canales de color. Su propuesta analiza qué factores son importantes para que la imagen conserve los patrones maliciosos y plantean mezclar diferentes fuentes de información (como histogramas de bytes o matrices de transición de

Markov) en una sola imagen RGB. Evaluaron su método en el Microsoft Malware Classification Dataset y obtuvieron mejores resultados que otros enfoques previos, logrando muy buenas tasas de clasificación entre 9 familias de malware. Lo potente de usar imágenes es que patrones de bytes o de opcodes, que en secuencia son difíciles de notar, pueden aparecer como texturas o regiones visibles que las CNN detectan fácilmente. (Nazim y cols., 2025) llevaron este enfoque un paso más allá con explicabilidad: entrenaron un modelo en imágenes de malware y aplicaron SHAP, LIME y Grad-CAM para ver qué regiones de la imagen (y por ende qué partes del binario) eran las que más aportaban a la detección. Sus resultados no solo alcanzaron más del 99 % de precisión, sino que además comprobaron que las zonas resaltadas por la IA coincidían con las secciones PE y patrones de código que un analista humano también consideraría sospechosos, lo cual aumenta la confianza en el modelo.

Otra línea importante es tratar el malware como secuencias de bytes crudos. Una forma de hacerlo es trabajar con las secuencias de instrucciones u opcodes tras desensamblar el binario. (Kakisim, Gulmez, y Sogukpinar, 2022) propusieron SOEMD (Sequential Opcode Embedding-based Malware Detection), que busca patrones dentro de secuencias muy largas de opcodes. Como esas secuencias pueden llegar a cientos de miles de instrucciones, aplican un random walk en el grafo de flujo del programa para obtener subsecuencias, y luego generan embeddings de baja dimensión para representarlas. Con esto lograron 100 % de detección en sus pruebas, superando a métodos base. Aunque ese resultado tan alto puede deberse en parte a sobreajuste en un dataset pequeño, muestra el potencial de combinar reducción de secuencia con word embeddings para código. De forma parecida, (Tupadha y Stamp, 2021) usaron modelos ocultos de Markov (HMM) y embeddings como Word2Vec o HMM2Vec para analizar secuencias de opcodes, pero no tanto para clasificar malware frente a software benigno, sino para estudiar cómo evolucionan las familias de malware a lo largo del tiempo. Sus resultados muestran que los embeddings reflejan cambios relevantes en la distribución de instrucciones, lo que ayuda a detectar nuevas variantes. Este uso es interesante porque va más allá de la detección estática: sirve para hacer seguimiento de la evolución del malware, que es clave ya que cambia muy rápido.

También están las secuencias de llamadas a API o de actividad dinámica, que ofrecen otra perspectiva. Algunos trabajos generan trazas de comportamiento (por ejemplo, la secuencia de llamadas al sistema durante la ejecución en una sandbox) y luego las procesan con redes recurrentes o transformers. (Song y cols., 2025) propusieron un modelo profundo

que convierte estas secuencias de llamadas API en representaciones densas para luego clasificarlas. Su método alcanzó entre un 98 y un 99 % de precisión frente a modelos más generales como VGG o ResNet, mostrando que una arquitectura especializada en secuencias puede ser tan buena o mejor que enfoques de visión por computador. Además, notaron que tener en cuenta el orden y la dependencia temporal de las llamadas es clave para diferenciar malware, lo cual concuerda con lo que suele pensar un analista: muchos tipos de malware (como el ransomware) siguen secuencias de llamadas muy características, por ejemplo abrir, leer y escribir muchos archivos rápidamente.

En conclusión, estas representaciones alternativas permiten que los modelos de IA aprovechen mejor la información que da el malware. Usar imágenes saca partido de los avances en visión por computador, trabajar con secuencias de bytes o instrucciones conecta con técnicas de NLP para "aprender el lenguaje del malware", y las secuencias de APIs permiten capturar comportamientos dinámicos. Los trabajos revisados muestran mejoras claras: visualizaciones que facilitan la clasificación de familias, embeddings de opcodes que alcanzan detecciones casi perfectas en ciertos contextos, o secuencias de APIs que reflejan patrones de comportamiento. Además, algunas de estas representaciones dan más interpretabilidad: por ejemplo, para un analista es más intuitivo ver un mapa de calor sobre la imagen de un binario que le indique qué zona es sospechosa, en lugar de intentar interpretar los pesos de una red.

Eso sí, no usamos esta técnica en este trabajo porque estas transformaciones pueden ser pesadas o complicadas de generar (por ejemplo, correr cada muestra en una sandbox para extraer las llamadas lleva mucho tiempo, o desensamblar perfectamente un binario puede fallar si está empaquetado). También está el problema de la pérdida de información: si la representación no logra capturar cierto indicador (como instrucciones específicas que en una imagen se pierden), el modelo podría pasarlo por alto. Algunos métodos, como MCTVD, añaden complejidad al combinar múltiples visualizaciones, pero aun así podrían ser engañados por malware diseñado para romper esos supuestos. En resumen, elegir cómo representar el malware es un paso crítico: estas alternativas han mostrado mejoras grandes en detección, pero siempre hay que probarlas en distintos escenarios para asegurar que sean realmente robustas.

2.2.5 Técnicas de inteligencia artificial explicable (XAI)

A medida que los modelos de detección de malware se han vuelto más complejos (especialmente con AA profundo), ha surgido la necesidad imperante de explicar sus decisiones. Las técnicas de Inteligencia Artificial Explicable (XAI) buscan proporcionar interpretaciones comprensibles del comportamiento de modelos de caja negra, bridando transparencia y facilitando la confianza de los analistas en las herramientas de detección. En ciberseguridad, la explicabilidad no solo aumenta la confianza en las alertas, sino que también ayuda a comprender por qué un binario fue marcado como malicioso, lo cual puede revelar indicadores de compromiso accionables. Entre 2020 y 2025, varios trabajos se han enfocado en adaptar y evaluar métodos XAI en el contexto de malware y seguridad.

Visión general de XAI: (Manthena y cols., 2024) ofrecen un panorama actual de XAI para malware. Destacan que los modelos de AA logran alta precisión pero sufren de falta de transparencia, lo que dificulta su adopción en entornos de seguridad críticos donde es esencial entender la razón de cada detección. XAI pretende cerrar esta brecha proporcionando explicaciones sin sacrificar mucho el rendimiento. Las técnicas XAI se dividen en dos grandes grupos: intrínsecas (usar modelos inherentemente interpretables, como árboles de decisión, aunque generalmente con menor precisión) y post-hoc (aplicar métodos de explicación a modelos ya entrenados, típicamente de alta precisión pero opacos). La mayoría de trabajos recientes en malware se centran en enfoques post-hoc model-agnostic, que tratan al modelo como caja negra y explican sus predicciones después. Dentro de estos, se distinguen explicaciones globales vs. locales. Una explicación global busca describir qué factores en general influyen más en el modelo (p.ej., "en este detector, las llamadas a CreateProcess y la presencia de ofuscación en cadenas son las características más importantes para decidir malware"). Por otro lado, una explicación local se refiere a por qué una instancia particular fue clasificada de cierta forma (p.ej., "el archivo X fue clasificado como ransomware porque hizo llamadas criptográficas y operaciones de borrado de backups, contribuyendo decisivamente a la predicción"). Ambos tipos son útiles: los analistas de malware pueden apreciar explicaciones globales para conocer las "reglas" generales del modelo, y explicaciones locales para cada alerta específica.

Métodos XAI populares: Entre las herramientas post-hoc más empleadas en seguridad están las explicaciones por aproximación local como LIME y las atribuciones de importancia de características como SHAP. También se usan métodos específicos de redes

neuronales, como Grad-CAM (en modelos de visión), LRP (layer-wise relevance propagation) o incluso aprovechar los pesos de atención en modelos que los tengan. (Galli y cols., 2024) integran cuatro métodos XAI (SHAP, LIME, LRP y attention) en un marco de detección de malware basado en comportamiento (llamadas API), aplicándolos a modelos LSTM y GRU entrenados en múltiples datasets. Su estudio comparativo encontró que cada método tiene fortalezas: por ejemplo, LRP (que retropropaga la contribución de cada neurona) ofreció explicaciones más consistentes ligadas a características de entrada específicas, mientras que LIME y SHAP proporcionaron mayor intuitividad a los analistas al resaltar las APIs más relevantes en cada decisión. Sin embargo, también notaron diferencias en costos computacionales: LIME puede ser más rápido pero menos preciso en algunos casos, mientras SHAP tiende a ser más estable pero computacionalmente intenso, y la interpretación de los mapas de atención requiere entender la arquitectura del modelo. En general, recomiendan combinar varios métodos para obtener una imagen completa (por ejemplo, usar SHAP para visión global de features importantes a nivel dataset, y LIME para explicaciones locales caso a caso).

Otro aspecto estudiado es cómo XAI puede exponer vulnerabilidades. Curiosamente, (Saqib, Mahdavifar, Fung, y Charland, 2024) advierten que aunque los modelos de Detección de Malware Explicable (EMD) aumentan la interpretabilidad, paradójicamente pueden abrir la puerta a atacantes para explotar ese conocimiento y generar malware adversarial. En su análisis señalan que publicar qué features o patrones utiliza un detector (por ejemplo, "el modelo presta mucha atención a las cadenas de importación") podría permitir a creadores de malware evadirlo manipulando esas características. Es decir, la XAI en ciberseguridad conlleva un delicado equilibrio: demasiada información sobre el modelo puede facilitar ataques (haciéndolo más vulnerable a adversarios informados), pero muy poca interpretabilidad resulta inaceptable para los defensores. Este punto resalta la necesidad de enfoques XAI robustos que no revelen detalles explotables o de definir qué nivel de explicación es seguro compartir.

En resumen, los métodos de XAI aplicados a malware ofrecen transparencia y capacidad de auditoría a herramientas de AA que, de otro modo, serían cajas negras. Esto ayuda a ganar confianza de los analistas y facilita la incorporación de conocimiento de dominio: por ejemplo, si un modelo señala que ciertas secciones del binario motivan la clasificación, un analista puede verificar si corresponden a shellcode conocido o a comportamiento malicioso esperado, consolidando la alerta. Además, XAI puede descubrir falsos positivos o sesgos;

varios trabajos reportan casos donde las explicaciones revelaron que el modelo se fijaba en rasgos no deseados (como el tamaño del archivo o un certificado), permitiendo corregirlo.

No obstante, las técnicas XAI no son infalibles. Muchas proporcionan aproximaciones de la explicación y pueden ser engañosas si se interpretan mal. Por ejemplo, con LIME se genera un modelo lineal local que no siempre refleja perfectamente la complejidad del modelo real en esa vecindad; con SHAP se asume contribuciones aditivas que pueden no capturar interacciones complejas de features. Además, en malware a veces las explicaciones son difíciles de validar: si un modelo profundo indica que cierta combinación de bytes es maliciosa, puede requerir bastante experticia entender esa combinación a nivel de código máquina. Finalmente, está el costo: aplicar XAI puede ser computacionalmente pesado (e.g., calcular valores SHAP para un solo binario puede requerir cientos de predicciones del modelo). A pesar de ello, la tendencia en la comunidad es que el beneficio de la explicabilidad supera estos inconvenientes, y se están desarrollando métodos más eficientes y específicos para ciberseguridad. En la siguiente sección, se examinan en detalle las aplicaciones de dos herramientas XAI muy extendidas – SHAP y LIME – en contextos de ciberseguridad.

2.2.6 Aplicaciones de SHAP y LIME en ciberseguridad

SHAP (Shapley Additive Explanations) y LIME (Local Interpretable Model-Agnostic Explanations) se han consolidado como dos de las herramientas XAI más usadas en ciberseguridad, debido a su capacidad para explicar predicciones de prácticamente cualquier modelo. Numerosos trabajos recientes las emplean para interpretar detectores de malware, sistemas de detección de intrusos (IDS) y otros clasificadores de seguridad, con resultados prometedores.

En detección de malware, SHAP y LIME se utilizan para desglosar la contribución de distintas características (por ejemplo, permisos de Android, n-gramas de bytes, llamadas API) a la decisión de un modelo. (Galli y cols., 2024), ya citado, evaluó ambas técnicas sobre modelos de secuencias API. Encontraron que SHAP tendía a proporcionar explicaciones más consistentes globalmente, identificando cuáles API y eventos tenían influencia positiva o negativa en la clasificación en general. LIME, por su parte, fue útil a nivel local: generó explicaciones inmediatas por muestra, destacando, por ejemplo, que para el archivo maligno.exe, las llamadas a RegOpenKey y WriteFile fueron las que más peso tuvieron en su clasificación como malware. Un hallazgo interesante fue que las explicaciones de SHAP

y LIME frecuentemente coincidían en las features críticas, lo que aumenta la confianza en la explicación (si dos métodos distintos resaltan la misma señal maliciosa, es menos probable que sea un artefacto). No obstante, también reportan casos donde difirieron: LIME a veces marcó una característica como importante en una muestra, mientras SHAP la consideró irrelevante; esto puede deberse a la variación aleatoria de LIME o a interacciones de características que SHAP pudo capturar.

(Nazim y cols., 2025) utilizaron SHAP y LIME para un modelo de clasificación de malware por imágenes (como ya mencionamos anteriormente), encontrando que ambas técnicas podían "aclarar las contribuciones de características individuales" a las predicciones, aumentando la confianza del analista. Con Grad-CAM señalaron regiones específicas de las imágenes de malware clave para la red, mientras que con SHAP y LIME permitieron traducir eso a términos de features originales (e.g., "la sección .text y el patrón de bytes X son relevantes"). Sus resultados mostraron que las explicaciones concordaban con el conocimiento de dominio: malware de cierta familia era detectado principalmente por la presencia de un payload en su sección de recursos, y efectivamente SHAP asignó altas importancias a bytes característicos en la sección .rsrc de esos binarios. Esto no solo valida el modelo, sino que provee información accionable – por ejemplo, un analista podría buscar ese patrón en otros archivos.

En el campo de detección de intrusiones y análisis de tráfico malicioso, SHAP y LIME también han sido utilizados con éxito. (Gaspar, Silva, y Silva, 2024) estudiaron la aplicabilidad de LIME y SHAP para explicar las decisiones de un modelo MLP en un IDS de IoT. Demostraron que podían obtener explicaciones locales de por qué ciertos flujos de red fueron marcados como ataques, identificando qué atributos (puertos, secuencias de paquetes, banderas TCP, etc.) impulsaron la alerta. Además, realizaron un experimento de perturbación: una vez que SHAP destacó, por ejemplo, que un alto número de paquetes SYN contribuyó a clasificar un flujo como "scan", modificaron ese valor y observaron que el modelo cambiaba su predicción a benigno, confirmando así que la explicación estaba señalando efectivamente una causa real y no espuria. Este tipo de validación refuerza la utilidad de SHAP/LIME en un entorno operativo: permite a los expertos comprender qué diferencia un tráfico malicioso de uno legítimo según la IA, y también detectar posibles falsos positivos (si SHAP indica que la única razón de una alerta es algo irrelevante, puede ser caso de umbral mal ajustado).

SHAP y LIME son flexibles (se pueden aplicar a cualquier modelo como "caja negra")

y proveen salidas intuitivas – típicamente un ranking de características con sus contribuciones positivas o negativas a la clasificación. Esto ha resultado ser inmensamente valioso en ciberseguridad, porque traduce las decisiones de modelos complejos a un lenguaje similar al que usaría un analista humano ("este archivo fue malware principalmente porque realiza A, B y C"). Muchas investigaciones reportan que el uso de estas explicaciones mejora la colaboración humano-IA: por ejemplo, analistas menos experimentados pueden entender y aprender del modelo mediante las explicaciones, mientras que analistas veteranos pueden validar y corregir al modelo. Además, como mostraron (Letoffe, Huang, y Marques-Silva, 2024), las técnicas de explicabilidad también pueden utilizarse para evaluar la robustez del modelo: si ciertas *features* dominan excesivamente las decisiones, el modelo podría estar sobreajustado o ser vulnerable a manipulaciones adversarias. SHAP y LIME son librerías que permiten detectar estos patrones e identificar cuándo una predicción se basa en señales frágiles o espurias, contribuyendo así a mejorar la fiabilidad del sistema.

Un inconveniente práctico es el costo computacional de SHAP, especialmente en conjuntos de datos grandes o modelos muy complejos – aunque existen aproximaciones más rápidas, SHAP clásico requiere calcular muchas combinaciones de features. LIME, si bien más ligero, introduce cierta aleatoriedad y su estabilidad depende de parámetros (tamaño de muestra local, etc.), lo que significa que diferentes corridas pueden dar variaciones en la explicación. Otra posible debilidad es que, si las características de entrada no son fácilmente interpretables por sí mismas (ej: vectores embebidos de 200 dimensiones), entonces explicar en términos de esas dimensiones es poco esclarecedor; en tales casos se necesita mapear de vuelta a algo de alto nivel (como piezas de código, secuencias de acciones) para que SHAP/LIME sean útiles. Por último, reforzando la advertencia de (Saqib y cols., 2024), existe la cuestión de qué hacer con las explicaciones en entornos compartidos: revelar demasiada información de SHAP/LIME a los usuarios podría, en teoría, filtrar pistas a atacantes sobre cómo evitar la detección. Este riesgo puede mitigarse limitando la explicabilidad hacia el analista humano interno y no en productos de cara al público.

En conclusión, SHAP y LIME han probado ser herramientas valiosas para dotar de transparencia a los sistemas de detección de malware e intrusiones de última generación. Su uso en investigaciones recientes evidencia una tendencia hacia modelos de ciberseguridad explicables, donde no solo importa qué tan bien detecta el modelo, sino por qué lo hace. Esto redunda en mayor confianza, mejores capacidades forenses (al identificar indicadores clave que justifican una alerta) y potencialmente modelos más robustos y jus-

tos. La integración de XAI en ciberseguridad aún enfrenta retos (esfuerzo computacional, garantizar que las explicaciones sean correctas y no engañosas, proteger las explicaciones de uso malicioso), pero el consenso de la comunidad es que avanzar en esta dirección es imprescindible para la próxima generación de herramientas de defensa cibernética.

2.2.7 El dataset EMBER: referencia para la detección de malware

Uno de los conjuntos de datos más utilizados para entrenar y evaluar modelos de aprendizaje automático en detección de malware en archivos PE (Portable Executable) es el dataset **EMBER** (*Endgame Malware Benchmark for Research*) introducido por (Anderson y Roth, 2018). EMBER se consolidó como un estándar de facto debido a su tamaño, disponibilidad pública y equilibrio entre muestras benignas y maliciosas.

Este dataset contiene más de un millón de archivos ejecutables, recolectados entre 2017 y 2018, etiquetados como malware o benignos. Incluye además un conjunto de muestras no etiquetadas para evaluación semisupervisada. EMBER proporciona una representación vectorizada de cada archivo, construida a partir de una combinación de **features estáticas extraídas del binario**, tales como:

- Hashes, tamaño de archivo, y entropía
- Importaciones y exportaciones de funciones
- Secciones PE y sus metadatos
- Strings, encabezados y recursos del archivo
- Estadísticas byte-level y opcode n-grams

Estas características están organizadas y almacenadas en formato JSON procesable, lo cual facilita su uso directo con algoritmos de machine learning. El dataset no incluye los binarios originales por razones de seguridad, pero ofrece suficientes vectores para permitir el entrenamiento robusto de modelos como Random Forest, XGBoost, LightGBM o redes neuronales densas.

Entre sus contribuciones más destacadas, EMBER permitió:

- Evaluar el rendimiento de modelos frente a un entorno realista y desbalanceado.
- Probar técnicas de *feature selection* e ingeniería de características sin necesidad de desensamblado o ejecución.

- Establecer una línea base reproducible y abierta para estudios posteriores.

Desde su publicación, múltiples trabajos han empleado EMBER como referencia para comparar nuevos enfoques, incluyendo arquitecturas profundas como MalConv, modelos interpretables, y defensas frente a ataques adversarios. Su estructura clara y su versatilidad lo convierten en un recurso clave para experimentos en detección estática de malware a gran escala.

2.3 Conclusiones

De la revisión se puede ver que tanto los modelos más clásicos como los basados en deep learning tienen su lugar en la detección de malware. Los primeros suelen ser más simples y transparentes, mientras que los segundos tienen mejor capacidad de generalización y aprovechan mejor grandes cantidades de datos. Aun así, ninguno por sí solo resuelve el problema del todo. Aquí es donde entran las técnicas de explicabilidad (XAI), que sirven de puente entre precisión y confianza porque permiten revisar cómo funcionan los modelos más complejos.

En los últimos años han salido bastantes trabajos usando SHAP o LIME, pero todavía hay muy pocos que comparen de forma sistemática algoritmos clásicos y redes profundas dentro del mismo marco de XAI, y menos aún enfocados en binarios PE. Esta falta de referencias es lo que justifica este proyecto: evaluar ambos enfoques no solo por su rendimiento, sino también por lo interpretables y útiles que pueden ser en la práctica.

En ciberseguridad no alcanza con detectar malware: también hay que poder explicar por qué se clasificó como tal. Esa explicación es clave para que los analistas puedan validarlo y para tener trazabilidad en auditorías. Por eso, combinar detección con XAI no es solo una idea interesante, sino cada vez más necesaria.

La elección del dataset *EMBER*, junto con la comparación entre modelos clásicos y redes neuronales usando SHAP y LIME, permite evaluar tanto el rendimiento puro como el nivel de comprensión que se puede lograr sobre las decisiones del sistema. Este enfoque va en la línea de las tendencias actuales en seguridad con IA y apunta a soluciones más sólidas, auditables y confiables.

3. Objetivos concretos y metodología de trabajo

En este capítulo se presentan tanto el objetivo principal como los objetivos más específicos del TFM, junto con la metodología pensada para cumplirlos. También se explican las fases de trabajo y las herramientas utilizadas, para que quede claro el camino seguido desde el diseño hasta la evaluación de los resultados.

3.1 Objetivo general

El objetivo principal de este Trabajo Fin de Máster es diseñar y comparar un sistema de detección de malware binario en modo estático, probando tanto algoritmos de machine learning más clásicos (como Random Forest o SVM) como modelos de deep learning (MLP, CNN o RNN). Para las pruebas se usará el dataset EMBER como referencia.

