UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

TFG del Grado en Ingeniería Informática

Detección de *malware* mediante técnicas de Inteligencia Artificial

Presentado por David Cezar Toderas
en Universidad de Burgos — 6 de julio de 2025
Tutor: Álvar Arnaiz González

UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

D. Álvar Arnaiz González, profesor del departamento de Ingeniería Informática, Área de Lenguajes y Sistemas Informáticos.

Expone:

Que el alumno D. David Cezar Toderas, con DNI X8300130M, ha realizado el Trabajo final de Grado en Ingeniería Informática titulado título de TFG.

Y que dicho trabajo ha sido realizado por el alumno bajo la dirección del que suscribe, en virtud de lo cual se autoriza su presentación y defensa.

En Burgos, 6 de julio de 2025

Vº. Bº. del Tutor:

Álvar Arnaiz González

Agradecimientos

Este trabajo se ha construido sobre dos pilares: el apoyo incondicional de mi familia, que ha sido mi ancla, y la infinita paciencia y confianza de mi tutor, que ha sido mi guía. A mis amigos, gracias por entender que mi ausencia no era olvido, sino la dedicación que este proyecto exigía, y por seguir ahí con el mismo ánimo de siempre. A todos vosotros, por ser el motor, el refugio y la razón por la que cada hora invertida en estas páginas ha merecido la pena.

Resumen

El crecimiento exponencial del sistema operativo Android lo ha convertido en el principal objetivo del *software* malicioso. Este trabajo aborda el desarrollo de un sistema de detección de *malware* para Android basado en inteligencia artificial y análisis estático, un enfoque que permite evaluar la seguridad de las aplicaciones sin necesidad de ejecutarlas. El proyecto se desarrolló en dos fases: una primera de prototipado para validar la viabilidad del enfoque, y una segunda de ingeniería en la que se construyó un *pipeline* completo, incluyendo la creación de un *dataset* propio con 20 000 aplicaciones y el desarrollo de un modelo de red neuronal a medida.

Los resultados demuestran la alta eficacia del sistema, con modelos capaces de alcanzar un *recall* y un F_1-*Score* superiores al 98 %. La conclusión más relevante del trabajo es el papel fundamental del *embedder* de la red neuronal, un componente que transforma los datos brutos en una representación numérica de alta calidad. Este *embedder* no solo permite que la red neuronal funcione, sino que, al ser utilizado como preprocesador para algoritmos de aprendizaje automático clásicos (como XGBoost o Regresión Logística), potencia su rendimiento a niveles excepcionales, demostrando el poder de las arquitecturas híbridas.

Descriptores

detección de *malware*, android, análisis estático, inteligencia artificial, redes neuronales, aprendizaje profundo, *embeddings*, interpretabilidad de modelos, SHAP, ciberseguridad.

Abstract

The exponential growth of the Android operating system has made it the primary target for malicious software. This project addresses the development of a malware detection system for Android based on artificial intelligence and static analysis, an approach that allows for the assessment of application security without the need for execution. The project was carried out in two phases: an initial prototyping phase to validate the approach's feasibility, and a second engineering phase in which a complete pipeline was built, including the creation of a custom dataset with 20,000 applications and the development of a tailored neural network model.

The results demonstrate the high effectiveness of the system, with models capable of achieving recall and F1-Score metrics exceeding 98 %. The most significant conclusion of this work is the fundamental role of the neural network's embedder, a component that transforms raw data into a high-quality numerical representation. This embedder not only enables the neural network to function but also, when used as a preprocessor for classical machine learning algorithms (such as XGBoost or Logistic Regression), boosts their performance to exceptional levels, showcasing the power of hybrid architectures.

Keywords

malware detection, android, static analysis, artificial inteligence, neural network, deep learning, embeddings, model interpretability, SHAP, cibersecurity.

Índice general

Índice de figuras

Índice de tablas

1. Introducción

La hegemonía de Android en el panorama móvil, superando el 70 % del mercado mundial [54], ha traído consigo una consecuencia inevitable: es el blanco predilecto del software malicioso. Para hacer frente a esta amenaza constante, este trabajo explora el potencial de la inteligencia artificial, y en concreto de las redes neuronales profundas, para automatizar la detección de *malware* a través del análisis estático, una metodología segura que evita la ejecución de código potencialmente peligroso.

El recorrido de este proyecto se puede dividir en dos actos. El primero fue un acto de exploración, donde se construyó un prototipo inicial en base a un *dataset* público llamado Drebin. Su éxito confirmó que una red neuronal podía aprender a diferenciar aplicaciones seguras de maliciosas, pero también nos enfrentó a una dura realidad: sin un proceso de extracción de datos transparente, el modelo estaba «atado» al *dataset* con el que fue creado, sin capacidad para analizar nuevas amenazas. Esta limitación, lejos de ser un fracaso, fue el punto de inflexión que dio sentido y propósito a la segunda parte del viaje.

El segundo acto, fue por tanto, un ejercicio de ingeniería. Se construyó un *dataset* propio de 20 000 aplicaciones con un *pipeline* de extracción de características totalmente replicable. Sobre esta nueva base, se entrenó un modelo de red neuronal que no solo alcanzó un rendimiento excelente (con un *recall* cercano al 99 %), sino que también nos llevó a la conclusión más importante del trabajo: la verdadera fortaleza del modelo no residía en su clasificación final, sino en su *embedder*. Este componente demostró ser una herramienta tan potente que, al usarlo para alimentar a modelos clásicos como XGBoost, estos lograron un rendimiento igual o incluso superior, validando los enfoques híbridos como una estrategia sumamente eficaz.

2. Objetivos del proyecto

Este apartado explica de forma precisa y concisa cuáles son los objetivos que se persiguen con la realización del proyecto. Se puede distinguir entre los objetivos marcados por los requisitos del software a construir y los objetivos de carácter técnico que plantea a la hora de llevar a la práctica el proyecto.

2.1. Objetivos funcionales

Los objetivos funcionales definen las capacidades y acciones que el sistema final debe ofrecer al usuario. Son, en esencia, las características con las que se podrá interactuar directamente.

- **Clasificación de aplicaciones Android:** El sistema debe ser capaz de recibir un archivo APK y analizarlo para determinar si es malicioso o benigno, devolviendo un resultado claro.

- **Desarrollo de una interfaz web de demostración:** Crear una aplicación web simple e intuitiva que permita a un usuario subir un archivo APK y obtener la predicción del modelo de forma visual.

- **Interpretabilidad del modelo:** Ser capaz de explicar el porqué de las decisiones del modelo y reconocer cuales son los factores que más afectan a este y entender cómo realiza las predicciones que estima.

2.2. Objetivos técnicos

Los objetivos técnicos se refieren a las metas de implementación, arquitectura y proceso necesarias para construir el sistema y asegurar su calidad y funcionalidad interna.