Además de medir el rendimiento, el proyecto incluye una parte centrada en la explicabilidad usando enfoques de Inteligencia Artificial Explicable (XAI). Para ello se aplicarán herramientas como SHAP y LIME, que permiten, de forma post-hoc, ver qué características han sido más relevantes en las predicciones de los modelos. La idea es que los resultados no se queden en una simple clasificación binaria, sino que también incluyan explicaciones claras que ayuden a entender el proceso de decisión.

Con esta base, el trabajo no solo busca comparar qué tan bien funcionan los enfoques clásicos frente a los profundos, sino también evaluar qué tan transparentes y útiles pueden ser en la práctica de la ciberseguridad. En definitiva, la meta es aportar una visión equilibrada que tenga en cuenta tanto la precisión como la interpretabilidad, de manera que los modelos resulten realmente útiles, auditables y confiables en escenarios reales.

3.2 Objetivos específicos

Para alcanzar el objetivo general, el trabajo se desglosa en los siguientes objetivos concretos:

1. **Revisión bibliográfica**: Analizar el estado del arte en detección de malware con aprendizaje automático y profundo, así como la evolución de las técnicas de expli-

cabilidad, prestando especial atención a las librerías SHAP y LIME, con el fin de identificar enfoques, tendencias y carencias de investigación.

2. **Preparación del dataset EMBER**: Descargar y comprender el conjunto de datos, realizar su limpieza y preprocesamiento. Este proceso incluye la vectorización de los binarios, la normalización de variables y el tratamiento del desbalance de clases, además de la división en subconjuntos de entrenamiento, validación y prueba.

3. **Entrenamiento de modelos clásicos**: Implementar algoritmos tradicionales de machine learning (como Regresión Logística, Random Forest, LightGBM, SVM, KNN y MLP de scikit-learn), ajustando sus hiperparámetros mediante validación cruzada para obtener un rendimiento óptimo.

4. **Entrenamiento de modelos de deep learning**: Diseñar y entrenar distintas arquitecturas neuronales en Keras (MLP, CNN 1D y LSTM), configurando parámetros como número de capas, neuronas, tasa de aprendizaje o número de épocas, y aplicando técnicas de regularización y early stopping para evitar sobreajuste.

5. **Evaluación comparativa de rendimiento**: Evaluar todos los modelos —clásicos y profundos— en el conjunto de prueba, utilizando métricas estándar (accuracy, precision, recall y F1-score), además de matrices de confusión para examinar falsos positivos y falsos negativos.

6. **Aplicación de técnicas de XAI**: Emplear las librerías SHAP y LIME sobre los modelos entrenados, comenzando por los algoritmos clásicos y posteriormente sobre las redes neuronales. En ambos casos, generar explicaciones tanto globales como locales para comprender qué variables influyen más en cada predicción.

7. **Análisis de interpretabilidad**: Contrastar los resultados de SHAP y LIME para identificar coincidencias y diferencias entre modelos clásicos y profundos respecto a las características más relevantes del malware. Evaluar la coherencia de estas explicaciones con el conocimiento de dominio (imports sospechosos, entropía, secciones anómalas, etc.), con el fin de determinar la utilidad práctica de cada enfoque.

8. **Síntesis y conclusiones**: Integrar los hallazgos en una comparación global que tenga en cuenta tanto precisión como interpretabilidad. Reflexionar sobre qué modelos resultan más adecuados en escenarios reales de ciberseguridad, considerando

limitaciones como coste computacional, robustez frente a ofuscación o necesidad de justificar cada alerta. A partir de ello, proponer recomendaciones sobre la aplicabilidad de los distintos enfoques en la práctica.

3.3 Metodología del trabajo

Para conseguir los objetivos planteados, se seguirá una metodología dividida en varias fases, apoyada en herramientas de software y frameworks de machine learning. A continuación, se explican las principales etapas del proyecto y los recursos que se usarán en cada una:

3.3.1 Fases del proyecto

- **Fase 1: Preprocesamiento de datos** – En esta parte se trabajará con el dataset EMBER (versión 2018) para dejarlo listo antes de entrenar los modelos. EMBER (Elastic Malware Benchmark for Empowering Researchers) contiene una gran colección de características estáticas extraídas de archivos PE de Windows, con alrededor de 1 millón de muestras ya etiquetadas como benignas o maliciosas. Cada archivo se representa como un vector con cientos de atributos (por ejemplo, conteos de bytes, entropía de secciones, importaciones, tamaño de cabecera, etc.). Estos datos se cargarán en Python con librerías como `pandas`. El preprocesamiento incluirá pasos como eliminar características que no aportan (ej.: hashes o nombres), manejar valores faltantes, normalizar variables numéricas y, si hace falta, codificar las variables categóricas. Además, se dividirá el dataset en conjuntos de entrenamiento, validación y prueba, manteniendo el balance entre clases. Dado que EMBER es muy grande, se podría recurrir a muestreo estratificado o reducción de dimensionalidad para que el entrenamiento sea manejable, sin perder información útil para modelos más complejos.

- **Fase 2: Entrenamiento de modelos clásicos de machine learning** – Con los datos ya procesados, se entrenarán algoritmos tradicionales de machine learning. En particular, se probarán modelos como Random Forest y SVM (Máquinas de Vectores de Soporte) usando la librería `scikit-learn` de Python. Se ajustarán los hiperparámetros y la elección final se basará en métricas de validación que equilibren falsos positivos y falsos negativos.

- **Fase 3: Entrenamiento de modelos de Deep Learning** – De forma paralela, se diseñarán y entrenarán redes neuronales profundas con frameworks como TensorFlow (Keras) o PyTorch. Se planteará un perceptrón multicapa (MLP) con capas densas y regularización, una red convolucional 1D (CNN) para capturar combinaciones locales de características en los vectores de EMBER, y una red recurrente (LSTM) que modele posibles dependencias entre grupos de características. Durante esta fase se aplicarán técnicas como *early stopping*, entrenamiento en lotes y uso de GPU para acelerar el proceso.

- **Fase 4: Evaluación y comparación de modelos** – Una vez entrenados, tanto los modelos clásicos como los de Deep Learning se evaluarán sobre el conjunto de prueba, que no habrá sido usado ni en entrenamiento ni en validación. Se calcularán métricas como *accuracy*, *precision*, *recall* y *F1-score*, además de matrices de confusión para ver en detalle el comportamiento de los clasificadores. La comparación tendrá en cuenta el rendimiento, el coste computacional (tiempo de entrenamiento e inferencia) y las limitaciones que se observen en cada caso.

- **Fase 5: Aplicación de técnicas de explicabilidad (XAI)** – Con los modelos ya evaluados, se aplicarán técnicas de explicabilidad post-hoc para interpretar sus predicciones. Se usarán librerías como SHAP y LIME, que permiten analizar la importancia de cada característica tanto a nivel global como en casos individuales. Se elegirán ejemplos representativos del conjunto de prueba para generar explicaciones y comparar cómo cambia la interpretación entre los modelos clásicos y las redes profundas.

- **Fase 6: Integración y análisis final** – En la última etapa se reunirán los resultados cuantitativos y cualitativos obtenidos, construyendo un análisis comparativo entre los modelos. A partir de ahí se extraerán conclusiones sobre qué enfoques resultan más adecuados y se propondrán posibles líneas de trabajo futuro.

3.3.2 Herramientas y frameworks

El proyecto se desarrollará principalmente en Python, aprovechando su ecosistema lleno de librerías para ciencia de datos y machine learning. En concreto, se usarán las siguientes herramientas y frameworks:

- **scikit-learn**: para entrenar modelos clásicos de ML (Random Forest, SVM, etc.) y para utilidades de preprocesamiento como escalado, división de datos o evaluación de métricas.
- **TensorFlow 2/Keras** (o en su lugar **PyTorch**): para diseñar, entrenar y evaluar las redes neuronales profundas (MLP, CNN, RNN). Estas librerías son flexibles, están optimizadas para datasets grandes como EMBER y permiten usar GPU para acelerar el entrenamiento.
- **SHAP library**: librería de Python que implementa los métodos SHAP (KernelSHAP, TreeSHAP, DeepSHAP) para explicar modelos. Se usará tanto con algoritmos clásicos (aprovechando su compatibilidad con árboles) como con redes neuronales (usando el explainer específico de DNN).
- **LIME (lime package)**: para generar explicaciones locales independientes del modelo. Servirá para contrastar y complementar las explicaciones de SHAP en ejemplos concretos, aportando otra visión (modelo lineal local frente a contribuciones de Shapley).
- **Pandas, NumPy**: para cargar, transformar y manipular los datos de EMBER, convirtiéndolos en matrices o dataframes fáciles de procesar.
- **Matplotlib/Seaborn**: para hacer gráficos cuando sea necesario, por ejemplo distribuciones de características en la exploración, o visualizaciones de explicaciones (barras con las features más relevantes según SHAP o las perturbaciones generadas por LIME).
- **Entorno de trabajo**: se necesitará un equipo con suficiente capacidad, ya que se trabajará con unas 1 millón de muestras y vectores de cientos de dimensiones. Lo ideal es contar con una GPU para acelerar el entrenamiento de modelos profundos. También se usará Jupyter Notebooks (o similar) para documentar los experimentos e integrar las visualizaciones de los resultados de XAI.

Estas herramientas se eligieron porque son estándar en la comunidad y tienen soporte activo. Por ejemplo, scikit-learn ofrece implementaciones optimizadas de Random Forest y SVM, mientras que TensorFlow/Keras facilita crear prototipos de redes rápidamente. SHAP y LIME son librerías de XAI muy usadas y bien documentadas en problemas de

clasificación tabular, lo que encaja perfectamente con el tipo de datos de EMBER (tablas con muchas columnas).

4. Desarrollo del trabajo

En este capítulo se detalla el proceso seguido para llevar a cabo el proyecto, desde la preparación del conjunto de datos hasta la implementación de los distintos modelos de aprendizaje automático y profundo. Se describe primero el dataset EMBER y los pasos de preprocesamiento aplicados, para después abordar el diseño y entrenamiento de los modelos clásicos y de redes neuronales. Finalmente, se presentan las herramientas y métricas utilizadas para la evaluación, preparando el terreno para el análisis de resultados en capítulos posteriores.

4.1 Descripción del dataset EMBER

El presente trabajo se basa en el uso del dataset **EMBER 2018** (Elastic Malware Benchmark for Empowering Researchers), un conjunto de datos de acceso público diseñado para entrenar modelos de aprendizaje automático capaces de detectar archivos PE (Portable Executable) maliciosos mediante análisis estático. Fue desarrollado por Endgame Inc. con el objetivo de proporcionar una base estándar para evaluar técnicas de detección de malware de forma reproducible y comparativa (Anderson y Roth, 2018).

Este dataset contiene información extraída de un total de **1 millón de archivos PE**, distribuidos en:

- 800,000 muestras para entrenamiento:
  - 300,000 maliciosas
  - 300,000 benignas
  - 200,000 sin etiquetar
- 200,000 muestras para test:
  - 100,000 maliciosas
  - 100,000 benignas

Cada muestra está representada como un objeto JSON que contiene metadatos y una colección de características derivadas del archivo ejecutable. Estas características han sido generadas mediante análisis estático, lo que significa que no requieren la ejecución del archivo para su extracción. Entre los atributos extraídos se encuentran:

- Información general del archivo (tamaño, presencia de firmas, número de strings, etc.).
- Encabezados PE: datos de la cabecera COFF y de la cabecera opcional (sistema operativo objetivo, versión del enlazador, DLL characteristics, etc.).
- Funciones importadas y exportadas: nombres de bibliotecas y funciones.
- Secciones del archivo PE: nombres, tamaños, entropía y propiedades de cada sección.
- Histogramas de bytes: distribución de los valores hexadecimales contenidos en el archivo.
- Histogramas de entropía de bytes: combinación de entropía local con los valores de bytes.
- Estadísticas sobre strings: número, longitud media, distribución de caracteres imprimibles, ocurrencias de URLs, rutas o claves de registro.

Este conjunto de características estáticas fue pensado para conservar la mayor cantidad posible de información útil sobre la estructura del archivo, de modo que los modelos de clasificación puedan aprender patrones que diferencien malware de archivos benignos. Además, el dataset incluye la **fecha estimada de aparición** de cada muestra (con resolución mensual) y su **hash SHA-256**, lo que hace posible rastrear los archivos originales en servicios como VirusTotal/Google Threat Intelligence o VirusShare.

Un punto clave del dataset EMBER es que no trae los binarios en crudo por razones legales y de seguridad, sino **características ya procesadas**. Esto lo convierte en un recurso seguro para investigación y facilita la reproducibilidad sin necesidad de manipular directamente malware real. Aparte, el proyecto pone a disposición código abierto para extraer nuevas muestras, lo que permite ampliar el dataset o crear conjuntos propios.

Las características extraídas se pueden dividir en dos grandes grupos:

1. **Características estructuradas y simbólicas**: recogen información textual o categórica (como nombres de DLLs, flags o secciones) que necesitan técnicas de codificación especializadas para vectorizarse (por ejemplo, feature hashing).
2. **Características numéricas y distribucionales**: incluyen histogramas, conteos, medidas de entropía y tamaños, que pueden usarse directamente como entradas numéricas.

Gracias a este dataset es posible entrenar tanto modelos clásicos como arquitecturas más complejas (por ejemplo, redes neuronales). También facilita el análisis de interpretabilidad a nivel de características, ya que cada muestra está descrita mediante vectores numéricos que vienen de atributos semánticos de los archivos.

En resumen, EMBER 2018 ofrece una base amplia y variada para investigar la detección de malware con machine learning, permitiendo estudiar tanto el rendimiento de los modelos como su capacidad de ser interpretados.

4.2 Preprocesamiento y vectorización

4.2.1 Vectorización

Un paso clave en el desarrollo de sistemas de detección de malware con aprendizaje automático es transformar los datos crudos en una representación numérica que los modelos puedan manejar. En este caso se parte del dataset EMBER 2018, que ofrece descripciones estáticas en formato JSONL de archivos PE. Sin embargo, esos datos vienen en una estructura jerárquica y textual que no sirve directamente para entrenar modelos de machine learning.

Para resolver este problema, se aplica un proceso de vectorización que convierte cada muestra en un vector numérico de longitud fija. Este procedimiento está implementado principalmente en la función `create_vectorized_features()` del módulo *ember* en Python, y se apoya en varias clases que extraen, procesan y codifican las características más importantes del binario. El código fuente de este proceso está disponible de forma pública en el repositorio oficial de EMBER (Anderson, 2018). A continuación, se explica cómo funciona y por qué se justifica su uso.

El pipeline de vectorización está basado en varios trabajos previos del área, y se organiza en módulos reutilizables que facilitan su aplicación:

- **ByteHistogram**: genera un histograma normalizado de los valores byte en el archivo.
- **ByteEntropyHistogram**: calcula la coocurrencia entre entropía local y valor byte.
- **StringExtractor**: analiza las cadenas imprimibles presentes, su entropía, distribución, y ocurrencias sospechosas (URLs, rutas, claves de registro).

- **GeneralFileInfo**: resume información general como tamaño, presencia de firmas o recursos.

- **HeaderFileInfo**: codifica metadatos del encabezado PE como plataforma, versión del enlazador, subsistema, etc.

- **SectionInfo**: resume secciones PE usando hashing de nombres, tamaños, entropía y características de la sección de entrada.

- **ImportsInfo**: resume las bibliotecas y funciones importadas mediante hashing.

- **ExportsInfo**: codifica las funciones exportadas como vectores.

- **DataDirectories**: captura tamaños y direcciones virtuales de estructuras clave de Windows (IAT, TLS, recursos, etc.).

Cada uno de estos componentes implementa dos métodos: `raw_features()` para extraer los datos crudos desde el binario utilizando `lief`, y `process_raw_features()` para convertir dicha información en vectores numéricos.

Ejemplo representativo: en la clase `ImportsInfo`, se recorre la tabla de importaciones del binario PE y se obtiene la lista de DLLs utilizadas y funciones invocadas. Estas cadenas se codifican usando `FeatureHasher` en vectores de 256 (DLLs) y 1024 (funciones) dimensiones, respectivamente. El resultado es un vector fijo que representa de forma aproximada el perfil de importación del archivo.

De la misma forma, `SectionInfo` utiliza hashing sobre los nombres de las secciones y de algunas propiedades como tamaño, entropía y permisos (lectura, ejecución, escritura), generando una codificación abstracta pero informativa del layout interno del binario.

El resultado de este pipeline es un vector de 2381 dimensiones por archivo. Todos los vectores se almacenan en formato `NumPy` para su uso eficiente posterior. El proceso de vectorización no considera atributos identificativos como el hash SHA256, ni información contextual como la fecha de aparición o etiquetas externas, ya que estos no aportan valor semántico interpretable para el modelo. A continuación, se detalla la correspondencia entre cada bloque de características y su rango de columnas en el vector final.

Bloque	Inicio	Fin	Descripción
`histogram`	0	255	Histograma de bytes (frecuencia de 0x00–0xFF)
`byteentropy`	256	511	Histograma 2D de bytes vs entropía
`strings`	512	615	Estadísticas de strings ASCII (longitud, entropía, URLs, etc.)
`general`	616	625	Información general del binario (tamaño, recursos, etc.)
`header`	626	687	Cabecera PE: timestamp, arquitectura, flags, etc.
`section`	688	942	Información de secciones PE (tamaños, entropías, características)
`imports`	943	2222	Imports: librerías y funciones importadas (hashing)
`exports`	2223	2350	Funciones exportadas
`datadirectories`	2351	2380	Entradas del directorio PE: RVA y tamaño de estructuras

Tabla 4.1: Desglose de bloques de características vectorizadas en el dataset EMBER

Este procedimiento estandariza la entrada para todos los modelos, independientemente de su naturaleza. Además, como cada componente captura aspectos estructurales y semánticos conocidos en la literatura como relevantes para la detección de malware, se asegura una base sólida tanto para clasificación como para interpretabilidad posterior (SHAP, LIME).

4.2.2 Filtrado, selección y normalización del conjunto de datos

Una vez realizada la vectorización completa de las muestras del dataset EMBER 2018, el siguiente paso consistió en depurar y preparar el conjunto de datos para su uso en modelos de clasificación. El archivo original contiene 1 millón de muestras distribuidas en tres clases distintas:

- **Etiqueta 0**: archivo benigno.
- **Etiqueta 1**: archivo malicioso (malware).
- **Etiqueta -1**: muestra no etiquetada, sin garantía de su naturaleza.

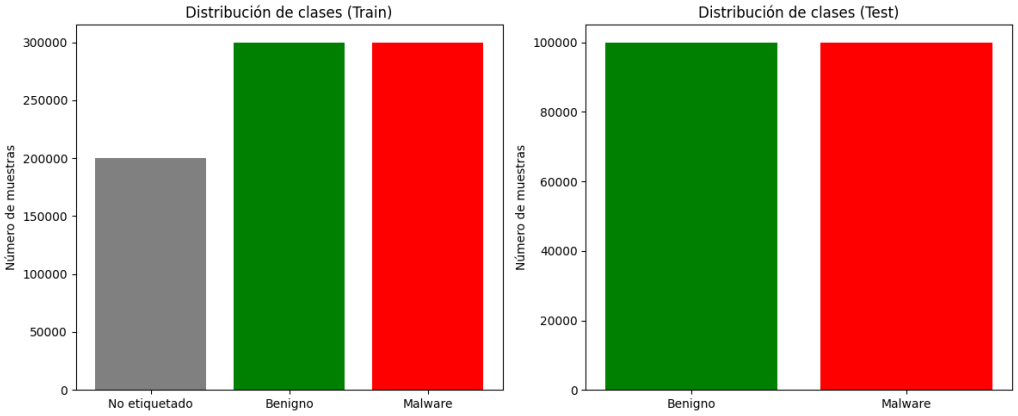


Figura 4.1: Distribución original de clases en el conjunto de datos EMBER. Las muestras no etiquetadas (etiqueta -1) suponen aproximadamente un 20 % del conjunto.

Estas últimas corresponden a binarios que no fueron verificados ni como malware ni como benignos, y por tanto **no son adecuados para el entrenamiento supervisado**. Como primer paso de depuración, se eliminan todas las entradas con etiqueta -1. Esta operación reduce el **conjunto útil de entrenamiento a 600,000 muestras etiquetadas**, distribuidas equitativamente entre malware (1) y benignos (0).

Previo al entrenamiento, se realizó una exploración inicial del conjunto vectorizado mediante la visualización de la primera muestra y el análisis estadístico básico de sus valores. En la Figura 4.2 se observa que ciertas características presentan valores extremadamente altos (hasta el orden de 10^9), mientras que el resto de dimensiones mantienen valores cercanos a cero. Esta desproporción puede afectar gravemente a modelos sensibles a la escala, como SVM o redes neuronales.

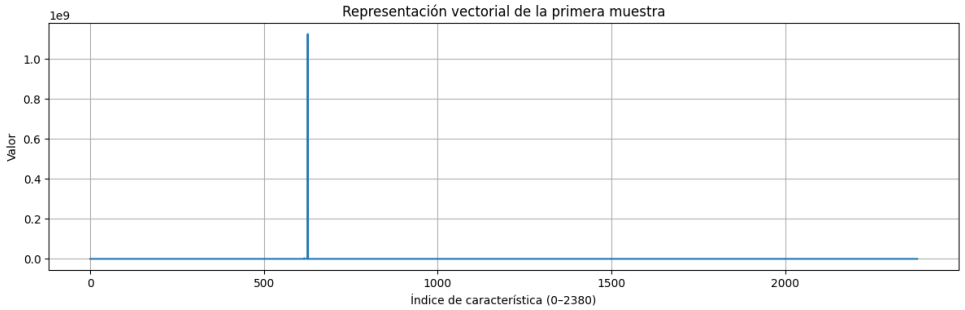


Figura 4.2: Representación vectorial sin normalizar de la primera muestra. Se observa una fuerte asimetría en las escalas de valores.

Este fenómeno se confirma en la inspección de los valores extremos, como se muestra

en la Tabla 4.2 y la Tabla 4.3. El valor máximo aparece en la posición 626 del vector, correspondiente a una característica numérica de la cabecera PE (`header`), y alcanza el 1.124.149.376, lo que justifica la necesidad urgente de aplicar un proceso de normalización.