- **Investigación del estado del arte:** Realizar un análisis de la literatura científica existente para comprender las técnicas actuales de detección de *malware* con IA y posicionar el proyecto.
- **Creación de un dataset propio:** Diseñar y construir un conjunto de datos a medida, extrayendo características útiles de una cantidad suficiente de ficheros APK.
- **Implementación de un *pipeline* de extracción de características:** Desarrollar un proceso completo y replicable que transforme los datos brutos de una APK en el formato numérico que el modelo necesita para su análisis.
- **Diseño y entrenamiento del modelo de red neuronal:** Construir un modelo de *deep learning* con PyTorch, entrenarlo con el *dataset* propio y optimizar su rendimiento para que sea capaz de clasificar *malware* correctamente.
- **Ajuste de hiperparámetros:** Utilizar herramientas de búsqueda automatizada, como Optuna, para encontrar la configuración de hiperparámetros que ofrezca el mejor equilibrio entre rendimiento y tiempo de entrenamiento.
- **Cuantización del modelo:** Intentar reducir el tamaño del modelo todo lo posible para optimizar su rendimiento y facilitar su despliegue en otras plataformas.
- **Análisis comparativo de modelos:** Entrenar clasificadores clásicos de aprendizaje automático (como KNN, SVM, Linear Regression, Random Forest o XGBoost) usando las características procesadas por la red neuronal y comparar sus resultados.

2.3. Objetivos personales

Estos objetivos describen las competencias y conocimientos que se esperan adquirir a nivel personal durante el desarrollo del proyecto.

- **Aplicar conocimientos de IA en un proyecto real:** Llevar la teoría a la práctica, diseñando, entrenando y evaluando un modelo de red neuronal desde cero con Python.
- **Adquirir experiencia en investigación académica:** Aprender a buscar, leer y sintetizar documentación científica, tanto para fundamentar las decisiones técnicas del proyecto, como para mejorar mi compresión a la hora de leer estudios de cualquier tipo.
- **Profundizar en IA y ciberseguridad:** Ganar un mejor entendimiento acerca de estos dos grandes campos que hoy en día dominan el mundo de la informática. Comprender a su vez mejor cómo funciona el *malware* y las técnicas que pueden usarse para detectarlo y combatirlo.
- **Obtener un modelo útil y con un buen rendimiento:** Ser capaz de entrenar un modelo que pueda competir con otros. El objetivo principal en este caso es obtener un modelo que sea capaz de acertar en la mayoría de los casos y que dé predicciones bastante fiables.

3. Conceptos teóricos

En esta sección se definen aquellos conceptos que son necesarios conocer para comprender el resto del documento.

3.1. Conceptos de Inteligencia Artificial

Para comprender cómo una máquina puede aprender a identificar amenazas, primero debemos sentar las bases de la inteligencia artificial (IA). Este apartado introduce las ideas fundamentales que hacen posible este proyecto, comenzando por el concepto general de la IA, centrándose en el aprendizaje automático como su motor principal. Se explora cómo los modelos computacionales pueden identificar patrones en los datos y el importante proceso de transformar información cruda, como un archivo, en un formato que la máquina pueda entender. El objetivo es ofrecer una visión clara del camino que se sigue para construir y entrenar un modelo de IA desde cero.

Inteligencia artificial

La inteligencia artificial (IA) es una disciplina de la informática que busca desarrollar sistemas capaces de realizar tareas que tradicionalmente asociamos con la inteligencia humana, como el razonamiento, el aprendizaje o la percepción del entorno. Más que intentar replicar la mente humana, un objetivo todavía lejano, la IA se centra en crear herramientas que puedan procesar información de forma autónoma y tomar decisiones para resolver problemas concretos. La IA nos proporciona un conjunto de técnicas para abordar estos desafíos, abriendo la puerta a nuevas formas de automatización y análisis.

El enorme interés que vemos hoy en día por la IA se debe a una combinación de factores que han coincidido en el tiempo: la disponibilidad de cantidades masivas de datos para entrenar los modelos, el diseño de algoritmos de aprendizaje cada vez más eficaces y, sobre todo, el acceso a una gran capacidad de cómputo gracias al abaratamiento de costes y el desarrollo de mejores Unidades de Procesamiento Gráfico (GPU). Estas circunstancias han permitido que la IA deje de ser un campo puramente académico para convertirse en una tecnología con aplicaciones en prácticamente cualquier sector.

Gracias a ello, la IA ha impulsado avances en un espectro muy amplio de aplicaciones. En el procesamiento del lenguaje (NLP), ha sido la base para mejorar los traductores automáticos [62] y ha permitido el desarrollo de asistentes virtuales mediante el desarrollo de la arquitectura de los *transformers* [58]. En el campo de la visión por computador, las técnicas de IA son fundamentales en sistemas de reconocimiento facial o en la asistencia al diagnóstico médico mediante imágenes [50]. Otros campos, como los sistemas de recomendación [39] o la conducción autónoma [40], son desafíos complejos donde la IA es un componente esencial, aunque su desarrollo y perfeccionamiento siguen siendo un trabajo en curso.

Aprendizaje automático

El aprendizaje automático (*machine learning* o ML) es el subcampo de la inteligencia artificial centrado en crear algoritmos que aprenden patrones directamente de los datos, sin necesidad de ser programados con reglas explícitas para cada tarea. La premisa fundamental del aprendizaje automático es que, al exponer a un modelo a una cantidad suficientemente grande de ejemplos, este puede identificar patrones, correlaciones y estructuras subyacentes en los datos para, posteriormente, realizar predicciones o tomar decisiones sobre datos nuevos y nunca antes vistos.

El tipo de datos y el nivel de guía que se le proporciona al modelo definen los principales paradigmas de aprendizaje que existen. El más extendido es el **aprendizaje supervisado** [35], donde el modelo se entrena con un conjunto de datos completamente etiquetado, es decir, cada ejemplo de entrada está emparejado con una salida, comúnmente conocida como etiqueta, o clase. El objetivo es que el modelo aprenda a mapear cada entrada con su salida correcta. En el otro extremo se encuentra el **aprendizaje no supervisado** [41], donde el modelo recibe datos sin ninguna etiqueta y su tarea es encontrar por sí mismo cualquier estructura o agrupación inherente en ellos, como organizar una biblioteca sin conocer los géneros de los libros.

Entre estos dos extremos, surge una solución pragmática: el **aprendizaje semisupervisado** [57]. Este enfoque es ideal cuando etiquetar datos es caro o laborioso, ya que utiliza un pequeño conjunto de datos etiquetados junto con una gran cantidad de datos sin etiquetar. La idea es que la estructura de los datos no etiquetados ayude al modelo a generalizar mejor a partir de las pocas etiquetas que tiene. Finalmente, el **aprendizaje por refuerzo** [37] funciona de manera distinta, ya que no se basa en aprender de un *dataset* estático. En este caso, un agente aprende interactuando con un entorno dinámico, tomando decisiones y recibiendo recompensas o penalizaciones por sus acciones, lo que le permite aprender una estrategia óptima a través de la experiencia.

El aprendizaje automático, en todas sus formas, constituye el motor que impulsa la mayoría de las aplicaciones modernas de IA, permitiéndolas adaptarse y ser útiles en multitud de situaciones.

Conjunto de datos (*dataset*)

Un conjunto de datos, o *dataset*, es una colección estructurada de datos que se utiliza como base para el aprendizaje de un modelo de inteligencia artificial. En esencia, es el material de estudio del que dispone el modelo para entrenar, validar su rendimiento y, finalmente, ser puesto a prueba. Los *datasets* pueden ser públicos y estar ya creados, permitiendo a los investigadores experimentar con ellos, o pueden ser construidos a medida para un proyecto específico, como ha sido el caso en este trabajo.

En el ámbito de la seguridad en Android, existen varios *datasets* de referencia. Uno de los más conocidos es el Drebin *dataset* [28], una colección que contiene miles de aplicaciones Android, tanto maliciosas como benignas, junto con un gran número de características estáticas extraídas de ellas. Otro ejemplo relevante es el CIC-MalDroid 2020 [45], que ofrece una colección más moderna y diversa de *malware*. Para la fase de prototipado de este proyecto, se utilizó una versión del Drebin dataset para realizar las pruebas iniciales y validar la viabilidad del proyecto.

Extracción de características

Un modelo de aprendizaje automático no «entiende» el mundo como nosotros; no ve una imagen, un texto o un archivo, sino que únicamente opera con números. La extracción de características es el proceso fundamental mediante el cual, se intenta traducir la información del mundo real al lenguaje matemático que el modelo puede procesar. Consiste en identificar

y cuantificar las propiedades más representativas de los datos en bruto para convertirlas en un formato numérico y estructurado.

Esta «traducción» debe capturar la esencia del problema. Por ejemplo, si queremos analizar un programa en busca de *malware*, no le pasamos el archivo binario directamente al modelo. En su lugar, extraemos «pistas» medibles como el número de permisos que solicita, la presencia de ciertas funciones sospechosas o si intenta conectarse a direcciones de internet conocidas por ser maliciosas. Cada una de estas pistas es una característica que, en conjunto, forma una especie de perfil numérico de la aplicación que el modelo es capaz de aprender mucho mejor que el binario completo.

Sin embargo, no todas las pistas son igual de importantes. Una parte crucial del proceso es la selección de características, que consiste en filtrar el ruido y quedarse solo con la información que realmente aporta valor predictivo. Esto sería el equivalente a un detective que descarta las pistas falsas para centrarse en solo aquellas que le permiten resolver el caso. Este proceso de selección y filtrado culmina en la creación de un *dataset*, una tabla organizada donde cada fila es una muestra y cada columna es una de las características relevantes, lista para que el modelo pueda aprender de ella.

Preprocesamiento de datos

El preprocesamiento de datos es una etapa crucial en cualquier proyecto de aprendizaje automático la cual consiste en aplicar un conjunto técnicas al *dataset* justo antes del entrenamiento del modelo. Su objetivo principal es limpiar, transformar y adecuar los datos para mejorar la calidad de la información de entrada y, en consecuencia, el rendimiento, la eficiencia y la capacidad de generalización del modelo. Un modelo solo puede ser tan bueno como los datos con los que se entrena, el preprocesamiento es la etapa que intenta que esos datos estén en la mejor forma posible para facilitar el aprendizaje del modelo. A su vez, también es útil para permitir simplificar el modelo internamente y, asegurar que los datos que le llegan a este son siempre de una forma concreta, lo cual permite optimizar enormemente el rendimiento de este.

Las tareas de preprocesamiento son variadas y dependen de la naturaleza de los datos y del modelo que se pretende utilizar. Una de las más fundamentales es la conversión de estos a un formato numérico, ya que, al final, todos los algoritmos no son más que puras matemáticas que operan exclusivamente con números. Esto implica, por ejemplo, transformar texto en

una representaciones numéricas mediante técnicas como la tokenización y la creación de vocabularios, o convertir características categóricas (como «tipo de archivo») en valores numéricos mediante métodos como la discretización o el *one-hot encoding*. El resultado final suele ser una matriz numérica. [43]

Otras tareas comunes incluyen:

- **Imputación de valores nulos:** Muchos *datasets* contienen valores faltantes. Estos deben ser tratados, ya sea eliminando las muestras o características afectadas (si son pocas) o, más comúnmente, rellenando los huecos con un valor por defecto o incluso un estadístico como la media, la mediana o la moda de dicha característica.
- **Escalado de características:** Cuando las características numéricas tienen rangos muy diferentes (p. ej., una contiene valores entre 0 y 1, mientras que otra de 0 a 1 000 000), los algoritmos sensibles a la escala, como las máquinas de soporte vectorial (SVM) o los que usan el descenso de gradiente, pueden verse negativamente afectados. Técnicas como la normalización (escalar a un rango [0, 1]) o la estandarización (centrar en media 0 y desviación estándar 1) resuelven este problema.
- **Discretización:** Consiste en convertir características numéricas continuas en categorías o *bins* que cubren un rango concreto de todos los valores posibles.
- **Filtrado de características:** Aunque a menudo se considera parte de la extracción, la selección final de características puede realizarse también en esta fase, eliminando aquellas con baja varianza o baja correlación con la variable objetivo.

Embedder (Incrustador)

En el contexto del aprendizaje automático, un *embedding* es una técnica que permite convertir características discretas o de alta dimensionalidad (como palabras, permisos de una aplicación o identificadores únicos) en vectores de números reales densos [30]. La idea fundamental es mapear estos elementos a un espacio vectorial de tal manera que las relaciones semánticas o funcionales entre ellos se vean reflejadas en la geometría de dicho espacio. Es decir, los elementos que son similares en el mundo real acabarán «agrupados» o cerca unos de otros en este nuevo espacio de características.

Un *embedder* es el componente del modelo, generalmente una capa de la red neuronal, que se encarga de realizar esta transformación. Aprende

a generar estos vectores durante el propio proceso de entrenamiento del modelo principal. Por ejemplo, en lugar de decirle al modelo que el permiso `READ_CONTACTS` es simplemente el número «5», el *embedder* aprende a representarlo como un vector (p. ej., $[0.12, -0.45, 01.89, ...]$) que lo sitúa cerca de otros permisos de naturaleza similar, como `WRITE_CONTACTS`. Este proceso transforma datos complejos y no numéricos en una representación rica y llena de significado que el resto del modelo puede utilizar para encontrar patrones de manera mucho más eficaz.