Comando	**Salida**
`max_index = np.argmax(X_train[0])`	`626`
`print(f"Máximo en la posición: {max_index}")`	`Máximo en la posición: 626`
`print(f"Valor: {X_train[0][max_index]}")`	`Valor: 1124149376.0`

Tabla 4.2: Identificación de la característica con mayor valor absoluto en una muestra.

Columna	**Media**	**Mínimo**	**Máximo**
616	$2.65 \cdot 10^7$	0	$1.15 \cdot 10^8$
617	$5.24 \cdot 10^6$	0	$1.76 \cdot 10^7$
618	$1.09 \cdot 10^6$	0	$3.82 \cdot 10^8$
...	...	...	...
626	$1.09 \cdot 10^8$	0	$3.82 \cdot 10^8$

Tabla 4.3: Estadísticos descriptivos de las columnas con valores extremos (616–626), que confirman la necesidad de estandarización.

Por eso se aplica una **normalización por características**, que resulta clave en modelos como SVM o MLP. Para ello se usa el escalado estándar de scikit-learn (`StandardScaler`), que transforma cada característica para que tenga media 0 y desviación típica 1. Este paso es muy importante cuando las variables originales están en escalas diferentes (por ejemplo, el número de secciones frente a la entropía), ya que asegura una convergencia más estable en los algoritmos basados en gradiente y evita que ciertas variables con valores grandes dominen el espacio de decisión.

Después de aplicar esta transformación al subconjunto filtrado del dataset EMBER, se genera un nuevo conjunto de datos en la carpeta `filtered/scaled`, que contiene las muestras ya normalizadas. Este dataset incluye solo las observaciones etiquetadas como malware (1) o benignas (0), dejando fuera las de etiqueta -1 (desconocidas). La estandarización permite que, una vez eliminados los valores extremos (*outliers*) y escalados, las características queden distribuidas de forma más uniforme y comparable entre sí.

Esto también se nota en la representación gráfica de los vectores de entrada. En con-

creto, la Figura 4.3 muestra el perfil de características de la primera muestra tras la normalización, donde se aprecia un rango mucho más acotado y centrado en torno a la media, en contraste con la versión sin escalar que mostraba valores muy desproporcionados.

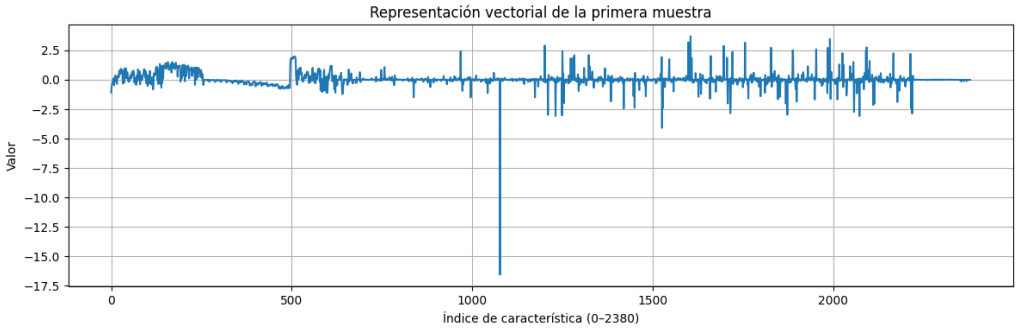


Figura 4.3: Representación vectorial de la primera muestra tras el escalado estándar.

Dado que este nuevo conjunto sigue siendo considerablemente grande (aproximadamente 600,000 muestras filtradas en el entrenamiento), se decide construir un subconjunto representativo de **50,000 muestras balanceadas**. Esta selección se realiza de forma aleatoria y estratificada, incluyendo 25,000 muestras de malware y 25,000 muestras benignas. Para mantener la coherencia experimental, se selecciona el mismo porcentaje de reducción en el conjunto de prueba: si se toma un 8.3 % del total del entrenamiento (50,000 de 600,000), se extrae aproximadamente un 8.3 % del conjunto de prueba filtrado, preservando también el equilibrio entre clases.

Esta decisión permite mantener una carga computacional razonable para múltiples experimentos, sin perder diversidad ni balance en los datos. Además, subconjuntos de este tamaño han demostrado ser suficientes en estudios previos del estado del arte para lograr una alta precisión, incluso sin utilizar el conjunto completo.

La Figura 4.4 ilustra la distribución de clases en ambos subconjuntos (entrenamiento y prueba), destacando el equilibrio logrado entre benignos y maliciosos tras el muestreo estratificado.

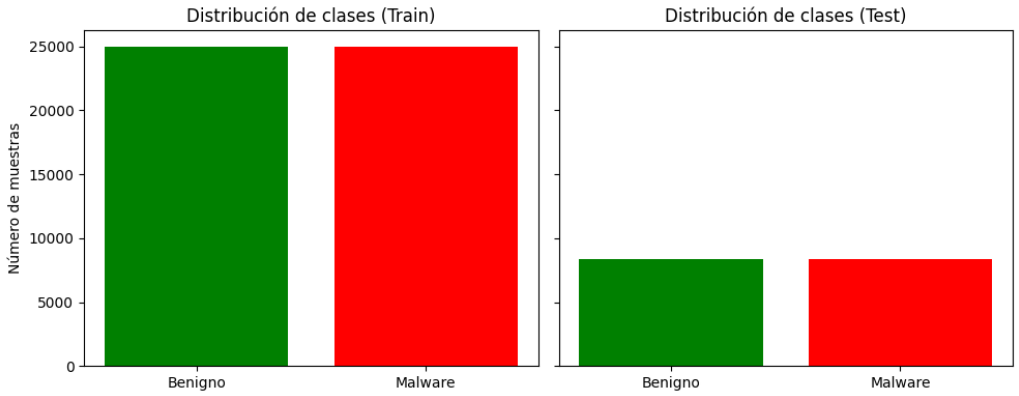


Figura 4.4: Distribución de clases en los subconjuntos de entrenamiento y prueba (50k y 16.6k muestras respectivamente).

En conclusión, este proceso de filtrado, reducción y escalado garantiza que los modelos se entrenen sobre un conjunto de datos confiable, balanceado y estandarizado, sin ruido ni desequilibrios que perjudiquen la evaluación de resultados o la posterior interpretación del comportamiento del sistema.

4.2.3 Exploración del dataset

En esta etapa se generaron distintas visualizaciones para observar cómo se distribuyen algunas características importantes antes de entrenar los modelos. Es importante señalar que estas gráficas se hicieron a partir de las características vectorizadas originales, sin aplicar todavía la normalización con `StandardScaler`. Esto se decidió para mantener la escala natural de las variables y que su interpretación fuera más clara (por ejemplo, en bytes, cantidades o frecuencias absolutas). Más adelante, estos valores sí se estandarizan antes de usarlos en los clasificadores.

Entropía en secciones PE. En la Figura 4.5 se muestra la distribución de la entropía en las secciones PE, calculada a partir del bloque `section`. La entropía mide qué tan aleatorio o desordenado es el contenido de una sección: valores bajos suelen indicar código sin ofuscar o estructuras simples, mientras que valores altos pueden apuntar a compresión, cifrado o empaquetado.

Para que la visualización sea más clara, se recortó el rango de entropía al percentil 1–99, dejando fuera los valores extremos poco frecuentes. La gráfica evidencia diferencias claras entre clases: los archivos maliciosos se concentran más en valores altos de entropía, con un segundo pico notable entre 4 y 8, mientras que los benignos tienden a estar en

valores bajos o medios, alrededor de 1–3.

Esta diferencia concuerda con el hecho de que muchos tipos de malware usan técnicas de evasión como *packers*, cifrado polimórfico o secciones inyectadas, que elevan la entropía. En cambio, los ejecutables legítimos suelen tener estructuras más regulares y predecibles, lo que hace que su entropía sea menor y menos variable.

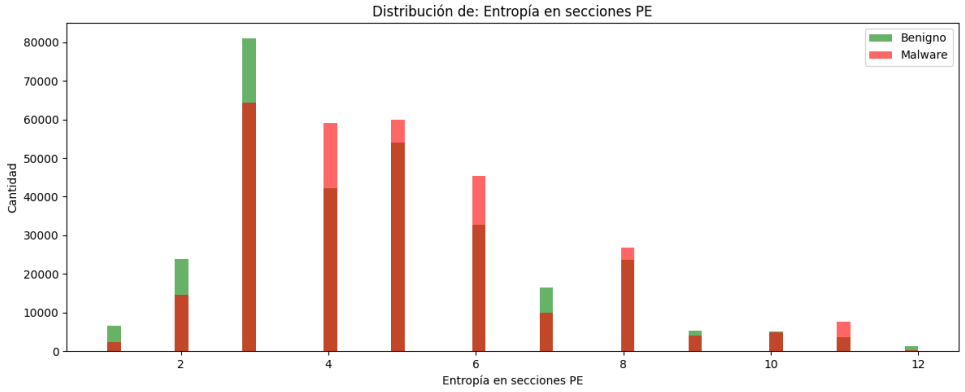


Figura 4.5: Distribución del valor de entropía en secciones PE ejecutables (rango centrado en percentiles 1–99).

Tamaño del archivo. En la Figura 4.6 se muestra la distribución del tamaño de los archivos PE, tomada del bloque `general`. Esta característica refleja el tamaño total del binario en bytes, incluyendo tanto el código ejecutable como recursos, datos o librerías incrustadas.

Para que la visualización sea más clara y no quede distorsionada por valores extremos, se limitó el rango a los percentiles 1 y 99. La gráfica deja ver que la mayoría de archivos, tanto benignos como maliciosos, están por debajo del megabyte, con una concentración marcada en tamaños pequeños (decenas o cientos de kilobytes).

Aun así, se nota que los archivos maliciosos presentan una distribución más amplia, con mayor dispersión hacia valores grandes. Esto tiene sentido si pensamos en el carácter modular de muchos tipos de malware (por ejemplo, *droppers*, *loaders*, *packers*), que suelen llevar varias capas, shellcodes o recursos embebidos. En cambio, muchos archivos legítimos tienden a ser más uniformes y optimizados, sobre todo los generados en entornos de desarrollo estándar.

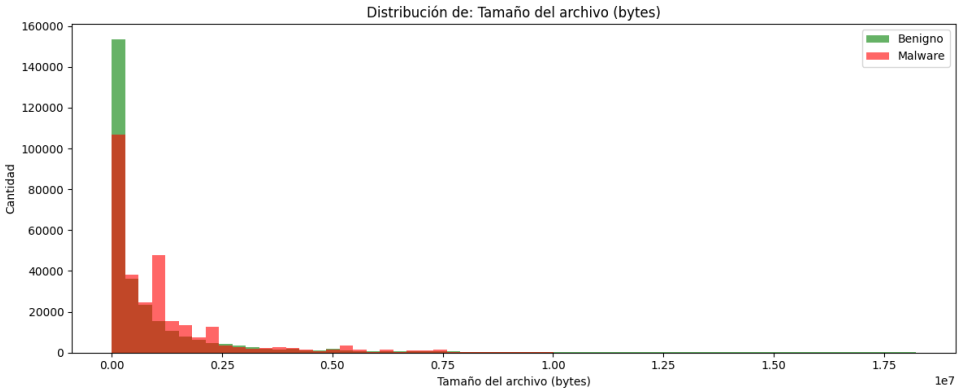


Figura 4.6: Distribución del tamaño del archivo PE (en bytes), recortada al rango de percentiles 1–99.

Número de cadenas ASCII. La Figura 4.7 muestra cómo se distribuye el número total de cadenas ASCII extraídas de cada binario, dato que se obtiene del bloque `strings`. Esta variable refleja la cantidad de secuencias imprimibles con al menos cinco caracteres consecutivos que aparecen dentro del ejecutable.

Como en otras visualizaciones, se limitó el rango a los percentiles 1 y 99 para que los valores extremos no distorsionen la escala. La distribución revela una concentración muy fuerte en los valores bajos: la mayoría de los archivos, tanto benignos como maliciosos, tienen menos de 10.000 cadenas. Sin embargo, se nota un patrón algo diferente entre las clases: los binarios maliciosos tienden a cubrir un rango un poco más amplio, con más presencia en niveles medios (entre 5.000 y 25.000), mientras que los benignos se agrupan más cerca de cero.

Esta diferencia puede explicarse porque muchos binarios maliciosos llevan mensajes internos, rutas de sistema, cadenas cifradas parcialmente legibles o fragmentos que quedan visibles tras el desempaquetado en memoria. En contraste, los ejecutables legítimos suelen contener cadenas más predecibles, relacionadas con interfaces gráficas, menús, recursos embebidos o librerías estándar.

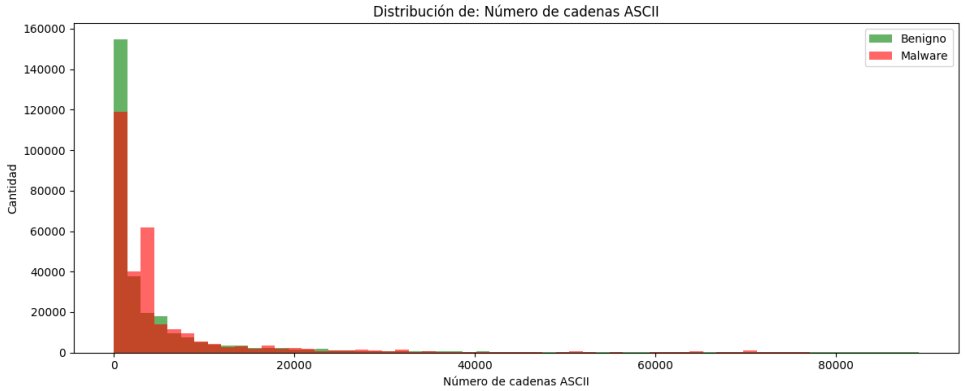


Figura 4.7: Distribución del número de cadenas ASCII extraídas por archivo, acotada al rango de percentiles 1–99.

Número de URLs detectadas. En la Figura 4.8 se muestra cómo se distribuye el número de URLs encontradas en cada binario, obtenidas del bloque `strings`. Esta variable cuenta los patrones que tienen formato de URL (por ejemplo, `http://`, `https://`) dentro de las cadenas imprimibles extraídas del ejecutable.

La distribución tiene un pico muy marcado en cero, lo que indica que la mayoría de los archivos (sobre todo los benignos) no contienen ninguna URL incrustada. Aun así, a medida que crece el número de URLs detectadas, la proporción de archivos maliciosos también aumenta de forma clara.

Esto encaja con el hecho de que muchos tipos de malware dependen de conectarse a servidores externos, ya sea para descargar payloads, recibir órdenes (C&C), robar información o actualizar componentes. En cambio, los programas legítimos rara vez incluyen URLs directamente en su código o recursos, salvo casos puntuales como instaladores o herramientas de red.

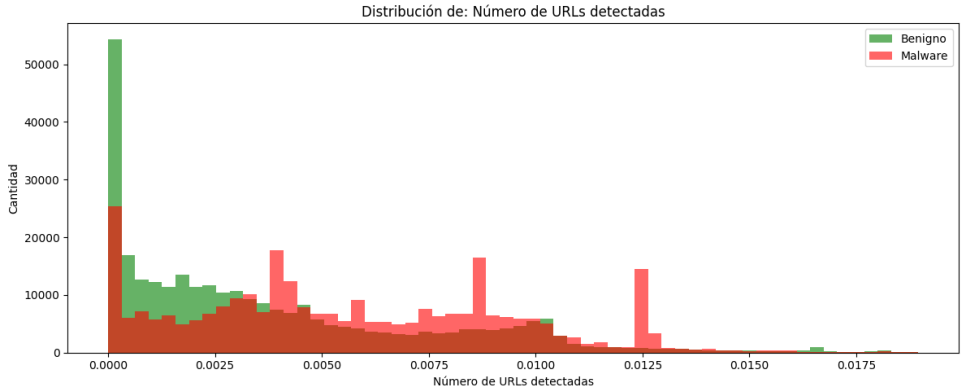


Figura 4.8: Distribución del número de URLs detectadas en cadenas ASCII, acotada al rango de percentiles 1–99.

Suma de funciones importadas. En la Figura 4.9 se muestra la distribución de una variable agregada: la suma total del vector correspondiente al bloque `imports`, que representa el uso estimado de funciones externas por parte de cada ejecutable.

Dado que esta representación se basa en hashing, es posible que aparezcan valores negativos o colisiones, lo que explica la presencia de valores por debajo de cero. Para facilitar su visualización, se ha aplicado escala logarítmica en el eje vertical.

En este caso, las distribuciones para archivos benignos y maliciosos presentan un alto grado de solapamiento, sin diferencias estructurales claras. Ambos tipos de archivos muestran un patrón disperso, con mayor densidad en torno a valores bajos y una caída progresiva hacia los extremos. Por tanto, esta característica, al menos de forma agregada, no parece especialmente discriminativa por sí sola, aunque podría aportar valor combinada con otras en un modelo multivariable.

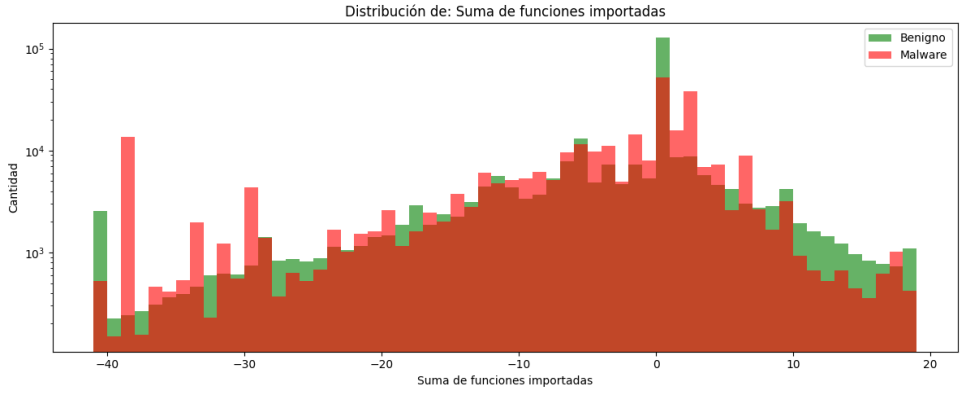


Figura 4.9: Distribución de la suma total del vector de funciones importadas (`imports`), recortada al rango de percentiles 1–99 y en escala logarítmica.

En resumen, el análisis exploratorio nos permitió detectar diferencias importantes en cómo se distribuyen algunas características entre archivos benignos y maliciosos. Variables como la entropía en las secciones PE o la presencia de URLs incrustadas muestran patrones bastante claros y prometen ser buenos indicadores de actividad maliciosa. Otras, como el número de cadenas ASCII o el tamaño del archivo, también aportan cierta capacidad de discriminación, aunque con bastante solapamiento entre clases. Por último, variables más generales como la suma de funciones importadas no parecen reflejar diferencias estructurales relevantes.

La Tabla 4.4 resume la utilidad observada de cada una de las características exploradas, sirviendo como referencia para decisiones posteriores sobre selección de variables y construcción de modelos de clasificación.

Variable	**Utilidad**	**Comentarios**
Entropía en secciones PE	Alta	El malware presenta entropías más elevadas, indicativas de cifrado u ofuscación.
Tamaño del archivo (bytes)	Moderada	Mayor concentración en tamaños pequeños, pero con dispersión más amplia en malware.
Número de cadenas ASCII	Moderada	El malware tiende a mostrar más cadenas en rangos medios, aunque con solapamiento.
Número de URLs detectadas	Alta	Fuerte correlación con comportamiento malicioso: más URLs incrustadas.
Suma de funciones importadas	Baja	Distribuciones similares entre clases; poca capacidad discriminativa observada.

Tabla 4.4: Resumen de utilidad observada para las variables exploradas en la fase de análisis exploratorio.

4.3 Métricas de evaluación

La evaluación del rendimiento de un modelo de clasificación es una etapa fundamental en cualquier proceso de aprendizaje supervisado. En el contexto específico de la detección de malware, esta evaluación adquiere especial importancia debido al impacto potencial de los errores de predicción: un falso negativo puede permitir que una amenaza pase desapercibida, mientras que un falso positivo puede bloquear injustificadamente archivos legítimos.

Con el objetivo de ofrecer una valoración integral del comportamiento de los modelos entrenados, se han seleccionado varias métricas complementarias, tanto a nivel general como detallado por clase. A continuación, se describen las métricas utilizadas y su significado práctico.

Matriz de confusión

La **matriz de confusión** resume las predicciones de un modelo frente a las etiquetas verdaderas. En un problema de clasificación binaria, como es el caso del presente estudio, la matriz adopta la siguiente estructura:

	Predicho: Malware (1)	**Predicho: Benigno (0)**
Real: Malware (1)	Verdaderos positivos (TP)	Falsos negativos (FN)
Real: Benigno (0)	Falsos positivos (FP)	Verdaderos negativos (TN)

Tabla 4.5: Estructura de una matriz de confusión binaria.

Cada una de estas cuatro cantidades se interpreta de la siguiente forma:

- **TP (True Positives)**: cantidad de archivos maliciosos correctamente identificados como malware.
- **TN (True Negatives)**: cantidad de archivos benignos correctamente clasificados como tales.
- **FP (False Positives)**: archivos benignos erróneamente marcados como malware.
- **FN (False Negatives)**: archivos maliciosos no detectados, clasificados como benignos.

Estas variables permiten construir métricas más avanzadas que reflejan aspectos clave del rendimiento del modelo.

Exactitud o precisión global (Accuracy)

La métrica más simple y general es la **exactitud** (*accuracy*), que mide el porcentaje de predicciones correctas sobre el total de muestras:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Aunque es útil como referencia global, esta métrica puede ser engañosa en problemas desbalanceados, ya que no distingue entre errores por clase.

Precisión y sensibilidad (Precision y Recall)

Dos métricas clave en tareas de detección binaria son la **precisión** y la **sensibilidad** (también conocida como *recall* o *true positive rate*):

$$\text{Precisión (Precision)} = \frac{TP}{TP + FP}$$
$$\text{Sensibilidad (Recall)} = \frac{TP}{TP + FN}$$

- La precisión indica qué proporción de las muestras clasificadas como malware lo eran realmente. Una baja precisión implica muchos falsos positivos.
- La sensibilidad mide la capacidad del modelo para detectar efectivamente archivos maliciosos. Una baja sensibilidad indica que muchos malware no están siendo detectados (alto número de falsos negativos).

En seguridad, la sensibilidad suele considerarse prioritaria, ya que un falso negativo puede dejar pasar una amenaza.

F1-Score

El **F1-score** es la media armónica entre precisión y sensibilidad. Proporciona un valor único que balancea ambos aspectos, penalizando más los modelos que tienen desequilibrios extremos entre ambas métricas:

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Este valor es especialmente útil cuando se quiere mantener un compromiso entre detección efectiva y bajo número de falsos positivos.

Área bajo la curva ROC (AUC)

Otra métrica fundamental es el **AUC-ROC** (Area Under the Curve - Receiver Operating Characteristic), que mide la capacidad del modelo para discriminar entre clases en todos los umbrales posibles. Esta métrica es independiente del umbral de decisión y se define como:

$$\text{AUC} = \int_0^1 \text{TPR}(FPR)\, d(\text{FPR})$$

donde:

$$\text{TPR} = \frac{TP}{TP + FN} \quad \text{y} \quad \text{FPR} = \frac{FP}{FP + TN}$$

El AUC puede interpretarse como la probabilidad de que el modelo asigne una puntuación mayor a una muestra positiva (malware) que a una negativa (benigno). Un AUC de 0.5 indica un modelo aleatorio, mientras que valores cercanos a 1.0 reflejan una alta capacidad discriminativa.

Consideraciones específicas en detección de malware

En el contexto de este trabajo, se hace especial énfasis en las métricas **F1-score** y **AUC**, ya que ofrecen una visión más completa del rendimiento cuando hay interés tanto en la capacidad de detección como en el control de falsos positivos. Aunque el conjunto de datos utilizado está balanceado (25 000 muestras de cada clase), en aplicaciones reales es habitual encontrar una fuerte asimetría, por lo que resulta necesario validar los modelos en métricas robustas ante desequilibrios.

Por este motivo, todos los modelos se evaluarán de forma sistemática usando las siguientes salidas:

- **Accuracy**, F1-score y AUC globales.
- **Matriz de confusión** completa.
- **Curvas ROC** por modelo.
- **Tablas comparativas** entre modelos.

Estos resultados se ampliarán en el capítulo siguiente, donde se presentan los modelos entrenados y su evaluación.

5. Resultados experimentales

Este capítulo recoge los resultados obtenidos tras aplicar distintos modelos de aprendizaje supervisado sobre el conjunto de datos previamente vectorizado y preprocesado. El objetivo principal es comparar el rendimiento de cada modelo desde un punto de vista cuantitativo, utilizando métricas como precisión, recall, F1-score y AUC.

El análisis se plantea de manera progresiva y reproducible: todas las muestras comparten una representación numérica común, sobre la que se entrenan diferentes algoritmos de clasificación. A partir de ahí, se evalúa su desempeño mediante métricas estándar, matrices de confusión y curvas ROC, lo que permite comparar de forma directa la eficacia de cada enfoque.

De este modo, el capítulo se centra en identificar qué modelos alcanzan un mayor rendimiento bruto y en discutir las diferencias más relevantes entre algoritmos clásicos y arquitecturas profundas.

A continuación se presentan los resultados de cada uno de los clasificadores considerados.

5.1 Modelos clásicos

En esta sección se presentan los resultados obtenidos con diferentes algoritmos clásicos de aprendizaje supervisado aplicados al problema de detección de malware. Se probaron enfoques bastante variados: desde modelos lineales como la Regresión Logística, pasando por métodos basados en distancias (K-Nearest Neighbors), clasificadores de margen máximo (SVM con núcleo RBF), hasta algoritmos de árboles de decisión, tanto en su versión con *bagging* (Random Forest) como con *boosting* (LightGBM). A este grupo se añadió además un perceptrón multicapa sencillo, que en la literatura suele verse como un punto intermedio entre los modelos clásicos y las redes neuronales profundas.

Todos los modelos se entrenaron con un mismo conjunto de 50,000 muestras balanceadas, y se evaluaron en un conjunto de prueba independiente con 16,666 instancias también equilibradas entre benignos y maliciosos. Para los algoritmos más sensibles a la escala de las variables (como la Regresión Logística, KNN, SVM y MLP) se aplicó una estandarización de características con `StandardScaler`. Para mantener la comparación consistente, esta transformación también se usó en los modelos basados en árboles, aunque en realidad

no la necesitan estrictamente.

El objetivo de este bloque experimental es doble: por un lado, tener una referencia clara para contrastar con los resultados de las redes profundas que se analizarán después; y por otro, revisar qué tipo de errores comete cada algoritmo en el conjunto de prueba y cómo varían sus patrones de clasificación.

5.1.1 Regresión Logística

La regresión logística es un modelo lineal muy usado en problemas de clasificación binaria. Sus principales ventajas son que se entrena rápido, apenas requiere recursos y resulta fácil de interpretar gracias a los coeficientes que genera. Esto último es especialmente útil en ciberseguridad, ya que aporta un plus de transparencia. Aunque no es capaz de capturar relaciones no lineales complejas, suele servir como buen punto de partida o *baseline* frente a métodos más avanzados (Pedregosa y cols., 2011).

En este trabajo, el modelo se entrenó con 50,000 muestras balanceadas (25,000 benignas y 25,000 maliciosas), aplicando antes una estandarización de variables para asegurar que el algoritmo convergiera correctamente y que las características fueran comparables entre sí. Además, se fijó el número máximo de iteraciones en 1000 para evitar los problemas de convergencia que aparecían con valores menores.

Ajuste de hiperparámetros

Se evaluaron distintas configuraciones de regularización y optimización. Concretamente, se exploraron valores de la constante C entre 0.01 y 100, penalizaciones L1 y L2, y los solvers `lbfgs` y `saga`. La Tabla 5.1 resume los valores probados y la configuración seleccionada.

Parámetro	**Opciones probadas**	**Opción seleccionada**
C	0.01, 0.1, 1, 10, 100	1
Penalización	l1, l2	l2
Solver	lbfgs, saga	lbfgs
Max_iter	100, 500, 1000	1000

Tabla 5.1: Parámetros evaluados y mejor configuración seleccionada para la regresión logística.

La mejor configuración se alcanzó con $C = 1$, penalización L2 y solver `lbfgs`. Esta

combinación logró un equilibrio adecuado entre sesgo y varianza, sin que incrementos adicionales de C o cambios de penalización aportaran mejoras sustanciales.

Resultados

El rendimiento obtenido se muestra en la Tabla 5.2. En el conjunto de test de 16.6k instancias, la exactitud fue del 73 %, el F1 macro de 0.73 y el AUC de 0.80. Aunque estas cifras corresponden al primer modelo evaluado, se verá que están en la misma línea que las alcanzadas por el resto de algoritmos, reflejando la dificultad del problema más que una limitación específica del modelo.

Métrica	**Test Accuracy**	**Test F1 macro**	**AUC**
Regresión Logística	0.73	0.73	0.80

Tabla 5.2: Rendimiento de la regresión logística en el conjunto de test (16.6k muestras).

La matriz de confusión de la Figura 5.1 muestra que el modelo tiende a marcar con más frecuencia instancias como *malware*, lo que reduce los falsos negativos pero incrementa los falsos positivos (3,130). En la clase *malware* el recall alcanza 0.83, mientras que en *benigno* el valor baja a 0.62, como se resume en la Tabla 5.3.

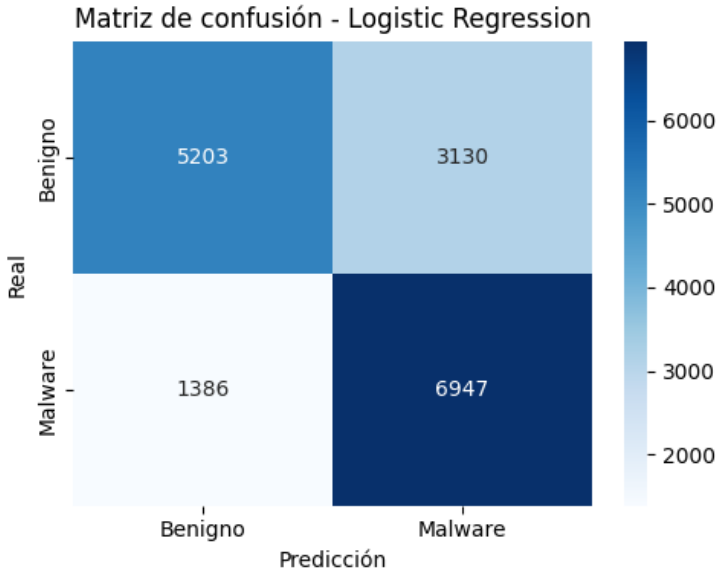


Figura 5.1: Matriz de confusión de la regresión logística en el conjunto de test (16.6k muestras balanceadas).

Clase	**Precisión**	**Recall**	**F1-score**
0 (Benigno)	0.79	0.62	0.70
1 (Malware)	0.69	0.83	0.75
Macro avg (Promedio macro)	0.74	0.73	0.73
Weighted avg (Promedio ponderado)	0.74	0.73	0.73

Tabla 5.3: Informe de clasificación de la regresión logística en el conjunto de test (16.6k muestras). Las clases se codifican como 0 = Benigno y 1 = Malware.

Valoración general

En general, la regresión logística consigue un rendimiento bastante competitivo y cercano al de otros algoritmos probados en este trabajo, lo que la convierte en una buena referencia a pesar de su sencillez. Su comportamiento refleja un compromiso clásico en ciberseguridad: priorizar la detección de la mayor parte del malware (menos falsos negativos) a costa de producir más falsos positivos. Más adelante veremos que los modelos no lineales pueden equilibrar mejor esta relación, pero los resultados confirman que incluso un clasificador lineal tan básico puede llegar a un nivel de rendimiento muy parecido al de técnicas más complejas.

5.1.2 Random Forest

El modelo de *Random Forest* se construye a partir de un conjunto de árboles de decisión entrenados sobre diferentes particiones y subconjuntos de características, y luego combina sus salidas mediante votación mayoritaria. De esta forma se reduce el sobreajuste típico de un solo árbol y se gana robustez frente al ruido o la variabilidad de los datos. Gracias a su capacidad para capturar relaciones no lineales y lidiar con atributos poco relevantes, es una técnica muy utilizada en problemas complejos como la detección de malware estático.

En este trabajo se usó la implementación de `scikit-learn` con 400 árboles, sin fijar un límite de profundidad y seleccionando atributos con el criterio `sqrt`, sobre un conjunto balanceado de 50 000 muestras. Aunque este modelo no necesita escalado de variables, aquí también se aplicó estandarización para mantener consistencia con el resto de algoritmos probados.

Ajuste de hiperparámetros

Durante el ajuste se exploraron diferentes configuraciones de número de árboles, profundidad máxima y criterio de selección de atributos. La Tabla 5.4 resume el espacio de búsqueda y la mejor combinación identificada.

Parámetro	**Opciones probadas**	**Opción seleccionada**
n_estimators	100, 400	400
max_depth	None, 30, 60	None
max_features	sqrt	sqrt

Tabla 5.4: Parámetros evaluados y mejor configuración seleccionada para Random Forest.

El incremento en el número de árboles por encima de 400 no supuso mejoras relevantes, mientras que limitar la profundidad reducía ligeramente el recall en la clase de malware. La opción `sqrt` para la selección de atributos ofreció un mejor equilibrio entre precisión y recall en el conjunto de prueba.

Resultados

El rendimiento alcanzado se resume en la Tabla 5.5. En el conjunto de test de 16.6k instancias, el modelo obtuvo una exactitud global del 82 %, un F1 macro de 0.82 y un AUC de 0.89. Estas cifras lo sitúan en la misma línea que otros algoritmos evaluados en este trabajo, sin diferencias sustanciales que permitan señalar un claro ganador.

Métrica	**Test Accuracy**	**Test F1 macro**	**AUC**
Random Forest	0.82	0.82	0.89

Tabla 5.5: Rendimiento de Random Forest en el conjunto de test (16.6k muestras).

La matriz de confusión mostrada en la Figura 5.2 permite observar con más detalle el patrón de errores. El modelo clasifica correctamente la mayoría de instancias, aunque mantiene un número apreciable de falsos positivos (2 233) y muy bajo de falsos negativos (781). Este comportamiento, similar al de otros algoritmos, refuerza la idea de que las métricas cuantitativas por sí solas no bastan para decidir qué modelo resulta más adecuado. Con todos estos resultados se realiza un informe de clasificación que se recoge en la Tabla 5.6, donde se observan valores más altos de recall para la clase *malware* frente a *benigno*, confirmando el sesgo del modelo hacia la detección de instancias maliciosas.

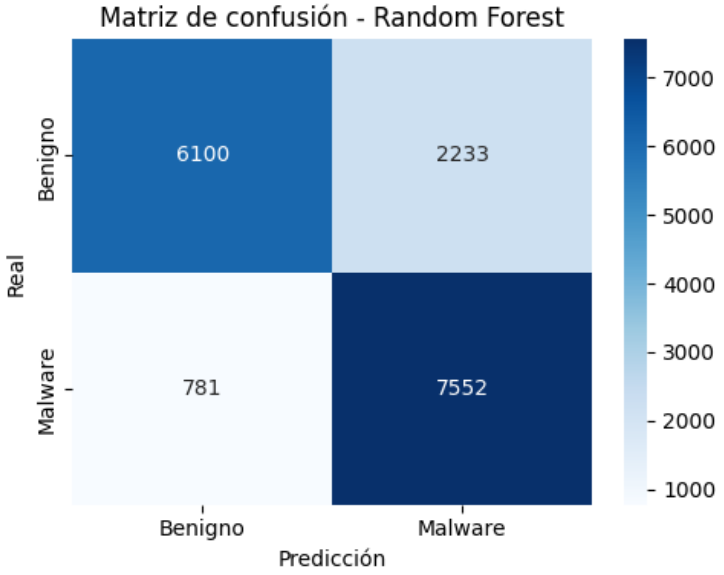


Figura 5.2: Matriz de confusión de Random Forest en el conjunto de test (16.6k muestras balanceadas).

Clase	**Precisión**	**Recall**	**F1-score**
0 (Benigno)	0.89	0.73	0.80
1 (Malware)	0.77	0.91	0.83
Macro avg (Promedio macro)	0.83	0.82	0.82
Weighted avg (Promedio ponderado)	0.83	0.82	0.82

Tabla 5.6: Informe de clasificación de Random Forest en el conjunto de test (16.6k muestras).

Valoración general

En conjunto, Random Forest se confirma como un modelo sólido y competitivo, capaz de manejar datos de alta dimensionalidad y con un rendimiento comparable al de otros algoritmos probados. Su tendencia a generar más falsos positivos puede ser un inconveniente en entornos de despliegue real, donde un exceso de alertas sobre archivos legítimos puede saturar a los analistas. Aun así, constituye una referencia importante en el marco experimental, especialmente porque permite integrar medidas de importancia de características que se explorarán en el Capítulo 6.

5.1.3 LightGBM

El modelo LightGBM (*Light Gradient Boosting Machine*) es un algoritmo de *boosting* que usa árboles de decisión entrenados de forma secuencial, donde cada nuevo árbol busca corregir los fallos de los anteriores. Su implementación está pensada para ser rápida y con bajo consumo de memoria, lo que lo hace muy adecuado en datasets grandes y con muchas variables como EMBER. A diferencia de Random Forest, que entrena árboles en paralelo, LightGBM va añadiendo árboles de manera iterativa para ir mejorando el ajuste.

En este trabajo se utilizó la clase `LGBMClassifier` con 100 árboles y una semilla fija (`random_state=42`), entrenado sobre un conjunto de 50 000 muestras balanceadas. Aunque los modelos de árboles no necesitan escalado de variables, aquí también se aplicó estandarización para mantener la coherencia con el resto de algoritmos.

Ajuste de hiperparámetros

Se probaron configuraciones básicas en torno al número de hojas, la profundidad máxima y la tasa de aprendizaje, manteniendo el resto de parámetros en sus valores por defecto. La Tabla 5.7 recoge las opciones analizadas y la configuración final empleada. Como puede observarse, la mejor combinación no difiere en exceso de los valores por defecto de LightGBM.

Parámetro	**Opciones probadas**	**Opción seleccionada**
num_leaves	31, 63	31
max_depth	-1, 20, 40	-1
learning_rate	0.03, 0.06, 0.1	0.1

Tabla 5.7: Parámetros evaluados y configuración seleccionada para LightGBM.

Resultados

Los resultados cuantitativos se presentan en la Tabla 5.8. En el conjunto de test de 16.6k instancias, LightGBM alcanzó una exactitud del 74 %, un F1 macro de 0.73 y un AUC de 0.85. Aunque estas cifras son comparables a las de otros modelos clásicos, el análisis detallado de errores revela un comportamiento más desequilibrado.

Métrica	Test Accuracy	Test F1 macro	AUC
LightGBM	0.74	0.73	0.85

Tabla 5.8: Rendimiento de LightGBM en el conjunto de test (16.6k muestras).

La matriz de confusión de la Figura 5.3 muestra que el modelo clasifica con éxito la mayoría del malware (recall de 0.95), pero lo hace a costa de marcar como maliciosas muchas instancias benignas. En total se registran 3 901 falsos positivos, una cifra notablemente superior a la observada en otros modelos. Este desequilibrio entre clases se refleja en el informe de la Tabla 5.9, donde la clase *benigno* obtiene un recall muy bajo (0.53).

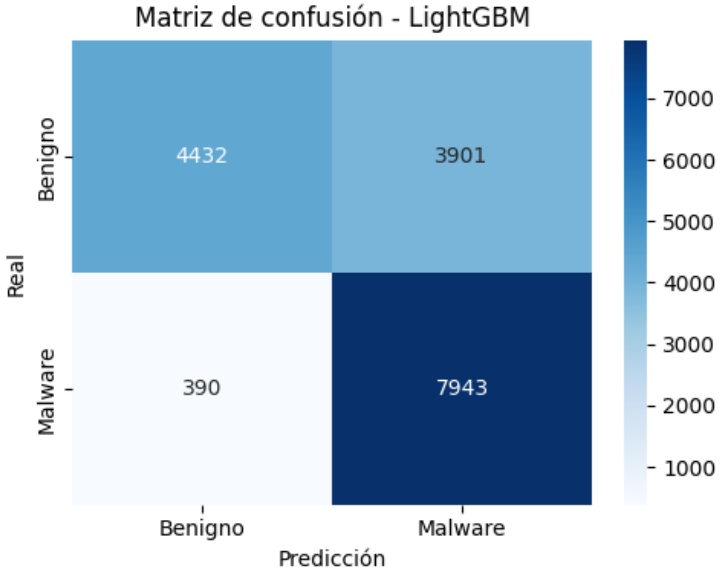


Figura 5.3: Matriz de confusión de LightGBM en el conjunto de test (16.6k muestras balanceadas).

Clase	Precisión	Recall	F1-score
0 (Benigno)	0.92	0.53	0.67
1 (Malware)	0.67	0.95	0.79
Macro avg (Promedio macro)	0.79	0.74	0.73
Weighted avg (Promedio ponderado)	0.79	0.74	0.73

Tabla 5.9: Informe de clasificación de LightGBM en el conjunto de test (16.6k muestras).

Valoración general

En general, LightGBM alcanza métricas globales parecidas a las de otros modelos clásicos, pero su patrón de errores resulta menos equilibrado. Su alta sensibilidad hacia la clase de malware hace que se reduzcan mucho los falsos negativos, pero al mismo tiempo genera una cantidad muy elevada de falsos positivos (casi 4 000 en el conjunto de prueba). En un escenario real, esto podría acabar provocando una sobrecarga de alertas falsas para los analistas, lo que limitaría bastante la utilidad práctica del modelo. Este caso muestra que no basta con mirar solo las métricas globales de rendimiento: también es clave estudiar cómo se reparten los errores y, más adelante, apoyarse en técnicas de explicabilidad (Capítulo 6) para entender qué factores están detrás de ese desequilibrio.

5.1.4 SVM (RBF)

El modelo **Support Vector Machine** con núcleo *Radial Basis Function* (RBF) es un clasificador robusto y bastante usado en tareas de clasificación binaria. Su principal ventaja es que puede manejar datos que no son linealmente separables, lo que lo hace útil para capturar patrones complejos como los que aparecen en el análisis estático de malware. Como este algoritmo es sensible a la escala de las variables, en este trabajo se aplicó después de estandarizar el conjunto vectorizado.

Para el experimento se utilizó la clase `SVC` de `scikit-learn` con `kernel=rbf"` y la opción `probability=True`, lo que permite calcular probabilidades y métricas como el área bajo la curva ROC (AUC). Los hiperparámetros principales (C y γ) se dejaron en sus valores por defecto de la librería, ya que ofrecen un ajuste razonable en este escenario.

Resultados

El rendimiento en el conjunto de prueba se resume en la Tabla 5.10. El modelo alcanzó una **exactitud del 78 %**, con un **F1 macro de 0.77** y un **AUC de 0.87**, lo que indica una capacidad de discriminación similar a la de otros modelos evaluados. La clase *malware* se ve favorecida con un recall alto (0.89), mientras que la clase *benigno* muestra un desempeño más bajo, con un recall de solo 0.66.

Métrica	Test Accuracy	Test F1 macro	AUC
SVM (RBF)	0.78	0.77	0.87

Tabla 5.10: Rendimiento de SVM (RBF) en el conjunto de test (16.6k muestras).

La matriz de confusión (Figura 5.4) muestra con detalle los aciertos y errores. El modelo clasifica correctamente 5 532 instancias benignas y 7 394 maliciosas, pero incurre en 2 801 falsos positivos y 939 falsos negativos. Este patrón refleja una tendencia a favorecer la detección de malware, reduciendo los falsos negativos pero a costa de incrementar notablemente las falsas alarmas.