Clasificador

Un clasificador es un modelo o algoritmo que se utiliza para asignar una categoría o etiqueta a un conjunto de datos, basándose en las características de los mismos. En el contexto de la IA, el proceso de clasificación implica entrenar un modelo para que aprenda a predecir la clase o categoría correcta de nuevas observaciones, basándose en ejemplos previos conocidos como datos de entrenamiento.

Clasificadores clásicos (ML)

Los clasificadores clásicos de aprendizaje automático son aquellos modelos que se establecieron como fundamentales en el campo antes de la popularización masiva de las redes neuronales profundas (*deep learning*). Estos algoritmos siguen siendo extremadamente útiles y relevantes, especialmente cuando se trabaja con datos estructurados o tabulares, o cuando la interpretabilidad del modelo es una prioridad, además, suelen requerir menos datos y recursos computacionales que los modelos de *deep learning*. A continuación se describen algunos de los más representativos:

- **Árbol de decisión (*Decision tree*):** Es uno de los modelos más intuitivos. Funciona creando una estructura similar a un diagrama de flujo, donde cada nodo interno representa una pregunta sobre una característica, cada rama representa la respuesta a esa pregunta, y cada nodo hoja representa una etiqueta de una de las clases. Para clasificar una nueva muestra, se la hace «descender» por el árbol desde la raíz, respondiendo a las preguntas en cada nodo hasta llegar a una hoja, la cual proporciona la predicción. Los árboles aprenden a realizar las mejores preguntas (o «cortes») para dividir los datos de la forma más pura posible en cada paso, basándose en métricas como la ganancia de información. [33]

- **k-Vecinos más cercanos (k-*Nearest Neighbors* o k-NN):** A diferencia de otros modelos, k-NN es un algoritmo «perezoso» (*lazy learner*), ya que no aprende una función como tal durante su entrenamiento. Simplemente, almacena todo el conjunto de datos de entrenamiento. Para clasificar una nueva muestra, busca las k muestras más similares (sus «vecinos más cercanos») en el conjunto de entrenamiento, basándose en una métrica de distancia como puede ser la distancia euclídea, la distancia de Manhattan o la distancia de Hamming. La clase de la nueva muestra se asigna por votación mayoritaria entre las clases de esos k vecinos. [36]

- **Máquinas de soporte vectorial (*Support Vector Machines* o SVM):** El objetivo de un SVM es encontrar el hiperplano óptimo que separe las distintas clases que se desean predecir en el espacio de características del conjunto de datos con el mayor margen posible. Dicho «margen» es la distancia entre el hiperplano de decisión y los puntos de datos más cercanos de cada clase, llamados «vectores de soporte». Al maximizar este margen, el modelo logra una mejor capacidad de generalización. Para problemas que no son linealmente separables, los SVM pueden utilizar lo que se conoce como el *kernel trick*, el cual consiste en mapear los datos a un espacio de mayor dimensión donde sí se pueda encontrar un hiperplano que separe las clases correctamente [35].

- **Regresión logística:** A pesar de su nombre, la regresión logística es un modelo de clasificación lineal estadístico bastante usado y que funciona sorprendentemente bien pese a su simplicidad. Se basa en estimar la probabilidad de que una muestra pertenezca a una clase determinada. Lo hace aplicando la función logística (o sigmoide) a una combinación lineal de las características de entrada. El resultado es un valor entre 0 y 1, que puede interpretarse como una probabilidad. Un umbral de decisión (típicamente 0.5) se utiliza para asignar la muestra a una de las dos clases.

Ensembles

En el aprendizaje automático, a menudo se sigue una filosofía similar a la de un comité de expertos: en lugar de confiar en la opinión de un único especialista, se combinan las decisiones de varios para obtener un veredicto final más robusto y fiable. Los ensembles o conjuntos de modelos se basan precisamente en esta idea. La lógica subyacente es que los errores o sesgos

individuales de cada modelo tienden a ser compensados por los aciertos de los demás, resultando en una predicción colectiva más sólida que la de cualquiera de sus miembros por separado.

Existen dos estrategias principales para crear estos «comités» de modelos. El *bagging* [31] consiste en entrenar a varios modelos en paralelo, cada uno con una muestra ligeramente diferente de los datos, y luego agregar sus predicciones (por ejemplo, mediante una votación). Uno de los modelos más usados que siguen este enfoque es el RandomForest [32], el cual se basa en aplicar este método de entrenamiento a varios árboles de decisión, debida su naturaleza «inestable» (pequeños cambios en su entrenamiento cambian drásticamente el árbol final) esto funciona muy bien con ellos, porque los diferentes árboles que se obtienen son muy distintos entre sí, lo cual permite que algunos sean mejores en ciertos aspectos que otros, complementándose y mejorando su rendimiento al ser evaluados en conjunto. Por otro lado, el *boosting* [38] es un proceso secuencial: se entrena un primer modelo y, a continuación, se entrena un segundo modelo que se centra específicamente en corregir los errores del primero, y así sucesivamente. Cada nuevo modelo se especializa aquellos ejemplos difíciles que el modelo anterior no supo clasificar correctamente, generando así una cadena de expertos cada vez mejor entrenados. El mejor modelo que representa esta técnica de entrenamiento es el XGBoost (eXtreme Gradient Boosting) [34], el cual, se basa nuevamente en entrenar árboles de decisión pero de manera secuencial y siguiendo el proceso descrito anteriormente.

Neurona artificial y perceptrón multicapa (MLP)

La neurona artificial [46] es la piedra fundamental que sirve como predecesor a las redes neuronales. Su diseño está inspirado en una neurona biológica, la cual recibe señales a través de sus dendritas, las procesa en el soma y emite una señal a través de su axón si el estímulo acumulado supera un cierto umbral. De forma análoga, una neurona artificial recibe una o más entradas numéricas $(x_1, x_2, ..., x_n)$, a cada una de las cuales se le asigna un peso ajustable $(w_1, w_2, ..., w_n)$. Estos pesos determinan la importancia de cada entrada. La neurona calcula la suma ponderada de sus entradas, a la que se le añade un término de sesgo (*bias*, b). Este resultado agregado pasa a través de una función de activación no lineal, que determina la salida final de la neurona.

El primer modelo formal de una neurona artificial fue el perceptrón [51], desarrollado por Frank Rosenblatt en 1957. Un único perceptrón podía aprender a resolver problemas de clasificación binaria que fueran linealmente

separables. Sin embargo, se demostró que un perceptrón simple era incapaz de aprender funciones no lineales tan básicas como la puerta XOR [48], lo que limitó gravemente su aplicabilidad y condujo a un período de menor interés en la investigación de redes neuronales.