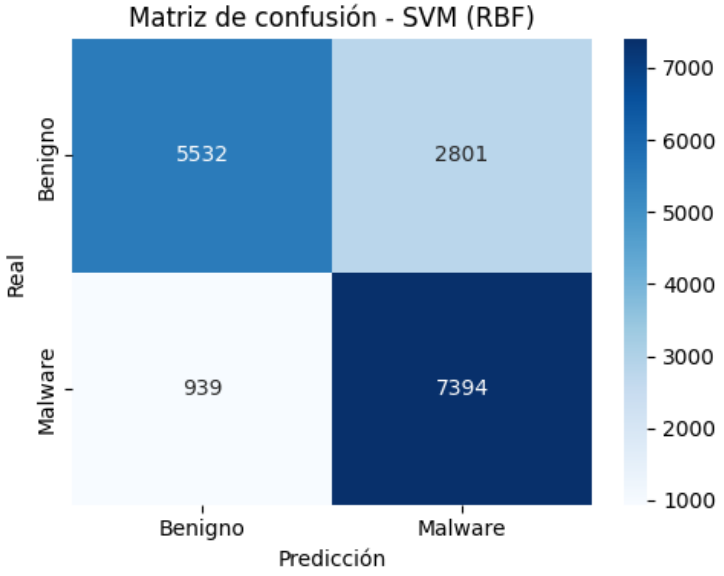


Figura 5.4: Matriz de confusión de SVM (RBF) en el conjunto de test (16.6k muestras balanceadas).

Clase	Precisión	Recall	F1-score
0 (Benigno)	0.85	0.66	0.75
1 (Malware)	0.73	0.89	0.80
Macro avg (Promedio macro)	0.79	0.78	0.77
Weighted avg (Promedio ponderado)	0.79	0.78	0.77

Tabla 5.11: Informe de clasificación de SVM (RBF) en el conjunto de test (16.6k muestras).

Valoración general

El clasificador SVM con núcleo RBF logra un rendimiento competitivo y comparable al de otros algoritmos probados, destacando sobre todo por su alta sensibilidad al detectar malware (recall del 89 %). Esto lo convierte en una opción interesante en escenarios donde los falsos negativos resultan especialmente problemáticos. Sin embargo, este beneficio viene acompañado de un número elevado de falsos positivos, lo que en un entorno real podría generar una carga excesiva de alertas para los analistas. A ello se suma el coste computacional considerable de este tipo de modelos, que puede dificultar su uso a gran escala en entornos de producción con muchos datos. En resumen, la SVM es una alternativa sólida, aunque con compromisos claros entre sensibilidad, practicidad y eficiencia.

5.1.5 K-Nearest Neighbors (KNN)

El modelo **K-Nearest Neighbors** es un algoritmo de clasificación que se basa en medir la similitud entre muestras. Para cada instancia nueva, asigna una clase según la mayoría de sus k vecinos más cercanos en el espacio vectorial. Su simplicidad lo convierte en un referente clásico dentro de los métodos supervisados, aunque su rendimiento depende bastante de la elección de k, de la métrica de distancia y del preprocesamiento de los datos. En este trabajo se aplicó una estandarización previa para que todas las características tuvieran un peso comparable al calcular las distancias.

Ajuste de hiperparámetros

Se probaron distintos valores de k en el rango $\{3, 5, 7, 9, 11, 15, 19\}$ junto con dos métricas de distancia (euclídea y manhattan). La configuración seleccionada fue $k = 5$ con distancia euclídea, ya que ofreció el mejor equilibrio entre precisión y recall. La Tabla 5.12 recoge el espacio de búsqueda y la opción final elegida.

Parámetro	**Opciones probadas**	**Opción seleccionada**
k	3, 5, 7, 9, 11, 15, 19	5
Métrica de distancia	Euclídea (p=2), Manhattan(p=1)	Euclídea

Tabla 5.12: Parámetros evaluados y mejor configuración seleccionada para KNN.

Resultados

Los resultados en el conjunto de test (Tabla 5.13) muestran que KNN alcanzó una **exactitud del 72 %** y un **F1 macro de 0.72**, con un AUC de 0.81. La clase *malware* presenta un recall superior (0.80) frente a la clase *benigno* (0.65), lo que refleja una tendencia a clasificar con más facilidad como maliciosos.

Métrica	**Test Accuracy**	**Test F1 macro**	**AUC**
KNN	0.72	0.72	0.81

Tabla 5.13: Rendimiento de KNN en el conjunto de test (16.6k muestras).

La matriz de confusión (Figura 5.5) muestra cómo el modelo clasifica correctamente 5 432 instancias benignas y 6 631 maliciosas, pero incurre en 2 901 falsos positivos y 1 702 falsos negativos. Esto confirma un sesgo hacia la detección de malware, con un coste elevado en términos de falsas alarmas.

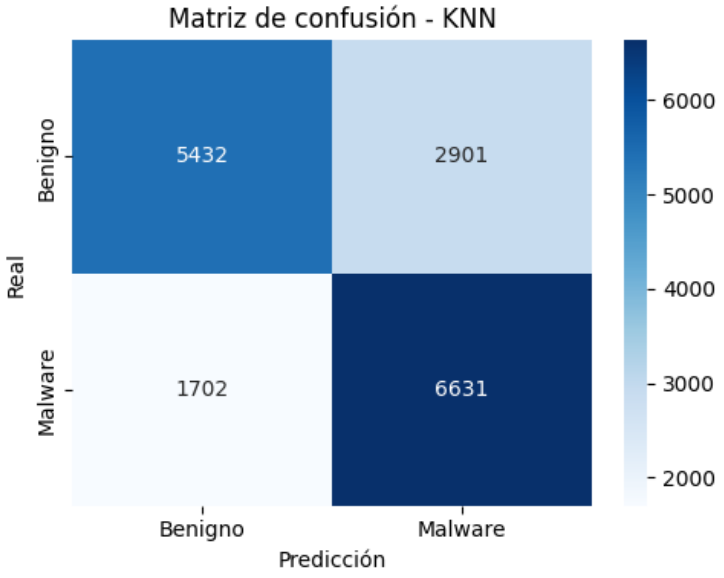


Figura 5.5: Matriz de confusión de KNN en el conjunto de test (16.6k muestras balanceadas).

Clase	**Precisión**	**Recall**	**F1-score**
0 (Benigno)	0.76	0.65	0.70
1 (Malware)	0.70	0.80	0.74
Macro avg (Promedio macro)	0.73	0.72	0.72
Weighted avg (Promedio ponderado)	0.73	0.72	0.72

Tabla 5.14: Informe de clasificación de KNN en el conjunto de test (16.6k muestras).

Valoración general

El modelo KNN muestra un rendimiento modesto si lo comparamos con técnicas más avanzadas. Aunque logra detectar buena parte del malware, también genera una cantidad considerable de falsos positivos. En un escenario real esto reduciría bastante su utilidad, ya que implicaría muchas alertas innecesarias sobre archivos legítimos. Su principal punto fuerte está en la simplicidad del enfoque y en que no necesita una fase de entrenamiento como tal, pero esa ventaja se compensa con un coste computacional alto en la fase de predicción, sobre todo en datasets grandes como EMBER. En este trabajo, KNN sirve como una referencia básica útil, aunque queda claro que su aplicación directa en entornos de producción es limitada frente a alternativas más sofisticadas.

5.1.6 Multi-Layer Perceptron (MLP)

El modelo **MLP** (Perceptrón Multicapa) constituye una red neuronal de tipo *feedforward* entrenada mediante retropropagación. En este trabajo se utilizó la implementación de `scikit-learn`, con una configuración por defecto de una capa oculta de 100 neuronas, función de activación ReLU y optimizador Adam con tasa de aprendizaje inicial de 0.001. El número máximo de iteraciones se fijó en 300, aplicando el criterio de parada temprana integrado en la librería para garantizar la convergencia.

Resultados

Los resultados obtenidos en el conjunto de test se presentan en la Tabla 5.15. El modelo alcanzó una **exactitud global del 75 %**, con un **F1 macro de 0.75** y un **AUC de 0.83**. La clase *malware* obtuvo un recall de 0.87, mientras que la clase *benigno* se vio más afectada, con un recall de 0.63. Esto refleja una tendencia del MLP a sobredetectar malware, lo que incrementa las falsas alarmas.

Métrica	Test Accuracy	Test F1 macro	AUC
MLP	0.75	0.75	0.83

Tabla 5.15: Rendimiento de MLP en el conjunto de test (16.6k muestras).

La matriz de confusión representada en la Figura 5.6 permite un análisis más detallado. El modelo clasifica correctamente 7 289 muestras de malware y 5 220 benignas, pero incurre en 3 113 falsos positivos y 1 044 falsos negativos. Estos valores son coherentes con el sesgo hacia la clase maliciosa que se aprecia también en el informe de clasificación (Tabla 5.16).

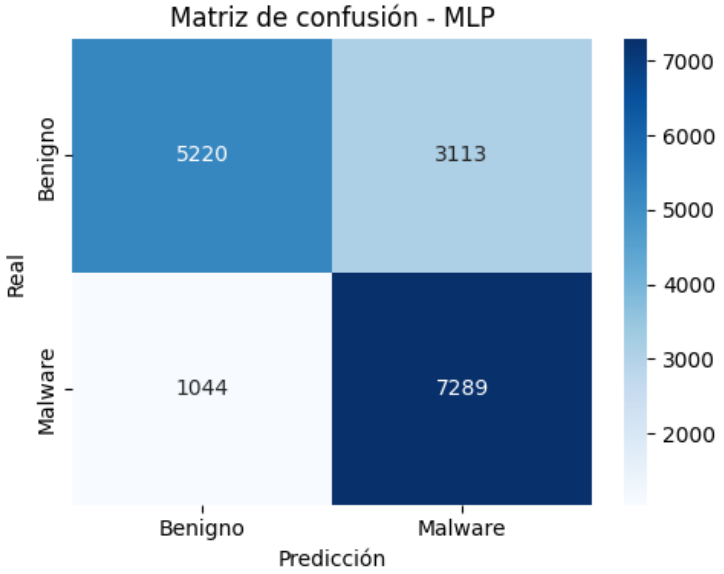


Figura 5.6: Matriz de confusión de MLP en el conjunto de test (16.6k muestras balanceadas).

Clase	Precisión	Recall	F1-score
0 (Benigno)	0.83	0.63	0.72
1 (Malware)	0.70	0.87	0.78
Macro avg (Promedio macro)	0.77	0.75	0.75
Weighted avg (Promedio ponderado)	0.77	0.75	0.75

Tabla 5.16: Informe de clasificación de MLP en el conjunto de test (16.6k muestras).

Valoración general

El MLP de `scikit-learn` puede considerarse un punto intermedio entre los modelos clásicos y las arquitecturas de deep learning. Aunque logra mejores resultados globales que la regresión logística y KNN, no llega al nivel de algoritmos ensemblados como Random Forest. Su capacidad para modelar relaciones no lineales le da cierta ventaja, pero también provoca un número alto de falsos positivos, lo que en un escenario real se traduciría en demasiadas alertas innecesarias. Aun con estas limitaciones, sigue siendo un modelo de referencia interesante dentro del marco experimental, ya que representa la transición hacia arquitecturas más complejas que se verán en el bloque de redes profundas.

5.2 Modelos de Deep Learning

En la sección anterior se evaluaron varios algoritmos clásicos de clasificación supervisada sobre las características vectorizadas del dataset EMBER. Estos modelos mostraron resultados bastante competitivos y consistentes entre sí, lo que confirma que incluso con enfoques tradicionales se puede lograr un buen rendimiento en la detección estática de malware. Aun así, tienen limitaciones claras: dependen de un diseño manual de características y su capacidad para capturar relaciones complejas entre variables es más reducida.

En esta parte se aborda la segunda línea de trabajo planteada: el uso de modelos de **aprendizaje profundo (Deep Learning)**. Concretamente, se implementaron tres arquitecturas neuronales —un perceptrón multicapa (MLP), una red convolucional unidimensional (CNN-1D) y una red recurrente LSTM— entrenadas directamente sobre los vectores de 2 381 características obtenidos en el pipeline de preprocesamiento. La idea es comprobar si estos modelos son capaces de aprender representaciones internas más ricas a partir de los datos y comparar su rendimiento con el de los algoritmos clásicos.

A continuación se describen las arquitecturas utilizadas, la configuración de entrenamiento y los resultados obtenidos sobre el conjunto de prueba. El análisis se centra tanto en métricas cuantitativas como en los errores que comete cada modelo, preparando el terreno para el capítulo siguiente, donde se aplicarán técnicas de explicabilidad (SHAP y LIME) tanto a modelos clásicos como a redes profundas.

5.2.1 Red neuronal MLP con Keras

Se implementó un modelo **Multilayer Perceptron (MLP)** mediante `Keras`, entrenado sobre el conjunto balanceado de 50 000 muestras estandarizadas. El objetivo es comprobar si una arquitectura neuronal puede capturar patrones no lineales de forma más eficiente que los clasificadores tradicionales.

Arquitectura del modelo

La red neuronal utilizada mantiene una estructura clásica pero efectiva:

- Capa de entrada de tamaño 2381 (una unidad por característica).
- Capa densa oculta con 256 neuronas y activación `ReLU`, seguida de `Dropout` del 30 %.
- Segunda capa oculta con 128 neuronas y activación `ReLU`, seguida de `Dropout` del 20 %.
- Capa de salida con una neurona y activación sigmoide para clasificación binaria.

Entrenamiento y convergencia

El entrenamiento se configuró con `Adam`, `binary_crossentropy`, *batch size* de 64 y un máximo de 50 épocas con *early stopping* (paciencia 5). La Tabla 5.17 resume las métricas de convergencia: la mejor precisión de validación se alcanzó en la época 6 (96.5 %), mientras que las últimas épocas mostraron ligeras oscilaciones y un aumento de la pérdida de validación, indicio de inicio de sobreajuste. Por ello no se observaron mejoras relevantes más allá de las 8 primeras épocas.

Época	Acc. (train)	Loss (train)	Acc. (val)	Loss (val)
1	0.9101	0.2236	0.9490	0.1288
2	0.9623	0.0979	0.9576	0.1112
3	0.9691	0.0753	0.9618	0.1053
4	0.9749	0.0649	0.9611	0.1102
5	0.9768	0.0567	0.9606	0.1228
6	**0.9800**	**0.0503**	**0.9650**	**0.1194**
7	0.9816	0.0460	0.9623	0.1356
8	0.9814	0.0462	0.9648	0.1302

Tabla 5.17: Resumen de convergencia del MLP (Keras). Mejor *val_accuracy* en la época 6 (=0.965).

Resultados en test

En el conjunto de test balanceado de 16.6k muestras, el modelo obtuvo un **accuracy del 76 %**, un **F1 macro de 0.76** y un **AUC de 0.85**. La Tabla 5.18 muestra el detalle por clase.

Clase	Precisión	Recall	F1-score
0 (Benigno)	0.81	0.68	0.74
1 (Malware)	0.72	0.84	0.78
Macro avg (Promedio macro)	0.76	0.76	0.76
Weighted avg (Promedio ponderado)	0.76	0.76	0.76

Tabla 5.18: Informe de clasificación del MLP implementado en Keras en el conjunto de test (16.6k muestras).

La matriz de confusión de la Figura 5.7 permite detallar los errores cometidos. El modelo clasifica correctamente 6970 muestras de malware y 5667 benignas, pero incurre en 2666 falsos positivos y 1363 falsos negativos. Estos resultados son coherentes con el sesgo hacia la detección de malware que también se refleja en el informe de clasificación.

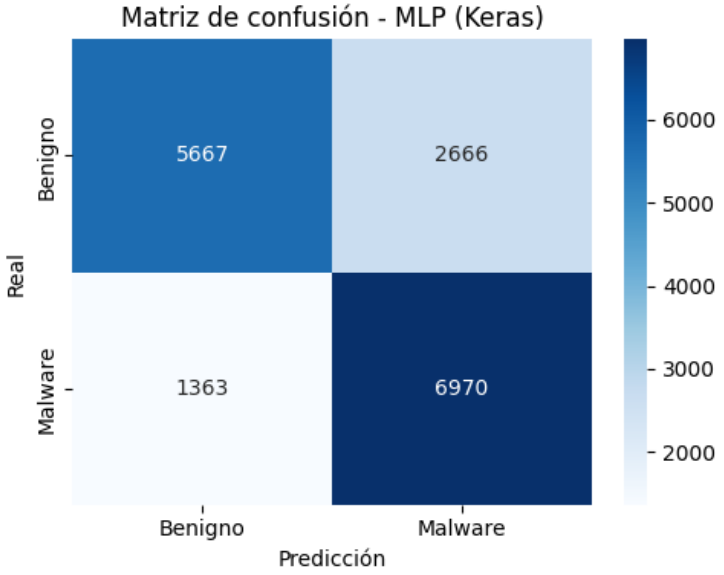


Figura 5.7: Matriz de confusión obtenida por el MLP en el conjunto de test (16.6k muestras balanceadas).

Valoración

Aunque este MLP no supera el rendimiento de los mejores modelos clásicos (como Random Forest, con un 82 % de exactitud), sí logra resultados competitivos y superiores a otros enfoques como la regresión logística o KNN. El modelo confirma la capacidad de las redes neuronales para capturar relaciones no lineales en los vectores de características, pero arrastra un problema recurrente: un número elevado de falsos positivos que comprometen su aplicabilidad en un escenario real. La discrepancia entre validación y test refleja además cierto sobreajuste, que podría mitigarse con una regularización más fuerte o un ajuste más exhaustivo de hiperparámetros. Aun así, constituye un punto de partida sólido para explorar arquitecturas más sofisticadas, como las redes convolucionales y recurrentes que se presentan a continuación.

5.2.2 Red neuronal convolucional 1D (CNN) con Keras

A continuación se evalúa el rendimiento de una red neuronal convolucional 1D implementada en `Keras`, entrenada sobre los vectores de características del dataset EMBER transformados para ser compatibles con capas convolucionales. Este tipo de red es especialmente útil cuando se desea capturar patrones locales dentro de secuencias ordenadas, como es el caso de los vectores de atributos extraídos.

Arquitectura del modelo

La arquitectura utilizada es la siguiente:

- Capa `Conv1D` con 64 filtros, tamaño de kernel 3 y activación `ReLU`.
- Capa de `MaxPooling1D` con tamaño de ventana 2.
- Dropout del 30 %.
- Segunda `Conv1D` con 128 filtros y activación `ReLU`.
- Otra capa de `MaxPooling1D` y aplanamiento (`Flatten`).
- Capa densa oculta de 128 neuronas con activación `ReLU`.
- Dropout del 30 %.
- Capa de salida con activación `sigmoid` para clasificación binaria.

El modelo se entrenó usando el optimizador `Adam` y la función de pérdida `binary_crossentropy`, con **early stopping** sobre la pérdida de validación (paciencia = 3) y batch size de 256.

Entrenamiento y convergencia

El entrenamiento se realizó durante un máximo de 20 épocas, aunque la red alcanzó su mejor precisión de validación (96.5 %) en la época 7. La Tabla 5.19 resume las métricas observadas durante las primeras épocas, donde se aprecia una rápida convergencia y un inicio de sobreajuste en validación a partir de la época 5.

Época	**Acc. (train)**	**Loss (train)**	**Acc. (val)**	**Loss (val)**
1	0.8423	0.4208	0.9479	0.1379
2	0.9518	0.1226	0.9556	0.1147
3	0.9638	0.0902	0.9595	0.1144
4	0.9699	0.0761	0.9595	0.1136
5	0.9751	0.0626	0.9628	0.1177
6	0.9736	0.0692	0.9584	0.1305
7	**0.9787**	**0.0538**	**0.9650**	**0.1200**

Tabla 5.19: Evolución del entrenamiento de la CNN 1D. Mejor *val_accuracy* en la época 7 (=0.965).

Resultados en test

En el conjunto de test (16.6k muestras balanceadas), la CNN alcanzó un **accuracy del 76 %**, un **F1 macro de 0.75** y un **AUC de 0.84**. La Tabla 5.20 recoge los valores detallados por clase.

Clase	**Precisión**	**Recall**	**F1-score**
0 (Benigno)	0.86	0.61	0.71
1 (Malware)	0.70	0.90	0.79
Macro avg (Promedio macro)	0.78	0.76	0.75
Weighted avg (Promedio ponderado)	0.78	0.76	0.75

Tabla 5.20: Informe de clasificación de la CNN 1D implementada en Keras en el conjunto de test (16.6k muestras).

Matriz de confusión

La matriz de confusión (Figura 5.8) confirma este comportamiento: la CNN clasifica correctamente 7515 instancias de malware y 5077 de benignos, pero incurre en 3256 falsos positivos y 818 falsos negativos.

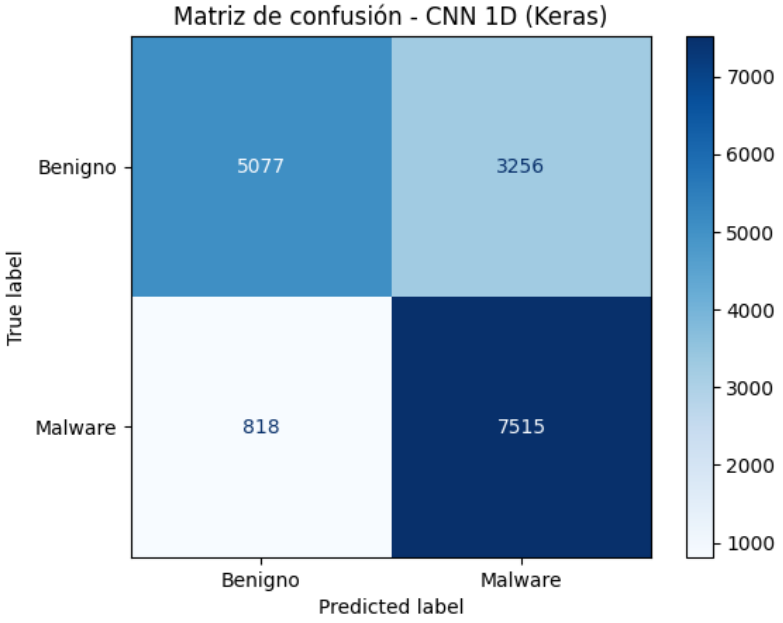


Figura 5.8: Matriz de confusión obtenida por el modelo CNN 1D en el conjunto de test (16.6k muestras).

Valoración

La CNN 1D demuestra que las convoluciones pueden aportar valor incluso sobre datos tabulares, logrando una alta sensibilidad en la detección de malware (recall de 0.90) a costa de una precisión reducida en la clase benigna. Este sesgo hacia clasificar como *malware* es coherente con el objetivo de minimizar falsos negativos en ciberseguridad, pero genera un número elevado de falsas alarmas (más de 3 200 en el conjunto de prueba). En términos globales, su rendimiento es comparable al del MLP y queda por debajo de los mejores modelos basados en árboles (como Random Forest, con un 82 % de exactitud). Estos resultados sugieren que, aunque las arquitecturas convolucionales son prometedoras, requieren ajustes adicionales o enfoques híbridos para mejorar su aplicabilidad práctica. Además, ponen de manifiesto la importancia de analizar no solo métricas cuantitativas, sino también las explicaciones detrás de las predicciones, aspecto que se aborda en el capítulo siguiente.

5.2.3 Red neuronal recurrente (LSTM) con Keras

Las redes neuronales recurrentes del tipo **Long Short-Term Memory (LSTM)** se diseñaron originalmente para procesar secuencias temporales y datos con dependencias a largo plazo. En este trabajo se planteó su aplicación sobre el conjunto de características vectorizadas del dataset EMBER, transformando los vectores de entrada en una representación tridimensional compatible con este tipo de arquitectura.

Arquitectura del modelo

El modelo implementado en `Keras` se compuso de las siguientes capas:

- Una capa `LSTM` de 64 unidades con `return_sequences=True`, que permite capturar dependencias locales a lo largo de la secuencia de 2381 características.
- Una capa de `Dropout` del 30 % para reducir sobreajuste.
- Una segunda capa `LSTM` de 32 unidades.
- Una capa densa intermedia de 64 neuronas con activación `ReLU`.
- Otra capa de `Dropout` del 30 %.
- Una capa de salida con activación sigmoide.

El entrenamiento se llevó a cabo con el optimizador `Adam`, función de pérdida `binary_crossentropy`, batch size de 256 y un máximo de 20 épocas, aplicando *early stopping* con paciencia de 3.

Entrenamiento y convergencia

A diferencia de los modelos anteriores, la LSTM no logró converger. Como se aprecia en la Tabla 5.21, tanto la pérdida como la exactitud se estabilizaron en valores cercanos al azar ya desde las primeras épocas, sin mejoras sustanciales en validación. El modelo alcanzó un techo del 50 % de exactitud, lo que equivale a clasificar de forma casi aleatoria en un problema binario balanceado.

Época	**Acc. (train)**	**Loss (train)**	**Acc. (val)**	**Loss (val)**
1	0.567	0.696	0.500	0.693
2	0.502	0.693	0.500	0.693
3	0.501	0.693	0.500	0.693
4	0.497	0.693	0.500	0.693
5	0.498	0.693	0.500	0.693

Tabla 5.21: Evolución del entrenamiento de la LSTM. El modelo no muestra mejoras en validación, manteniéndose en valores cercanos al azar.

Resultados en test

En el conjunto de test de 16.6k muestras balanceadas, la LSTM obtuvo un **accuracy del 50 %**, un **F1 macro de 0.33** y un **AUC de 0.50**. La Tabla 5.22 muestra el detalle por clase. El modelo predijo prácticamente todas las muestras como benignas, lo que se refleja en la matriz de confusión (Figura 5.9): se clasificaron correctamente las 8333 instancias benignas, pero apenas se identificó 1 muestra de malware sobre un total de 8333.

Clase	**Precisión**	**Recall**	**F1-score**
0 (Benigno)	0.50	1.00	0.67
1 (Malware)	1.00	0.00	0.00
Macro avg (Promedio macro)	0.75	0.50	0.33
Weighted avg (Promedio ponderado)	0.75	0.50	0.33

Tabla 5.22: Informe de clasificación de la LSTM implementada en Keras en el conjunto de test (16.6k muestras).

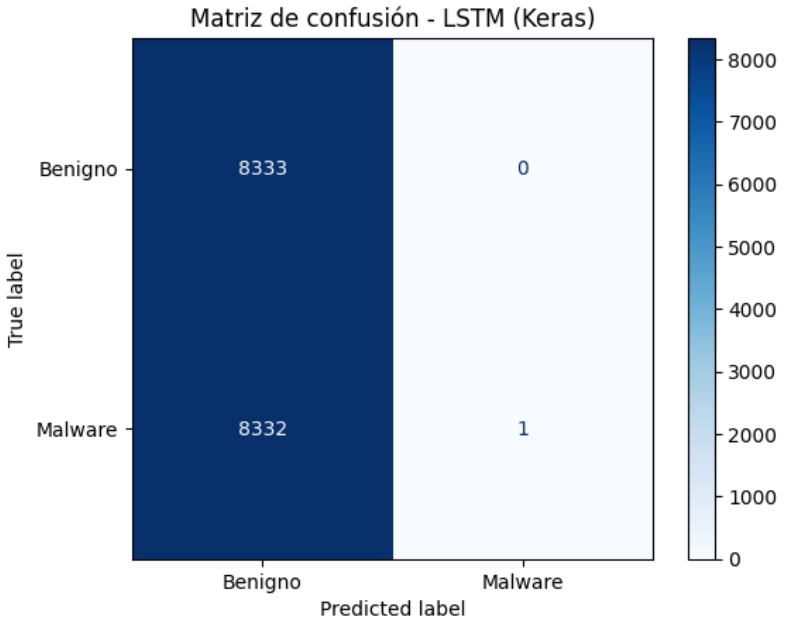


Figura 5.9: Matriz de confusión obtenida por el modelo LSTM en el conjunto de test (16.6k muestras).

Valoración

Los resultados tan flojos de la LSTM pueden explicarse por varias razones técnicas:

- **Naturaleza de los datos**: las LSTM están pensadas para trabajar con secuencias y capturar dependencias temporales. En este caso, el vector de 2381 características no sigue un orden secuencial real, sino que corresponde a atributos tabulares. Forzarle una estructura secuencial probablemente complica más el aprendizaje de lo que ayuda.

- **Dificultad de optimización**: los modelos recurrentes suelen tener problemas de gradiente (desvanecimiento o explosión), sobre todo con secuencias largas. Aunque las LSTM reducen este problema, la longitud de 2381 pasos sigue siendo excesiva para una arquitectura tan simple como la probada.

- **Capacidad limitada del modelo**: la red implementada, con solo 64 y 32 unidades recurrentes, puede quedarse corta para capturar patrones relevantes en un espacio de entrada tan grande.

- **Sobreajuste hacia la clase mayoritaria**: el hecho de que la red predijera casi

siempre la clase *benigna* indica que cayó en una solución trivial, minimizando la pérdida a costa de ignorar la clase minoritaria en el cálculo del gradiente.

En conjunto, estos resultados dejan claro que la LSTM no es una arquitectura adecuada para este tipo de representación vectorizada en la detección de malware. Dado que su desempeño estuvo cerca del azar y tendió a favorecer en exceso la clase benigna, este modelo no se incluye en el análisis de interpretabilidad del Capítulo 6, ya que no ofrece una base sólida para ser explicado.

A diferencia de los modelos MLP y CNN, que sí alcanzaron métricas comparables a los algoritmos clásicos, la LSTM muestra que no todas las arquitecturas de *deep learning* encajan bien con datos tabulares de alta dimensionalidad. Aun así, el experimento resulta útil porque marca los límites de cada enfoque y refuerza la idea de que la elección del modelo debe basarse en la naturaleza del problema y no solo en la popularidad de una técnica.

5.3 Discusión y análisis de resultados

5.3.1 Comparativa de rendimiento entre modelos

Los experimentos realizados permiten comparar de forma clara los distintos enfoques probados, tanto de aprendizaje automático clásico como de *deep learning*. A primera vista, los resultados siguen un mismo patrón: salvo la LSTM —que no logró superar un rendimiento cercano al azar—, todos los modelos alcanzaron métricas bastante parecidas entre sí. En general, las exactitudes globales quedaron en torno al 72–82 %, con valores de F1 macro y AUC dentro de intervalos similares, independientemente del tipo de algoritmo.

Entre los modelos clásicos, los basados en árboles (Random Forest y LightGBM) fueron especialmente competitivos y en algunos casos superaron a las redes neuronales. Random Forest ofreció un buen equilibrio entre precisión y recall, aunque con bastantes falsos positivos. LightGBM, en cambio, se inclinó más hacia la detección de malware, logrando alta sensibilidad pero a costa de una tasa de falsas alarmas elevada sobre archivos legítimos.

Los clasificadores de margen y de distancia (SVM y KNN) repitieron ese sesgo hacia la clase maliciosa: recalls altos para malware, pero bajos para benignos. La regresión logística, usada como modelo lineal de referencia, mostró un desempeño aceptable, situándose en un punto intermedio. Por su parte, el MLP de `scikit-learn` actuó como un puente entre

los clásicos y las redes profundas: superó a los modelos lineales, pero sin llegar al nivel de los métodos ensemblados.

En cuanto a los modelos profundos implementados en `Keras`, tanto el MLP como la CNN alcanzaron métricas muy similares a las de los algoritmos clásicos. Esto confirma que, al menos con el dataset EMBER en su versión vectorizada, añadir más complejidad con arquitecturas neuronales no supone necesariamente una mejora clara en el rendimiento. De hecho, las diferencias entre ellos fueron mínimas, lo que resalta la solidez de los enfoques clásicos en este dominio.

En resumen, la comparativa de rendimiento muestra que no hay un ganador absoluto en términos puramente cuantitativos: los modelos clásicos y los neuronales rinden de forma muy parecida, con pequeñas diferencias en el balance entre precisión y recall. Esta convergencia abre una pregunta clave que guía el siguiente capítulo: si todos los modelos obtienen métricas tan similares, ¿cómo decidir cuál es más adecuado en un entorno real de ciberseguridad? La respuesta requiere ir más allá de los números y enfocarse en la interpretabilidad de las predicciones, donde la Inteligencia Artificial Explicable (XAI) cobra un papel central.

5.3.2 Impacto del preprocesamiento y la vectorización

El pipeline de preprocesamiento y vectorización de características del dataset EMBER juega un papel fundamental en los resultados obtenidos. A diferencia de otros contextos en los que se trabaja directamente con datos crudos (como secuencias de bytes o trazas de ejecución), aquí se parte de 2381 variables que ya resumen información estática importante de los binarios, como distribuciones de bytes, entropías de secciones, importaciones de librerías o metadatos del ejecutable. Esta forma compacta de representar los archivos facilita aplicar desde modelos lineales sencillos hasta redes neuronales profundas.

La estandarización de todas las variables tuvo un efecto especialmente positivo en los algoritmos sensibles a la escala (regresión logística, SVM, KNN y MLP). Sin normalización, estos modelos habrían tomado decisiones sesgadas por la diferencia de magnitudes entre características. Aunque los algoritmos basados en árboles no necesitan este paso, se aplicó igualmente a todo el conjunto para mantener consistencia, y los resultados mostraron que su rendimiento no se vio afectado.

Otro punto a tener en cuenta es que el vector de características ya incluye una selección manual hecha por los creadores del dataset, lo que reduce la ventaja que podrían tener

los modelos de *deep learning* frente a los clásicos. En vez de aprender representaciones directamente de datos sin procesar, las redes reciben un vector ya estructurado y bastante compacto. Esto ayuda a entender por qué sus métricas no se diferencian mucho de las que logran modelos como Random Forest o SVM.

En resumen, el impacto del preprocesamiento y la vectorización es doble: por un lado, permite que modelos muy distintos trabajen sobre la misma base de forma comparable; y por otro, condiciona qué patrones pueden descubrirse, limitando el valor extra que aportan arquitecturas más complejas. Esto refuerza la idea de que, en la detección de malware, la calidad del diseño de las características de entrada es tan importante como el modelo de aprendizaje que se use.

5.3.3 Conclusión del análisis

El análisis comparativo realizado en este capítulo muestra que, a pesar de las diferencias entre los modelos clásicos y los de aprendizaje profundo, los resultados en métricas como precisión, recall, F1 y AUC son bastante parecidos. Con la excepción de la LSTM, que tuvo un rendimiento claramente insuficiente, el resto de algoritmos alcanzaron valores similares y con patrones de error próximos entre sí. Esto sugiere que, al menos usando el dataset EMBER con una vectorización fija, la elección del modelo por sí sola no garantiza una mejora sustancial en la detección de malware.

Ahora bien, quedarse solo con las métricas numéricas puede dar una visión incompleta. Dos modelos con resultados globales similares pueden diferir mucho en la forma en que toman decisiones y en las características en las que se apoyan. En un campo tan sensible como la ciberseguridad, donde la validación humana y la confianza en el sistema son clave, la interpretabilidad de los modelos se vuelve fundamental.

Por eso, en el siguiente capítulo se introduce el análisis de explicabilidad con librerías de XAI (SHAP y LIME), con el objetivo de complementar las métricas vistas aquí y valorar hasta qué punto cada modelo ofrece explicaciones claras y útiles para los analistas de seguridad.

6. Explicabilidad

En el capítulo anterior vimos un panorama curioso: usando el dataset EMBER, la mayoría de los modelos —tanto clásicos como de *deep learning*— obtuvieron resultados muy parecidos en métricas como precisión, F1 y AUC. Pero en un entorno real esto no resuelve el dilema principal: necesitamos un sistema que no solo acierte, sino que además permita entender y auditar sus decisiones. En la detección de malware, esta trazabilidad es esencial: un falso positivo puede bloquear software legítimo y generar costes extra, mientras que un falso negativo abre la puerta a incidentes graves. Por eso, además de medir, es igual de importante *comprender* qué señales está usando el modelo.

Este capítulo se centra en esa parte, aplicando técnicas de *post-hoc explainability*, que permiten analizar modelos ya entrenados sin cambiar su arquitectura. El objetivo es doble: (i) comprobar si los algoritmos aprenden patrones con sentido dentro del dominio y (ii) comparar la *calidad* de las explicaciones entre modelos cuando las métricas numéricas no bastan para decidir.

Se trabajará con dos niveles de análisis que se complementan:

- **Global**: identifica, en promedio, qué características influyen más en el comportamiento del modelo. Esto sirve para auditar si la señal proviene de elementos estructurales del ejecutable (p. ej., cabecera PE o secciones) o de indicadores más débiles (p. ej., bins de entropía).

- **Local**: explica por qué se generó una predicción en un caso concreto. Es clave para revisar ejemplos frontera y diagnosticar la causa de falsos positivos o falsos negativos.

En cuanto a herramientas, se emplean **SHAP** y **LIME**. SHAP descompone la salida del modelo en contribuciones por característica —lo que permite análisis tanto globales como locales—, mientras que LIME genera modelos lineales aproximados alrededor de una muestra concreta para dar una explicación rápida de esa predicción.

Para dar sentido a las explicaciones, se vinculan los índices de las características del vector EMBER con sus *bloques* (Tabla 4.1): `histogram`, `byteentropy`, `strings`, `general`, `header`, `section`, `imports`, `exports` y `datadirectories`. Este mapeo permite interpretar cada contribución en términos de partes del formato PE, lo que facilita conclusiones útiles para el analista.

La exposición se organiza en dos partes. Primero se analizan los **modelos clásicos** (regresión logística, Random Forest y LightGBM), que si bien ofrecieron un rendimiento competitivo, como veremos, sus explicaciones tienden a ser más fragmentadas e inestables. Después se pasa a los **modelos de *deep learning*** (MLP y CNN 1D en Keras; la LSTM se descarta por su bajo rendimiento), donde comprobaremos que las explicaciones obtenidas con SHAP y LIME resultan más coherentes, repetibles y alineadas con la semántica del PE. Esta evidencia ayudará a resolver el "empate" que vimos en las métricas: en este escenario, la explicabilidad se convierte en el criterio clave que inclina la balanza hacia las arquitecturas de *deep learning*.

6.1 Explicabilidad en modelos clásicos

Los modelos clásicos de aprendizaje automático ofrecen una ventaja clave frente a arquitecturas más complejas: su capacidad de ser interpretados con relativa facilidad. A pesar de que su rendimiento pueda ser ligeramente inferior al de ensembles más sofisticados o a redes neuronales profundas, su valor reside en que permiten comprender con claridad qué señales están influyendo en cada decisión. En un dominio como la ciberseguridad, donde las predicciones deben ser auditables y explicables para generar confianza, esta cualidad se convierte en un recurso fundamental.

Regresión logística. La regresión logística constituye probablemente el modelo más directo de interpretar, ya que cada coeficiente asociado a una característica actúa como un peso que incrementa o reduce la probabilidad de que un archivo sea clasificado como *malware*. En nuestro caso, el modelo se entrenó sobre el subconjunto balanceado de 50 000 vectores normalizados, lo cual garantiza que la magnitud de los coeficientes sea comparable entre atributos. La Figura 6.1 muestra los doce coeficientes de mayor magnitud, mientras que la Tabla 6.1 presenta esos mismos índices mapeados a los bloques del dataset EMBER, junto con una breve interpretación.

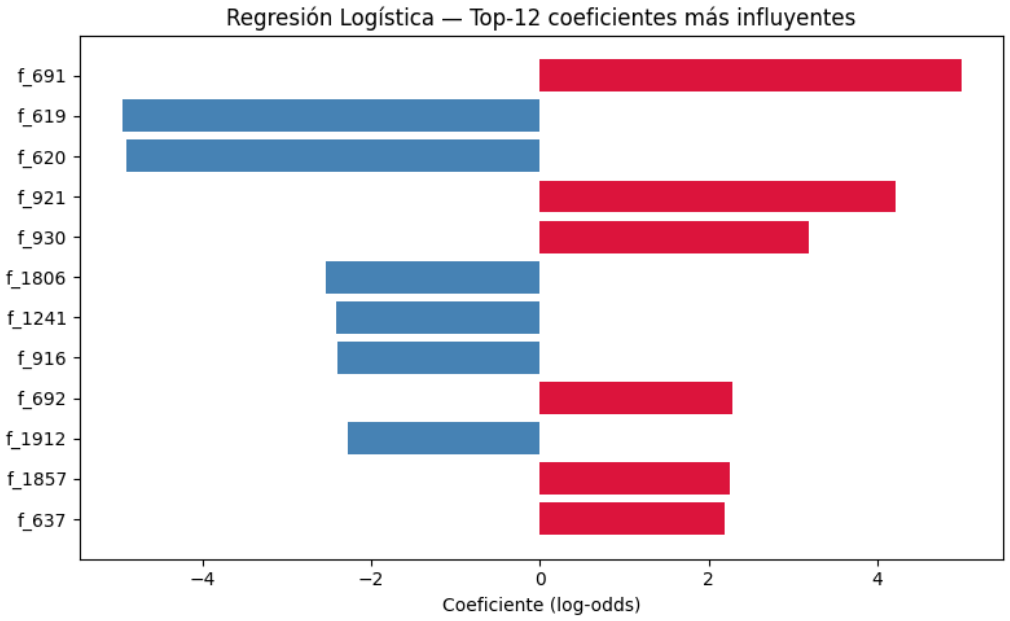


Figura 6.1: Regresión Logística: 12 coeficientes con mayor magnitud (signo: rojo → *malware*, azul → *benigno*).

Feature (índice)	**Bloque**	**Signo**	**Interpretación**
f_691	section	+	Señal estructural de sección poco común, asociada a empaquetado.
f_619	general	–	Metadato global típico de software legítimo.
f_620	general	–	Rasgo coherente con binarios normales (tamaño/recursos).
f_921	section	+	Sección atípica que eleva probabilidad de *malware*.
f_930	section	+	Variación estructural vinculada a técnicas de ofuscación.
f_1806	imports	–	Conjunto de DLLs frecuentes en software benigno.
f_1241	imports	–	Patrón de importaciones legítimas.
f_916	section	–	Sección estándar que arrastra la predicción hacia benigno.
f_692	section	+	Irregularidad en secciones que incrementa el riesgo.
f_1912	imports	–	Importaciones poco sospechosas, comunes en ejecutables legítimos.
f_1857	imports	+	Import concreto con fuerte asociación a actividad maliciosa.
f_637	header	+	Cabecera PE con metadatos inusuales (flags/timestamps).

Tabla 6.1: Top-12 coeficientes de la regresión logística: bloque EMBER, signo e interpretación.

El análisis revela tres patrones claros. En primer lugar, los coeficientes positivos se concentran en el bloque `section` y en parte de `header`, lo que apunta a que el modelo percibe como señales de riesgo las estructuras de secciones poco comunes y los metadatos anómalos de cabecera. Estos comportamientos son consistentes con técnicas de empaquetado y ofuscación propias de binarios maliciosos. En segundo lugar, varios coeficientes negativos aparecen en el bloque `imports`, indicando que ciertos perfiles de importaciones frecuentes

en programas legítimos desplazan la predicción hacia la clase *benigno*. Este hallazgo refuerza la idea de que la ausencia de APIs sospechosas y la presencia de bibliotecas estándar son fuertes indicadores de legitimidad. Por último, los atributos en el bloque `general` actúan como señales de equilibrio: metadatos coherentes con software convencional reducen la probabilidad de etiquetado como malicioso.

Este ejercicio demuestra que incluso un modelo lineal sencillo es capaz de capturar patrones con sentido de dominio. Si bien la regresión logística no detecta interacciones no lineales entre bloques de características, sí ofrece una visión clara y auditable de las señales en las que se apoya el sistema. Para un analista de seguridad, esta transparencia resulta fundamental: permite validar rápidamente que el clasificador está aprendiendo comportamientos razonables y facilita la detección de sesgos o errores de modelado.

Random Forest. En **Random Forest** la interpretabilidad no proviene de coeficientes directos, sino de medidas *post-hoc* como SHAP, que descomponen la salida en contribuciones por rasgo y permiten leer efectos no lineales e interacciones entre árboles.

La Figura 6.2 recoge el *ranking* global (media de |SHAP|) y muestra un patrón claro: un par de atributos de `header` (p. ej., `f_626`, `f_637`) concentran mucha importancia, seguidos por una *cola larga* de variables de `byteentropy`. Esta mezcla es coherente con el dominio (la cabecera PE concentra metadatos sensibles y la entropía refleja empaquetado/ofuscación), pero también adelanta un rasgo menos deseable del bosque: gran parte de la señal se reparte en decenas de *micro-contribuciones* de entropía, lo que dificulta articular una regla simple y estable.