La solución a esta limitación llegó con el desarrollo del perceptrón multicapa (*Multilayer Perceptron* o MLP). Un MLP consiste en apilar las neuronas artificiales en múltiples capas: una capa de entrada (*input layer*), que recibe los datos iniciales; una o más capas intermedias u ocultas (*hidden layers*), que realizan la mayor parte del procesamiento y permiten aprender patrones complejos; y una capa de salida (*output layer*), que produce el resultado final (p. ej., una clasificación o un valor de regresión). Al interconectar las neuronas de esta manera, donde la salida de las neuronas de una capa se convierte en la entrada de las neuronas de la siguiente, la red en su conjunto es capaz de aprender representaciones jerárquicas y aproximar cualquier función continua, superando las limitaciones del perceptrón simple. El MLP es el modelo base de cualquier red neuronal actual y uno de los avances más importantes en el campo.

Redes Neuronales Profundas (*Deep Learning*)

El aprendizaje profundo o *deep learning* es un subcampo del aprendizaje automático que ha revolucionado la inteligencia artificial en la última década. Su seña de identidad es el uso de redes neuronales profundas, que no son más que redes neuronales con una arquitectura que incluye múltiples capas ocultas apiladas entre la capa de entrada y la de salida. Esta «profundidad» es lo que les otorga su extraordinario poder, ya que les permite aprender representaciones de los datos de forma jerárquica y con distintos niveles de abstracción.

La idea es que cada capa aprende a reconocer patrones basándose en la salida de la capa anterior. Las primeras capas aprenden a detectar características muy simples y de bajo nivel. A medida que la información fluye a través de la red, las capas posteriores combinan estas características simples para construir conceptos cada vez más complejos y abstractos. Por ejemplo, en el análisis de imágenes, las primeras capas podrían detectar bordes y colores, las siguientes podrían combinarlos para reconocer formas como ojos o narices, y las capas finales podrían combinar toda esta información para identificar una cara.

El éxito arrollador del *deep learning* no se debe únicamente a la elegancia de su arquitectura, sino también a la unión de dos factores externos: la

disponibilidad de enormes volúmenes de datos (*big data*) para entrenar estos modelos tan complejos, y el desarrollo de *hardware* de computación paralela, principalmente las Unidades de Procesamiento Gráfico (GPU), que proporcionan la potencia de cálculo necesaria.

Métricas de evaluación de un modelo

Para saber si un modelo de clasificación funciona bien, no basta con mirar su porcentaje de aciertos. Necesitamos un análisis más profundo que nos revele cómo acierta y cómo se equivoca. Esto es especialmente crítico cuando las clases están desbalanceadas, es decir, cuando tenemos muchos más ejemplos de una categoría que de otra. Para ello, primero se definen los cuatro posibles resultados que puede tomar una predicción:

- **Verdadero positivo (TP - *True Positive*):** El modelo predice correctamente que una muestra es positiva (p. ej., predice «*malware*» y la aplicación es *malware*).
- **Falso positivo (FP - *False Positive*):** El modelo predice incorrectamente que una muestra es positiva (p. ej., predice «*malware*» y la aplicación es benigna). También se conoce como **error de tipo I**.
- **Verdadero negativo (TN - *True Negative*):** El modelo predice correctamente que una muestra es negativa (p. ej., predice «benigno» y la aplicación es benigna).
- **Falso negativo (FN - *False Negative*):** El modelo predice incorrectamente que una muestra es negativa (p. ej., predice «benigno» y la aplicación es *malware*). También se conoce como **error de tipo II**.

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

- **Exactitud (*Accuracy*):** Es la métrica más intuitiva, representa el porcentaje total de aciertos. Aunque útil, puede ser muy engañosa en escenarios con clases desbalanceadas. Un 99 % de exactitud suena genial, pero si solo el 1 % de las muestras son *malware*, un modelo que siempre predice «benigno» lograría esa cifra sin aportar ningún valor.

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

- **Precisión (*Precision*):** Responde a la pregunta: de todas las veces que el modelo predijo «*malware*», ¿cuántas veces acertó? Una alta precisión es vital cuando el coste de un falso positivo es alto (por ejemplo, bloquear una aplicación legítima).

$$Precision = \frac{TP}{TP + FP}$$

- **Sensibilidad (*Recall* o *Sensitivity*):** Responde a una pregunta diferente pero igualmente crucial: de todas los ejemplos de *malware* real que existían, ¿qué porcentaje fue capaz de detectar el modelo? Es la métrica más importante cuando el coste de un falso negativo es alto, como es en este caso, equivaldría a no detectar una aplicación maliciosa que sí lo era.

$$Recall = \frac{TP}{TP + FN}$$

- **Especificidad (*Specificity*):** Es el equivalente a la sensibilidad, pero para la clase negativa. De todas las aplicaciones benignas, ¿cuántas fueron correctamente identificadas como tal?

$$Specicity = \frac{TN}{TN + FP}$$

- **Puntuación F1 (F_1-*Score*):** Dado que la precisión y la sensibilidad suelen estar siempre en conflicto (mejorar una a menudo empeora la otra), la puntuación F1 ofrece un equilibrio entre ambas. Es la media armónica de las dos, lo que significa que penaliza a los modelos que tienen un desequilibrio extremo entre ellas. Es una de las métricas de referencia para evaluar clasificadores más usada.

$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

Más allá de las métricas individuales, existen dos curvas de evaluación muy utilizadas que nos ofrecen una visión más detallada del comportamiento de un modelo. Nos permiten ver cómo se comporta un clasificador no en un único punto, sino a través de todos los posibles umbrales de decisión. Un umbral es, sencillamente, el nivel de «confianza» que el modelo necesita para clasificar una muestra como positiva. Al variar este umbral (desde ser muy estricto a muy permisivo), obtenemos las siguientes curvas.

- **Curva ROC (*Receiver Operating Characteristic*):** Es una de las dos curvas clásicas usadas para representar el rendimiento de un clasificador binario. Su principal objetivo es mostrar el compromiso entre encontrar los positivos reales y evitar las falsas alarmas. La curva se obtiene de enfrentar las siguientes métricas:

  - **Eje Y (Vertical):** La **Tasa de Verdaderos Positivos (TPR)**, que es exactamente lo mismo que la sensibilidad ($TPR = TP/(TP+FN)$).
  - **Eje X (Horizontal):** La **Tasa de Falsos Positivos (FPR)**, similar a la sensibilidad pero para los falsos positivos ($FPR = FP/(TN+FP)$).