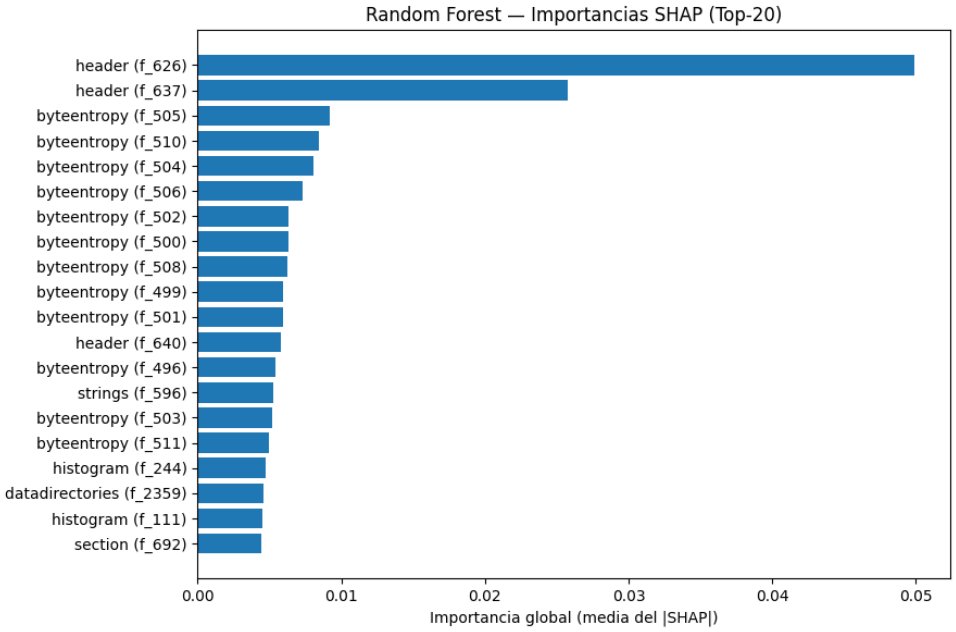


Figura 6.2: Importancias globales SHAP para Random Forest (Top-20 características).

La Figura 6.3 añade la direccionalidad y la dispersión de los efectos. En los rasgos de `header` se observan impactos consistentes hacia *malware* cuando sus valores son altos. Sin embargo, en `byteentropy` los puntos se agrupan en torno a cero con colas pronunciadas en ambos sentidos: tanto valores muy altos como muy bajos pueden empujar la predicción. Esta asimetría sugiere que el modelo está captando patrones reales, pero a costa de explicaciones *variables* según el vecindario de la muestra; un comportamiento que, en la práctica, puede traducirse en falsos positivos sobre binarios legítimos comprimidos o con secciones atípicas.

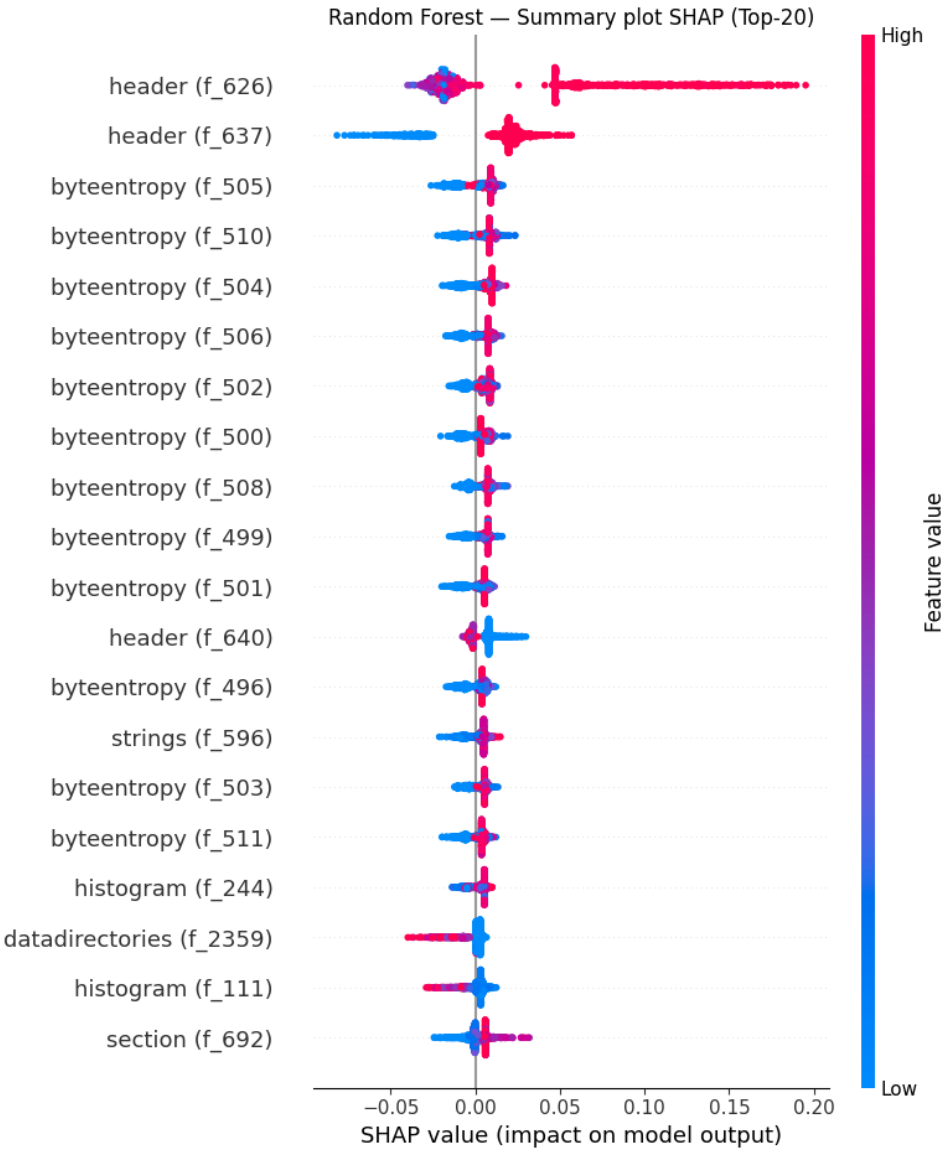


Figura 6.3: Summary plot de SHAP para Random Forest (Top-20 características).

Otro aspecto a destacar es la menor presencia del bloque `imports` en el *top*. A diferencia de otros modelos, el bosque parece apoyarse casi por completo en `header` y `byteentropy`, relegando patrones de importaciones que, en seguridad, suelen aportar contexto semántico más estable.

En conjunto, Random Forest ofrece una interpretación razonable del problema, pero sus explicaciones tienden a ser **más fragmentadas y ruidosas**: muchas variables con impacto pequeño, alta variabilidad local y fuerte dependencia de bins de entropía. Aunque el rendimiento global es competitivo, esta inestabilidad interpretativa dificulta convertir SHAP en reglas operativas robustas. Este punto será relevante más adelante al comparar

con los modelos de *deep learning*, cuyas explicaciones resultan más coherentes y repetibles a nivel global y local.

LightGBM. Entre los algoritmos de ensamble evaluados, **LightGBM** fue el que alcanzó un mejor rendimiento en métricas como precisión y AUC. No obstante, su carácter de *caja negra* obliga a recurrir a técnicas de explicabilidad para entender en qué señales se apoya realmente el modelo.

La Figura 6.4 presenta el *summary plot* de SHAP. En él se aprecia un predominio de atributos de `header` (como `f_626` y `f_637`), acompañados de varias variables de `section`, `imports` y `datadirectories`. En principio, este resultado es coherente con el dominio, pues los metadatos de cabecera y las secciones atípicas son indicadores clásicos de manipulación en binarios maliciosos. Sin embargo, la dispersión de los puntos alrededor de cero revela una cierta inestabilidad: el modelo reparte importancia en muchos rasgos pequeños, lo que dificulta sintetizar reglas claras y consistentes.

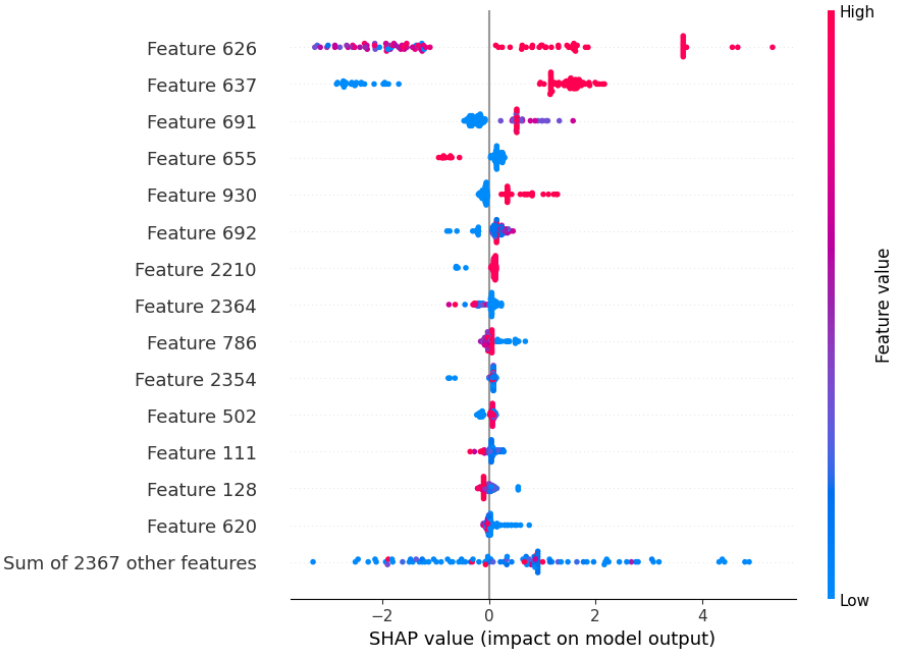


Figura 6.4: Summary plot de SHAP para LightGBM: importancia global de características.

La Figura 6.5 muestra un *decision plot* de SHAP para una muestra clasificada como *malware* con probabilidad 0.74. La predicción surge de la combinación de múltiples factores: varios atributos empujan hacia la clase maliciosa (zonas rojas), mientras que otros moderan la decisión hacia benigno (zonas azules). Aunque este tipo de visualización resulta útil para desglosar el proceso, la explicación se basa en la suma de muchas contribuciones

pequeñas, lo que reduce su claridad para un analista que busca señales dominantes.

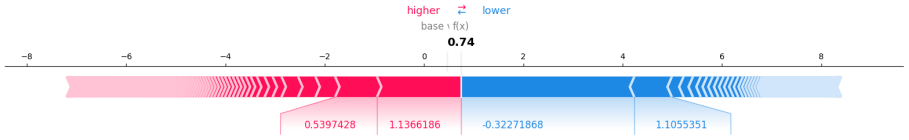


Figura 6.5: Decision plot de SHAP para LightGBM: explicación local de una muestra.

De forma complementaria, la Figura 6.6 muestra la explicación generada con LIME sobre la misma instancia. Aquí los factores aparecen como barras, en rojo para impulsar la predicción hacia `malware` y en verde hacia `benigno`. Aunque existe cierta coherencia con los resultados de SHAP —por ejemplo, en la influencia de secciones y valores de entropía—, también se aprecian diferencias en los atributos concretos destacados. Esto apunta a que la explicación local de LightGBM varía en función de la técnica empleada, lo que añade un grado de inestabilidad interpretativa.

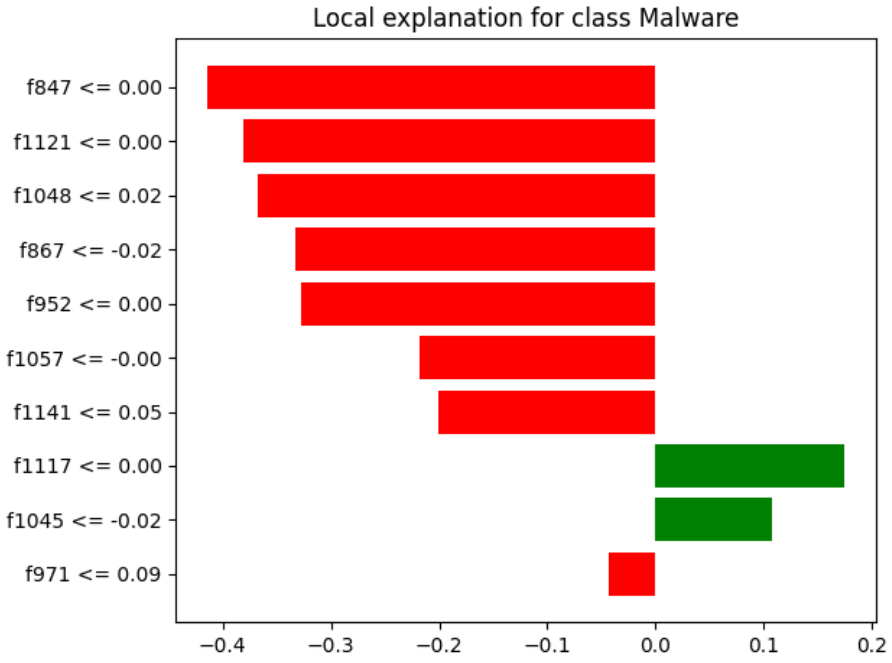


Figura 6.6: Explicación local con LIME para LightGBM: barras de impacto por característica.

En conjunto, LightGBM combina un **alto rendimiento predictivo** con explicaciones razonables, pero éstas resultan **más fragmentadas y menos consistentes** que en otros

modelos. A nivel global, la importancia se reparte en un gran número de variables con contribuciones pequeñas, y a nivel local, SHAP y LIME no siempre coinciden plenamente en la narrativa. Esto dificulta transformar las explicaciones en reglas operativas robustas. Por tanto, aunque LightGBM es competitivo en métricas, su explicabilidad resulta más frágil frente a las arquitecturas de *deep learning*, que como se verá ofrecen patrones más coherentes y repetibles.

6.2 Explicabilidad en modelos de Deep Learning

Los modelos de *Deep Learning* destacan por su capacidad para representar de forma muy flexible los datos y detectar relaciones complejas entre variables que suelen escaparse a los algoritmos clásicos de *Machine Learning*. En el capítulo anterior vimos que, en cuanto a métricas, su rendimiento era bastante parecido al de los modelos tradicionales. Sin embargo, su valor real aparece cuando analizamos la **explicabilidad**: aunque a menudo se perciben como *cajas negras*, las redes profundas pueden ofrecer patrones de interpretación más claros y estructurados siempre que se utilicen las herramientas adecuadas.

En ciberseguridad, esta capacidad no es un detalle menor. Un sistema de detección de malware no solo debe acertar, también tiene que ser transparente y auditable. Mientras que técnicas como la regresión logística o los árboles permiten obtener explicaciones rápidas a través de coeficientes o reglas, las redes neuronales necesitan herramientas de *post-hoc explainability* más avanzadas, como **SHAP** y **LIME**. Estas herramientas descomponen cada predicción en contribuciones de las variables y permiten visualizar qué bloques del dataset (cabeceras, secciones, entropía, cadenas o importaciones) han tenido más peso en la decisión del modelo.

En lo que sigue se presentan los resultados para dos arquitecturas representativas: un perceptrón multicapa (MLP) y una red convolucional (CNN). La LSTM se descarta por el bajo rendimiento mostrado en el capítulo anterior, por lo que no entra en este análisis. En ambos casos se incluyen ejemplos de explicabilidad global y local, y se comparan con los modelos clásicos. Como veremos, aunque la complejidad interna de las redes neuronales es mayor, las explicaciones obtenidas resultan **más consistentes y con mayor sentido semántico**, lo que aporta un criterio diferenciador frente a algoritmos tradicionales con métricas similares.

MLP (Keras). El perceptrón multicapa (MLP) no expone coeficientes ni reglas legi-

bles como los modelos clásicos, por lo que su lectura requiere técnicas *post-hoc*. En nuestro caso, combinamos **SHAP** (global y local) y **LIME** (local). El objetivo no es sólo "explicar" una salida, sino verificar que la red está apoyándose en señales con *sentido de dominio* dentro del formato PE.

Mapa de *features* a bloques EMBER. Usando los rangos de la Tabla 4.1, los índices que aparecen en nuestras figuras se mapean a:

- **header (626–687):** `f_626`, `f_637`, `f_654`, `f_655`, `f_658`, `f_679`, `f_684`.
- **section (688–942):** `f_691`, `f_692`, `f_704`, `f_760`, `f_775`, `f_865`, `f_887`, `f_906`, `f_918`, `f_921`, `f_930`.
- **general (616–625):** `f_618`, `f_621`, `f_622`.
- **imports (943–2222):** `f_946`, `f_960`, `f_1004`, `f_1031`, `f_1052`, `f_1094`, `f_1101`, `f_1117`, `f_1141`, `f_1176`, `f_1190`, `f_1194`, `f_1424`, `f_1432`, `f_1608`, `f_1987`, `f_2037`, `f_2070`, `f_2210`.
- **strings (512–615):** `f_598`, `f_604`.
- **histogram (0–255):** `f_123`.
- **byteentropy (256–511):** `f_464`.

Lectura global (SHAP). La Figura 6.7 muestra el *summary plot* del MLP. A diferencia de RF/LightGBM, donde el peso se dispersa en muchos bins de `byteentropy`, aquí emergen patrones **compactos y coherentes**: dominan rasgos de `header` (p. ej., `f_637`, `f_626`, `f_654`/`f_655`/`f_658`) y de `section` (p. ej., `f_691`, `f_692`, `f_775`, `f_921`, `f_930`), acompañados por `general` (`f_618`, `f_621`, `f_622`) y algunos `imports` (`f_1608`). Esto *cuadra* con el análisis PE: la cabecera concentra *timestamps*, *flags* y campos de tamaño/alineamiento; las secciones recogen número, tamaños y entropías; y los imports reflejan el perfil API. La direccionalidad es interpretable: valores altos en `header`/`section` atípicos empujan hacia *malware*; metadatos `general` coherentes tiran de la predicción hacia *benigno*. En cambio, `byteentropy`/`histogram` (p. ej., `f_464`, `f_123`) tienen menos protagonismo que en los árboles, lo que reduce sensibilidad a empaquetadores y, por ende, el riesgo de falsos positivos por ofuscación.

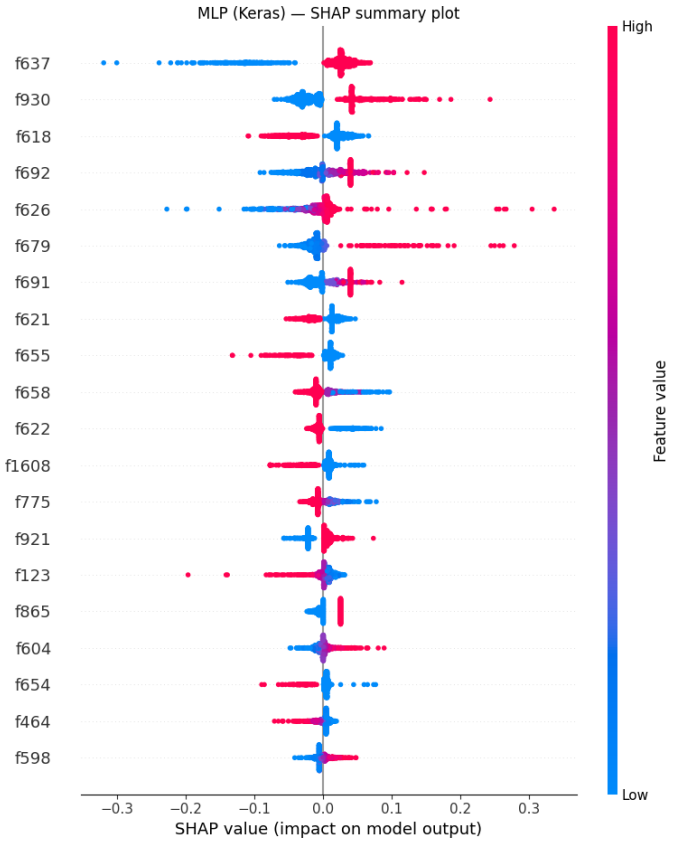


Figura 6.7: MLP (Keras) — SHAP *summary plot*.

Lectura local (LIME + SHAP-waterfall). En la Figura 6.8, LIME explica una muestra etiquetada como `malware`. Los rasgos que más empujan a `malware` (rojo) son `section` (`f_887`, `f_760`, `f_704`) y `imports` (`f_1052`, `f_1004`, `f_1094`, `f_1176`, `f_1194`); el único de `header` destacado es `f_684`. La mezcla es razonable: secciones con tamaños/entropías inusuales y un patrón de importaciones concreto son señales típicas de familias empaquetadas o con carga dinámica.

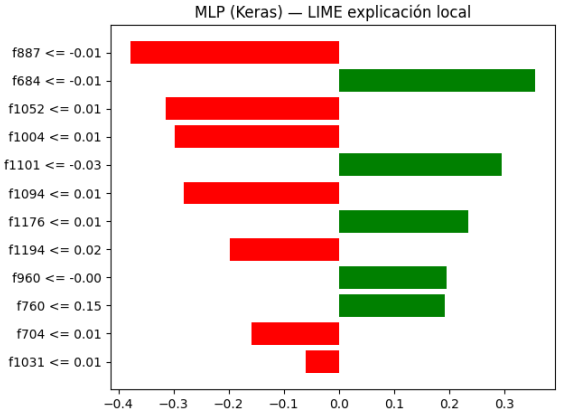


Figura 6.8: MLP (Keras) — LIME explicación local.

La Figura 6.9 confirma la **coherencia global–local**. El *waterfall* suma pocas contribuciones de gran tamaño y muchas de magnitud despreciable; entre las relevantes aparecen `imports` (`f_946`, `f_1608`, `f_1987`, `f_2037`, `f_2210`, `f_2070`, `f_1424`, `f_1432`, `f_1190`) y `section/general` (`f_930`, `f_775`, `f_618`, `f_622`). Esto tiene sentido en ingeniería inversa:

- Un patrón de **imports** concreto (aunque EMBER los codifique hasheados) suele delatar familias/behaviors: manipulación de procesos, red, cifrado, *shellcode*, etc.

- **Secciones** con tamaños/entropías anómalos y combinaciones inusuales (p. ej., `.text` muy pequeña + `.rsrc` grande) son típicas de empaquetadores.