  Interpretación:

  - **El modelo perfecto:** Una curva que sube verticalmente hasta el punto $(0, 1)$ y luego se desplaza horizontalmente. Este punto en la esquina superior izquierda representa un 100 % de sensibilidad (encuentra todos los positivos) con un 0 % de falsos positivos (cero falsas alarmas). Cuanto más se acerque la curva a esta esquina, mejor será el modelo.
  - **La línea de azar:** Una línea diagonal de $(0, 0)$ a $(1, 1)$ representa un modelo sin ninguna capacidad de discriminación, como lanzar una moneda al aire. Cualquier curva por debajo de esta línea indica que el modelo es peor que el azar.
  - **A medida que la curva crece:** Un crecimiento rápido hacia arriba significa que el modelo está ganando mucha sensibilidad sin un coste alto de falsos positivos, lo cual es ideal. Si la curva se desplaza demasiado hacia la derecha, significa que para encontrar más positivos, el modelo está cometiendo demasiadas falsas alarmas.

- **Curva PR (Precisión-Sensibilidad o *Precision-Recall*):** Aunque la curva ROC es muy útil, puede ser engañosa en problemas con un gran desbalance de clases (por ejemplo, cuando la clase positiva es muy rara). En estos casos, la curva PR suele ser más informativa. Cómo su nombre indica, la curva se obtiene de enfrentar la precisión (eje Y) con la sensibilidad (eje X).

  Interpretación:

- **El modelo perfecto:** La curva ideal se sitúa en la esquina superior derecha, en el punto $(1, 1)$. Esto significa que el modelo mantiene una precisión del 100 % incluso cuando logra encontrar a todos los positivos reales (sensibilidad del 100 %).
- **La línea de base:** A diferencia de la curva ROC, la línea de base de una curva PR no es la diagonal. Es una línea horizontal que corresponde a la proporción de ejemplos positivos en el *dataset*. Por ejemplo, si solo el 2 % de los datos son positivos, un modelo aleatorio tendrá una precisión en torno al 2 % y esta sería su línea de base. Un modelo solo es útil si su curva PR está significativamente por encima de esta línea de base.
- **A medida que la curva crece (hacia la derecha):** Lo que buscamos es que la curva se mantenga lo más alta posible. Una caída brusca en la precisión a medida que aumenta la sensibilidad nos dice que el modelo, para encontrar más positivos, empieza a cometer muchos errores (falsos positivos), «contaminando» sus predicciones.

- **AUC (*Area Under the Curve* o Área Bajo la Curva):** Ambas de las curvas anteriores son muy útiles si se desea obtener una visión más global del rendimiento del modelo, pero, en muchos casos, es conveniente poder condensar toda la información que estas curvas nos proveen en un único número, para esto existe el AUC. Este no es más que, literalmente, el área debajo de cada una de las curvas y provee un valor numérico que resume el rendimiento general del clasificador a través de todos los umbrales.

  La interpretación de esta medida varía en función de la curva que estemos intentado interpretar pero, su significado es equiparable al de las curvas en sí pero condensado en un único valor. Suelen darse en tanto por uno o como probabilidad.

  - **ROC-AUC *(Area Under the ROC Curve)***: Resume cuánto de bueno es el modelo a al ahora de diferenciar entre las clases. Si el valor es cercano a uno (100 %) indica que estamos ante un clasificador excelente. Significa que para un ejemplo positivo y negativo elegidos al azar, hay una probabilidad muy alta (cercana al 100 %) de que el modelo asigne una puntuación más alta al positivo. En cambio, si el valor es cercano a 0.5, implica que el modelo no es capaz de distinguir bien entre las clases y se puede equiparar a lanzar una moneda al aire.

- **PR-AUC (*Area Under the Precision-Recall Curve*):** Representa la precisión media del modelo a lo largo de todos los umbrales de sensibilidad. Si el valor es cercano a 1, indica que la precisión se mantiene cerca del 100 % a medida que la sensibilidad aumenta. Por otro lado, si el valor es cercano al de la línea de base, esta métrica no aporta mucha más que, el hecho de que la precisión media es por estadística la proporción de ejemplos positivos en el *dataset* y escala de esa manera a medida que aumenta la sensibilidad.

Entrenamiento de un modelo

El entrenamiento de un modelo de IA es el proceso mediante el cual este aprende a realizar una tarea a partir de los datos. Aunque el proceso varía en función del modelo, la idea general es siempre la misma: se le presentan al modelo unos datos de entrada, este genera una predicción y se compara dicha predicción con el resultado correcto para calcular un error. Este error se utiliza posteriormente para ajustar los parámetros internos del modelo, de modo que la próxima vez su predicción sea un poco mejor. Este ciclo se repite durante un número de iteraciones dado o hasta que se cumpla una condición de parada. Esta condición suele ser. que se haya sobrepasado un umbral de rendimiento o convergencia o, que el modelo no mejore de forma significativa tras cierto número de iteraciones.

Los modelos de aprendizaje automático clásicos, como Random Forest, suelen tener un proceso de entrenamiento más directo y estandarizado. En cambio, los modelos de *deep learning*, como las redes neuronales, utilizan un mecanismo más complejo basado en el algoritmo de *backpropagation* para ajustar sus parámetros.

Para asegurar que el modelo se evalúa de forma justa y, para evitar que este simplemente «memorice» los datos, se utilizan técnicas como la división del *dataset* en conjuntos de entrenamiento, validación y prueba (*train-validation-test split*). Una técnica aún más robusta es la validación cruzada (*cross-validation*), que consiste en dividir el *dataset* en k partes o *folds*. El modelo se entrena k veces, usando en cada ocasión una de las partes no usadas anteriormente para la validación y el resto para el entrenamiento. Esto garantiza que la evaluación del rendimiento sea mucho más fiable y menos dependiente de la suerte en el reparto inicial de los datos.

Interpretabilidad de un modelo

La interpretabilidad de un modelo se refiere a nuestra capacidad para entender y explicar cómo llega a sus conclusiones. Muchos modelos clásicos de aprendizaje automático, como un árbol de decisión, son relativamente transparentes: podemos seguir su lógica interna para ver por qué han tomado una decisión. Sin embargo, a medida que los modelos se vuelven más complejos, como los ensembles o, especialmente, las redes neuronales profundas, su funcionamiento interno se convierte en una especie de «caja negra». Todo se reduce a millones de operaciones matemáticas y pesos ajustados que, aunque funcionan, no nos dicen explícitamente por qué funcionan.

Aquí es donde nace el campo de la interpretabilidad de modelos (*XAI - Explainable AI*). La confianza en un sistema de IA, sobre todo en campos críticos como la medicina o la ciberseguridad, depende cada vez más de que sus decisiones sean explicables. Por ejemplo, un modelo que predice la probabilidad de que un paciente tenga una enfermedad con un 99 % de acierto es impresionante, pero poco fiable para un médico si no puede explicar en qué se basa. Si, por el contrario, el modelo puede señalar qué factores (características) han influido más en su predicción para ese paciente, gana una enorme credibilidad.

Existen principalmente dos enfoques de interpretabilidad. El análisis local se centra en explicar una predicción individual (p. ej., ¿por qué esta aplicación concreta fue clasificada como *malware*?). El análisis global, en cambio, busca dar una visión más general del comportamiento del modelo, explicando qué características son las más importantes para él en promedio.

UMAP (*Uniform Manifold Approximation and Projection*)

UMAP (*Uniform Manifold Approximation and Projection*) [47] es un algoritmo moderno de reducción de dimensionalidad basado en principios matemáticos de la topología y el aprendizaje de variedades (*manifold learning*). La idea fundamental de UMAP es que la mayoría de los conjuntos de datos de alta dimensión en realidad viven en una «variedad» o superficie de una dimensionalidad mucho más baja que está «curvada» dentro de ese espacio. La misión de UMAP es encontrar esa variedad y «desplegarla» en un espacio de pocas dimensiones (como una hoja de papel 2D) perdiendo la menor cantidad de información estructural posible, facilitando así su visualización e interpretabilidad.

Para lograrlo, su algoritmo se basa en dos etapas. Primero, construye una representación topológica de los datos en el espacio de alta dimensión. Para ello, crea un grafo ponderado donde los pesos de las aristas se calculan asumiendo que la distancia entre los puntos varía localmente a lo largo de la variedad. En la segunda etapa, busca una «incrustación» (*embedding*) de baja dimensión de ese grafo, es decir, una disposición de los puntos en 2D o 3D, que tenga una estructura topológica lo más parecida posible. Para medir esta «similitud» entre el grafo de alta y baja dimensión, utiliza una función de coste basada en la entropía cruzada, que optimiza para encontrar la mejor proyección posible.

Función de pérdida (*Loss function*)

Para que un modelo pueda aprender, necesita una forma de saber si sus predicciones son correctas o no. La función de pérdida (*loss function*), o función de coste, cumple exactamente este papel, es una fórmula matemática que estima el error que el modelo está cometiendo en sus predicciones. Después de cada predicción que este hace, la función de pérdida la compara con la respuesta correcta y calcula un número que representa cuánto de «equivocado» estaba el modelo. Si el número es alto, el error es grande; si es cercano a cero, indica que la predicción fue muy buena.

Existen diferentes tipos de funciones de pérdida y, su elección depende de la tarea específica a realizar por el modelo. Por ejemplo, para problemas de regresión se suele utilizar el error cuadrático medio (*mean squared error*), mientras que para problemas de clasificación se emplea la entropía cruzada (*cross-entropy*).

A su vez, durante el entrenamiento, es fundamental monitorizar dos métricas de pérdida:

- **Pérdida de entrenamiento (*training loss*):** Es el valor de la función de pérdida calculado sobre el conjunto de datos de entrenamiento. Este valor es el que el algoritmo de optimización intenta minimizar directamente ajustando los parámetros del modelo. Una disminución constante de la pérdida de entrenamiento indica que el modelo está aprendiendo los patrones presentes en los datos de entrenamiento.

- **Pérdida de validación (*validation loss*):** Es el valor de la función de pérdida calculado sobre un conjunto de datos separado, llamado conjunto de validación, que el modelo no utiliza para aprender. Esta

métrica es un indicador mucho más fiable de la capacidad de generalización del modelo, es decir, de su rendimiento en datos nuevos y no vistos. Si la pérdida de entrenamiento sigue disminuyendo pero la de validación comienza a aumentar, es una señal clara de sobreajuste (*overfitting*).

Sobreajuste (*overfitting*) y subajuste (*underfitting*)

El sobreajuste (*overfitting*) y el subajuste (*underfitting*) son posiblemente los dos problemas más comunes que pueden surgir durante el entrenamiento de modelos de aprendizaje automático, representando extremos opuestos del rendimiento de un modelo. Ambos fenómenos se refieren a la incapacidad del modelo para generalizar adecuadamente a partir de los datos de entrenamiento y, por lo tanto, para realizar predicciones precisas sobre datos nuevos y no vistos.

En concreto, el sobreajuste ocurre cuando un modelo aprende los datos de entrenamiento «demasiado bien», capturando no solo los patrones subyacentes, sino también el ruido y las fluctuaciones aleatorias específicas del conjunto de datos. Dicho de otra forma, el modelo se vuelve excesivamente complejo y «memoriza» los ejemplos de entrenamiento en lugar de aprender a generalizar. La manera en la que esto se manifesta durante el entrenamiento es mediante un rendimiento excelente en los datos de entrenamiento (*training loss* muy baja y que cae constantemente) pero un rendimiento pobre en los datos de validación (*validation loss* alta y que se estanca o empieza a subir tras un punto en el entrenamiento).

Por otro lado, el subajuste sucede cuando un modelo es demasiado simple para capturar la complejidad y los patrones inherentes en los datos. En este caso, el modelo no tiene la capacidad suficiente para aprender relaciones significativas, lo que resulta en un mal rendimiento tanto en el conjunto de entrenamiento como en el de validación. Un modelo subajustado no aprende eficazmente, manifestándose en una *loss* alta para ambos conjuntos de datos.

Descenso de gradiente, aprendizaje y *backpropagation*

Tres conceptos intrínsecamente ligados, los cuales forman el núcleo del mecanismo que permite a las redes «aprender» de los datos. El proceso de aprendizaje consiste en ajustar iterativamente los parámetros del modelo (pesos y sesgos) para minimizar una función de pérdida.

El descenso de gradiente [27] (*gradient descent*) es el algoritmo de optimización que permite realizar este ajuste. La función de pérdida puede

visualizarse como una superficie con valles y colinas en un espacio multidimensional, donde cada punto de la superficie representa un valor de pérdida para una configuración específica de los pesos del modelo. El objetivo es encontrar el punto más bajo de esta superficie (el mínimo global de la pérdida). El «gradiente» es un vector que apunta en la dirección del máximo crecimiento de la función en un punto dado. Por lo tanto, para «descender» y minimizar la pérdida, el algoritmo da pequeños pasos en la dirección opuesta al gradiente. El tamaño de estos pasos está controlado por un hiperparámetro llamado tasa de aprendizaje (*learning rate*). Una tasa adecuada es crucial: si es demasiado pequeña, el aprendizaje será muy lento; si es demasiado grande, el algoritmo podría sobrepasar el mínimo y no converger nunca.