- **Metadatos de cabecera** fuera de lo común (timestamps nulos o "futuros", *characteristics* no estándar) son indicios de manipulación.

En conjunto, la decisión del MLP se explica con **pocos rasgos principales** y una narrativa estable (mismo vocabulario de bloques que en el análisis global), lo que facilita la validación pericial.



Figura 6.9: MLP (Keras) — SHAP *waterfall* de un ejemplo local.

Discusión. El MLP aprende señales *estructurales* del PE (cabecera, secciones y metadatos generales) y las complementa con el perfil de *imports*. Esto resulta más **estable y semánticamente rico** que el apoyo de los árboles en numerosos bins de `byteentropy`. Además, la coincidencia entre el *summary* global y las explicaciones locales (LIME/SHAP) sugiere que la red mantiene un criterio consistente entre muestras. Desde la práctica de análisis de malware, esto es valioso por dos motivos: (i) reduce la propensión a falsos positivos en ejecutables comprimidos/ofuscados, y (ii) produce explicaciones *accionables* (campos de cabecera, secciones o imports concretos a revisar) que el analista puede contrastar con herramientas forenses. En un contexto donde las métricas de rendimiento empatan, esta **calidad de la explicación** inclina la balanza a favor del *Deep Learning*.

Redes Convolucionales (CNN). El uso de redes convolucionales sobre los vectores del dataset EMBER ofrece una forma distinta de abordar el problema respecto a los modelos clásicos. Mientras que algoritmos como la regresión logística o los árboles se centran en relaciones directas entre variables, la CNN aprende representaciones locales gracias a los filtros convolucionales, lo que le permite capturar patrones combinados entre distintas partes del vector, como conexiones entre bytes y atributos de secciones PE. Esta capacidad extra también hace que el modelo sea más difícil de interpretar, de ahí la necesidad de

apoyarse en técnicas de explicabilidad post-hoc.

En la Figura 6.10 se muestra el *summary plot* de SHAP. Los atributos con mayor influencia se concentran en bloques con sentido semántico claro: `f637` y `f626` (cabecera PE), `f930`, `f921` y `f692` (secciones) y, en menor medida, `f618` (información general del binario). Este patrón encaja con lo visto en otros modelos: irregularidades en la cabecera o estructuras de sección poco habituales suelen ser señales fiables de malware. Aunque los valores absolutos de SHAP son algo menores que los observados en Random Forest o LightGBM, la CNN consigue concentrar la importancia en un conjunto reducido de atributos con fuerte sentido de dominio.

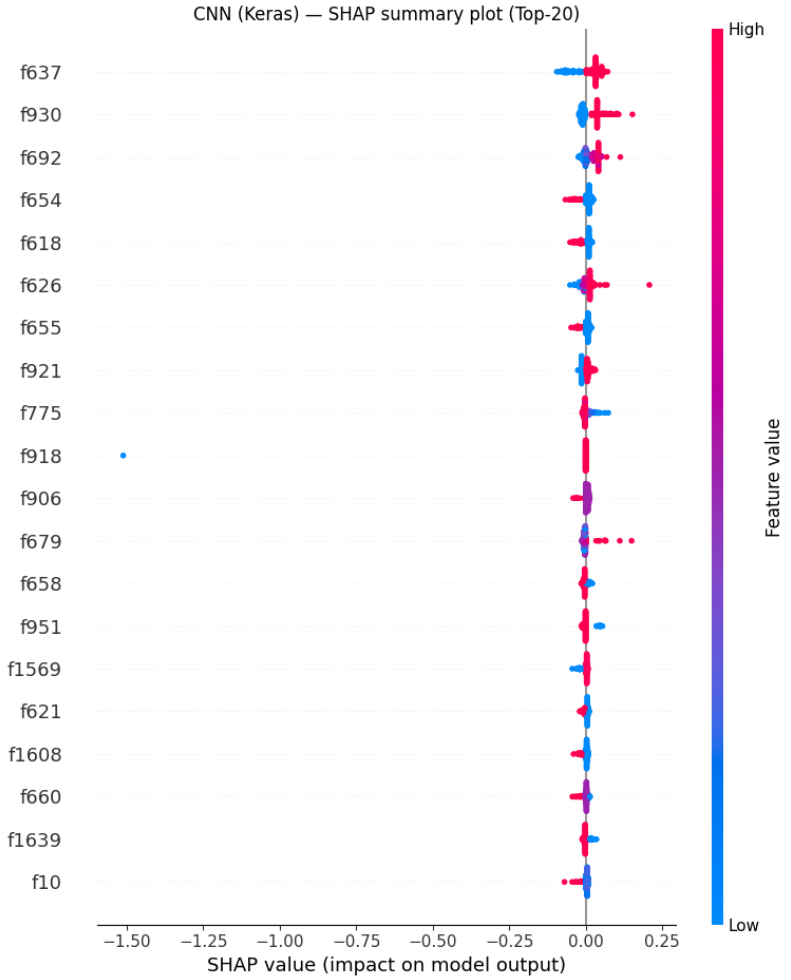


Figura 6.10: CNN (Keras) — *Summary plot* de SHAP (Top-20).

Para ilustrar la direccionalidad de estas señales, la Figura 6.11 presenta dos gráficos de dependencia. En el caso de `f637` (cabecera), el impacto se comporta de manera casi binaria: ciertos valores desplazan hacia benigno, mientras que otros refuerzan con fuerza

la hipótesis de *malware*. Por su parte, la característica `f930` (sección) muestra un patrón progresivo: a medida que el valor aumenta, crece la probabilidad de clasificación como malicioso. Esto sugiere que el modelo está captando secciones con tamaños o distribuciones poco frecuentes en binarios legítimos.

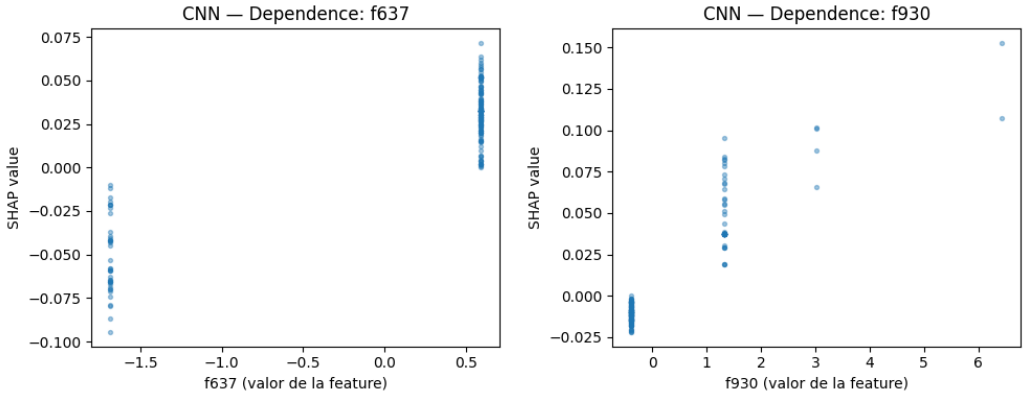


Figura 6.11: CNN (Keras) — Gráficos de dependencia SHAP para `f637` (header) y `f930` (section).

En conjunto, los resultados muestran que la CNN no sólo reproduce señales ya detectadas por modelos clásicos (cabeceras anómalas, secciones irregulares), sino que además aporta un nivel de granularidad adicional en la forma en que estas variables interactúan. De este modo, aunque sigue siendo un modelo de "caja negra", las explicaciones obtenidas con SHAP revelan que sus decisiones se apoyan en características con sentido real en el análisis de binarios de malware, reforzando su utilidad práctica frente a algoritmos más fragmentados como LightGBM.

Conclusiones sobre explicabilidad en Deep Learning

El análisis de la explicabilidad en las redes neuronales deja varias ideas interesantes. Tanto el MLP como la CNN terminan fijándose en los mismos indicadores que ya habían aparecido en los modelos clásicos: campos anómalos en la `header`, estructuras poco comunes en las `sections` y, en algunos casos, patrones concretos en las `imports`. Que algoritmos diferentes coincidan en estas señales es positivo, ya que confirma que realmente están captando propiedades relevantes dentro del análisis de binarios.

La principal diferencia está en cómo se organizan las explicaciones. En modelos como Random Forest o LightGBM, la importancia suele dispersarse entre muchas características, a menudo correlacionadas, lo que dificulta quedarse con un mensaje claro. En cambio, las

redes neuronales tienden a concentrar la atención en un grupo más pequeño de atributos, lo que hace que la interpretación sea más directa y menos redundante. Con el MLP, las visualizaciones de SHAP y LIME muestran de forma bastante clara cómo ciertas cabeceras o secciones empujan la predicción hacia *malware* o hacia *benigno*. En la CNN, los gráficos de dependencia revelan relaciones más matizadas: hay variables que actúan casi como interruptores (como `f637`), mientras que otras muestran un efecto progresivo (por ejemplo, `f930`).

En conjunto, esto sugiere que, aunque las redes profundas se suelen ver como *cajas negras*, con las herramientas adecuadas es posible obtener explicaciones incluso más claras y coherentes que en los modelos clásicos. Por tanto, cuando varios algoritmos alcanzan métricas similares en términos de rendimiento, la calidad de estas explicaciones puede ser el factor decisivo. Desde este punto de vista, los modelos de *Deep Learning* aportan una ventaja extra: generan interpretaciones más compactas, menos dispersas y más cercanas a la lógica real de cómo se manipulan los binarios maliciosos.

7. Conclusiones y trabajo futuro

7.1 Conclusiones

El presente Trabajo Fin de Máster ha tenido como objetivo central diseñar y comparar un sistema de detección de malware binario en modo estático, combinando algoritmos clásicos de aprendizaje automático y arquitecturas de aprendizaje profundo, y evaluando no sólo su rendimiento, sino también su capacidad de ser interpretados mediante técnicas de explicabilidad. A continuación se presentan las conclusiones principales, organizadas en torno a los objetivos planteados al inicio del trabajo.

1. **Revisión bibliográfica**. El análisis del estado del arte permitió constatar que la detección de malware ha transitado desde modelos clásicos (árboles de decisión, SVM, Random Forest) hacia enfoques de aprendizaje profundo, especialmente CNN y variantes de RNN. Asimismo, se identificó un creciente interés en la explicabilidad como requisito para la adopción en ciberseguridad. En este contexto, librerías como `SHAP` y `LIME` se consolidan como herramientas de referencia para aportar interpretaciones post-hoc.

2. **Preparación del dataset EMBER**. Se realizó con éxito la vectorización completa de las 600.000 muestras etiquetadas (benignas y maliciosas), eliminando las instancias sin etiqueta y aplicando normalización con `StandardScaler`. El muestreo estratificado generó subconjuntos balanceados de 50.000 instancias para entrenamiento y 16.600 para prueba, permitiendo ejecutar experimentos de forma eficiente sin perder representatividad.

3. **Entrenamiento de modelos clásicos**. Se implementaron y evaluaron regresión logística, Random Forest, LightGBM, SVM, KNN y un MLP de `scikit-learn`. Los mejores resultados se obtuvieron con Random Forest y LightGBM, alcanzando F1-macro en torno al 0.80–0.82 y AUC cercanos al 0.90. Estos modelos confirmaron su solidez y su idoneidad como líneas base en el problema.

4. **Entrenamiento de modelos de deep learning**. Se desarrollaron arquitecturas en Keras: un MLP denso, una CNN 1D y una LSTM. Los resultados mostraron un comportamiento desigual: el MLP y la CNN lograron métricas competitivas,

comparables a los modelos clásicos, mientras que la LSTM sufrió sobreajuste y un rendimiento deficiente, lo que limitó su utilidad en el análisis final.

5. **Evaluación comparativa de rendimiento**. En términos de accuracy, precisión, recall y F1, los modelos clásicos y profundos mostraron un desempeño similar (72–82 % de F1 macro). Esta cercanía evidenció que las métricas tradicionales por sí solas no permiten establecer un "ganador" claro, reforzando la necesidad de criterios adicionales para la selección de modelos.

6. **Aplicación de técnicas de XAI**. Las librerías SHAP y LIME se usaron tanto en modelos clásicos como en redes neuronales. En Random Forest y LightGBM, SHAP permitió identificar la relevancia de características como la entropía de secciones y las importaciones sospechosas, mientras que en redes neuronales (MLP y CNN) se observó una mayor coherencia en la atribución de importancia a bloques semánticamente interpretables (cabecera PE, secciones, histogramas de bytes). LIME, por su parte, facilitó explicaciones locales útiles para analizar casos frontera.

7. **Análisis de interpretabilidad**. La comparación mostró que los modelos de *deep learning*, pese a ser considerados habitualmente "cajas negras", ofrecieron explicaciones más compactas y consistentes gracias a SHAP. Los modelos clásicos tendieron a repartir la importancia entre un número elevado de características correlacionadas, dificultando la extracción de conclusiones claras. Este hallazgo sugiere que la combinación de redes neuronales con técnicas de explicabilidad puede resultar más adecuada en escenarios reales de ciberseguridad.

8. **Síntesis y conclusiones**. En conjunto, se confirma que la mera comparación de métricas predictivas no basta para decidir entre modelos. La explicabilidad emerge como criterio diferenciador, permitiendo validar que las predicciones se basan en señales legítimas del dominio y aumentando la confianza de los analistas. Así, los modelos basados en *deep learning*, particularmente el MLP y la CNN, se perfilan como alternativas prometedoras: alcanzan un rendimiento competitivo y, con el apoyo de técnicas de XAI, ofrecen un marco de interpretación claro y útil para la toma de decisiones en ciberseguridad.

7.2 Líneas de trabajo futuro

Este trabajo deja abiertas varias rutas interesantes que pueden ayudar a mejorar tanto la capacidad predictiva como la utilidad práctica y explicativa de los sistemas de detección de malware basados en aprendizaje automático.

Una primera posibilidad sería ampliar el alcance experimental al dataset EMBER completo e incluso combinarlo con otros repositorios de muestras, como VirusShare o colecciones obtenidas en entornos de sandboxing. De esta manera se podría comprobar hasta qué punto los modelos generalizan frente a un abanico más amplio de familias de malware, incluyendo variantes recientes u ofuscadas que quizá no estén presentes en EMBER.

Otra línea prometedora es incorporar características dinámicas extraídas de ejecuciones controladas con herramientas como Cuckoo Sandbox. Este tipo de enfoque capturaría trazas de comportamiento en tiempo real (secuencias de llamadas a API, accesos de red, consumo de recursos, etc.), complementando la visión estática y aportando señales adicionales para mejorar la detección.

También resulta interesante explorar técnicas de selección y reducción de características que aprovechen la información de explicabilidad. Estrategias como la poda guiada por valores de SHAP o la proyección mediante PCA o UMAP podrían simplificar la representación de los binarios, mantener su capacidad discriminativa y, al mismo tiempo, hacer más fácil la interpretación global, algo especialmente útil en escenarios con recursos limitados.

Otra idea con gran potencial es integrar la explicabilidad en tiempo real dentro de plataformas de monitorización y respuesta a incidentes. Así, cada alerta podría acompañarse de evidencias claras y auditables que respalden la decisión humana en entornos críticos como el financiero, sanitario o gubernamental.

Igualmente, cabe la opción de diseñar arquitecturas híbridas que combinen modelos altamente interpretables (como los árboles de decisión) con otros más complejos y expresivos (como redes neuronales profundas). Bajo este esquema, las muestras más sencillas podrían clasificarse con el modelo interpretable, dejando las más ambiguas al modelo profundo. De esta forma se alcanzaría un equilibrio práctico entre transparencia y rendimiento.

Finalmente, sería interesante probar representaciones alternativas de los ejecutables, más allá del vector estático de EMBER. Opciones como la ejecución dinámica, el análisis de secuencias de *opcodes* desensamblados o la conversión de binarios en representaciones visuales han mostrado buenos resultados en trabajos recientes. Aun así, estas aproxima-

ciones tienen retos claros: ejecutar en sandbox es costoso y vulnerable a evasiones, el desensamblado puede fallar con binarios ofuscados, y las representaciones visuales corren el riesgo de perder semántica si no están bien diseñadas. Enfoques como MCTVD (Deng y cols., 2023), que combinan distintas visualizaciones, son prometedores aunque añaden un nivel de complejidad extra.

En conjunto, estas posibles líneas de investigación no solo apuntan a mejorar las métricas de detección, sino también a avanzar hacia sistemas más robustos, explicables y confiables, capaces de funcionar en escenarios reales donde precisión y transparencia son igual de importantes.

Referencias

Aamir, M., Ali, A. S., Ishmanov, F., y cols. (2024). Amddl model: Android smartphones malware detection using deep learning. *PLoS One*, *19*(1), e0296722.

Aboaoja, F. A., Zainal, A., Ghaleb, F. A., Al-rimy, B. A. S., Eisa, T. A. E., y Elnour, A. A. H. (2022). Malware detection issues, challenges, and future directions: A survey. *Applied Sciences*, *12*(17), 8482.

Anderson, H. S. (2018). *Ember: An open dataset for training static pe malware ml models.* `https://github.com/elastic/ember`. (Accessed: 2025-07-27)

Anderson, H. S., y Roth, P. (2018). *Ember: An open dataset for training static pe malware machine learning models.* Descargado de `https://arxiv.org/abs/1804.04637`

Deng, H., Guo, C., Shen, G., Cui, Y., y Ping, Y. (2023, 03). Mctvd: A malware classification method based on three-channel visualization and deep learning. *Computers Security*, *126*, 103084. doi: 10.1016/j.cose.2022.103084

Ferdous, J., Islam, R., Mahboubi, A., y Islam, M. Z. (2025). A survey on ml techniques for multi-platform malware detection: Securing pc, mobile devices, iot, and cloud environments. *Sensors*, *25*(4). Descargado de `https://www.mdpi.com/1424-8220/25/4/1153` doi: 10.3390/s25041153

Gaber, M. G., Ahmed, M., y Janicke, H. (2024). Malware detection with artificial intelligence: A systematic literature review. *ACM Computing Surveys*, *56*(6), 148.

Galli, A., La Gatta, V., Moscato, V., Postiglione, M., y Sperlì, G. (2024). Explainability in ai-based behavioral malware detection systems. *Computers & Security*, *130*, 103842.

Gaspar, D., Silva, P., y Silva, C. (2024). Explainable ai for intrusion detection systems: Lime and shap applicability on multi-layer perceptron. *IEEE Access*, *12*, 30164-30175. doi: 10.1109/ACCESS.2024.3368377

Gupta, U., Kandpal, S., Alamro, H., Asiri, M. M., Alanazi, M. H., Al-Sharafi, A. M., y Sorour, S. (2025). Efficient malware detection using nlp and deep learning model. *Alexandria Engineering Journal*, *124*, 550-564. Descargado de `https://www.sciencedirect.com/science/article/pii/S1110016825004260` doi: https://doi.org/10.1016/j.aej.2025.03.118

Johny, J. A., P., V., A., A. K., Radhamani, G., A., R. R. K., y Conti, M. (2024). *Deep learning fusion for effective malware detection: Leveraging visual features.* Descargado de `https://arxiv.org/abs/2405.14311`

Kakisim, A. G., Gulmez, S., y Sogukpinar, I. (2022). Sequential opcode embedding-based malware detection method. *Computers & Electrical Engineering*, *98*, 107703.

Letoffe, O., Huang, X., y Marques-Silva, J. (2024). *Towards trustable shap scores.* Descargado de `https://arxiv.org/abs/2405.00076`

Lu, X., Zhao, J., Zhu, S., y Liò, P. (2024). Sndgcn: Robust android malware detection based on subgraph network and denoising gcn. *Expert Systems with Applications*, *250*, 123922.

Madamidola, O. A., Ngobigha, F., y Ez-zizi, A. (2025). Detecting new obfuscated malware variants: A lightweight and interpretable machine learning approach. *Intelligent Systems with Applications*, *25*, 100472.

Manthena, H., Shajarian, S., Kimmell, J., Abdelsalam, M., Khorsandroo, S., y Gupta, M. (2024). Explainable artificial intelligence (xai) for malware analysis: A survey of techniques, applications, and open challenges. *IEEE Access*. (Early Access)

Nataraj, L., Karthikeyan, S., Jacob, G., y Manjunath, B. S. (2011). Malware images: visualization and automatic classification. En *Proceedings of the 8th international symposium on visualization for cyber security.* New York, NY, USA: Association for Computing Machinery. Descargado de `https://doi.org/10.1145/2016904.2016908` doi: 10.1145/2016904.2016908

Nazim, S., Alam, M. M., Rizvi, S. S., Mustapha, J. C., Hussain, S. S., y Suud, M. M. (2025). Advancing malware imagery classification with explainable deep learning: A state-of-the-art approach using shap, lime and grad-cam. *PLoS One*, *20*(5), e0318542.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Rahali, A., y Akhloufi, M. A. (2023). Malbertv2: Code aware bert-based model for malware identification. *Big Data and Cognitive Computing*, *7*(2), 60.

Saqib, M., Mahdavifar, S., Fung, B. C. M., y Charland, P. (2024). A comprehensive analysis of explainable AI for malware hunting. *ACM Computing Surveys*, *57*(6), 119.

Song, Y., Zhang, D., Wang, J., Wang, Y., Wang, Y., y Ding, P. (2025). Application of deep learning in malware detection: a review. *Journal of Big Data*, *12*(99).

Sun, Y., Chen, H., Guo, J., Sun, A., Li, Z., y Liu, H. (2025). *Roma: Robust malware attribution via byte-level adversarial training with global perturbations and adversarial*

consistency regularization. Descargado de https://arxiv.org/abs/2502.07492

Tupadha, L. S., y Stamp, M. (2021). *Machine learning for malware evolution detection.* Descargado de https://arxiv.org/abs/2107.01627

Wajahat, A., Murtaza, S., Bakhshi, H., y cols. (2024). Outsmarting android malware with cutting-edge feature engineering and machine learning techniques. *Computers, Materials & Continua*, *79*(1), 651–673.

Wu, Y., Zhuang, H., Jia, Y., y Zhang, Y. (2025). A survey of machine learning approaches for malware detec-tion. En *Proceedings of the 2025 5th international conference on computer network security and software engineering* (p. 269–273). New York, NY, USA: Association for Computing Machinery. Descargado de https://doi.org/10.1145/3732365.3732410 doi: 10.1145/3732365.3732410