Para ahora poder realizar dicho ajuste de los pesos es necesario usar un famoso algoritmo conocido como *backpropagation* [52]. Tras realizar una pasada hacia adelante (*forward pass*) para obtener una predicción y calcular la función de pérdida, *backpropagation* aplica la regla de la cadena del cálculo para propagar el error hacia atrás, desde la capa de salida hasta la capa de entrada. De esta manera, calcula la contribución de cada peso y sesgo al error total, es decir, el gradiente de la pérdida con respecto a cada parámetro.

El proceso de aprendizaje completo se resume a continuación:

- **Pasada hacia adelante (*forward pass*):** Se introducen los datos en la red y se propagan a través de las capas para generar una predicción.
- **Cálculo de la pérdida:** Se compara la predicción con el valor real usando la función de pérdida para cuantificar el error.
- **Pasada hacia atrás (*backward pass* / *backpropagation*):** Se calcula el gradiente de la función de pérdida con respecto a cada parámetro de la red.
- **Actualización de parámetros:** Se utiliza el descenso de gradiente para ajustar los pesos y sesgos en la dirección opuesta al gradiente, reduciendo así la pérdida.

Este ciclo se repite de forma iterativa con diferentes lotes (*batches*) de datos, permitiendo que la red refine gradualmente sus parámetros y mejore su rendimiento.

Tokenización

Los modelos de inteligencia artificial, en su más puro estado, son modelos matemáticos complejos los cuales consiguen aprender patrones de sus datos de entrada y permiten realizar posteriormente predicciones en base a dichos patrones.

El problema fundamental que esto presenta a la hora de procesar diferentes tipos de datos es principalmente que los modelos matemáticos se basan en el uso de números y la búsqueda de patrones en datos numéricos. Esto implica que si usamos datos que no sean numéricos, como puede ser el texto, caemos en un grave problema puesto que no podemos simplemente aplicar el modelo a este tipo de dato directamente puesto no está preparado para funcionar con el.

Una solución a este problema es emplear el proceso de tokenización, el cual consiste en romper el texto que se obtiene como entrada en unidades más sencillas llamadas tokens que el modelo pueda llegar a entender posteriormente. Este proceso puede realizarse de diferentes maneras, ya sea partiendo cada palabra en un token individual, partiendo las palabras en subpalabras o incluso procesando cada carácter de manera independiente.

El proceso de tokenización parece bastante trivial a primera vista pero presenta muchos retos como pueden ser los siguientes. Procesar diferentes idiomas se vuelve rápidamente un problema en función del método que se utilice para obtener los tokens, diferentes idiomas permiten contracciones o expresiones especiales las cuales pueden ser interpretadas de diferentes maneras y han de poder ser divididas en tokens que mantengan dicho significado. También es posible que un texto presente errores gramaticales, símbolos especiales o jerga específica de un campo que ha de ser entendida y tokenizada correctamente para mantener su significado.

Vocabulario

Para que un modelo pueda entender los tokens que se obtienen en el paso de la tokenización, es necesario convertir dichos tokens en números de alguna forma. Esto puede realizarse de forma sencilla mediante el uso de un diccionario o vocabulario para el modelo.

El vocabulario del modelo no es más que la colección de todos los tokens que se pretende que el modelo pueda comprender asociados a un valor numérico que siempre será el mismo. De esta forma, el texto de entrada puede ser dividido en tokens, los cuales, a su vez, pueden ser convertidos

siempre en el mismo valor numérico mediante el uso de un diccionario. Permitiendo así que el modelo procese texto y pueda encontrar patrones en este.

De forma similar al proceso de la tokenización, la creación y selección de un vocabulario presenta muchos problemas que pueden no ser tan evidentes a primera vista, algunos de ellos son, la extensión del diccionario y el número de tokens que son necesarios en este para poder procesar diferentes idiomas o, la necesidad de poder incluir valores de control que puedan ser usados en casos en los que un token no exista o se le quiera dar una información especial a ciertos tokens en algunas circunstancias.

3.2. Conceptos de ciberseguridad

Antes de poder construir una buena defensa, es imprescindible conocer a fondo la amenaza que se pretende combatir. Este capítulo se dedica a explorar el mundo del software malicioso o *malware*, el principal adversario que nuestro modelo de inteligencia artificial está diseñado para combatir. Se comenzará por definir qué es el *malware* y se describirán sus diferentes manifestaciones, desde los virus clásicos hasta el *ransomware* moderno. A continuación, se explicarán las técnicas que se utilizan para inspeccionar programas sospechosos, sentando así las bases para comprender de dónde extraerá nuestro modelo la información para tomar sus decisiones.

Malware

El término *malware*, (proveniente de *malicious software*) o software malicioso, se refiere a cualquier programa, código o script diseñado con el propósito explícito de causar daño, comprometer la seguridad, o realizar actividades no autorizadas en un sistema informático, una red o un dispositivo. Coloquialmente, se utiliza el término *malware* para referirse a una amplia gama de diferentes subcategorías de programas que puedan considerarse dañinos, cada una con características, formas de ataque y objetivos específicos, pero todas compartiendo la intención de perjudicar al usuario, robar información, o tomar el control de aquellos sistemas afectados.

Una clasificación bastante común del *malware* es la siguiente:

Virus

Un virus es considerado el comodín de los programas maliciosos, al menos desde un punto de vista coloquial, ambas palabras son intercambiable.

En cambio, desde un punto de vista técnico, la definición de un virus informático es equiparable a su equivalente biológico. Un virus es un tipo de *malware* que requiere de un huésped para poder realizar su función, es decir, suele ir incrustado o adjunto a otros archivos, generalmente, ejecutables o documentos los cuales, una vez abiertos, permiten al virus infectar el sistema y realizar acciones no deseadas. Una característica importante de los virus, es el hecho de que pueden auto replicarse, es decir, una vez infectada una máquina, pueden emplear diferentes métodos y técnicas para propagarse a otras, comúnmente, estas suelen ser mediante el uso de la red o mediante el uso de medios físicos extraibles. Los virus suelen ser detectados mediante firmas específicas, aunque las técnicas de ofuscación y polimorfismo pueden dificultar su identificación.

Gusano (*worm*)

Un gusano es un tipo de *malware* diseñado para propagarse automáticamente a través de las redes informáticas, explotando las vulnerabilidades de los diferentes sistemas o utilizando técnicas de ingeniería social para engañar a los usuarios. A diferencia de los virus, los gusanos no necesitan de un huésped al cual adjuntarse para poder desempeñar su función y para auto replicarse, puesto que pueden propagarse de manera independiente. Su principal objetivo es infectar tantos dispositivos como sea posible, lo que puede resultar en la saturación de las redes, la degradación del rendimiento del sistema, o la creación de puertas traseras para permitir el paso a otros tipos de *malware*. Los gusanos son particularmente peligrosos en entornos corporativos, donde pueden propagarse rápidamente a través de las redes internas de una empresa o una organización.

Troyano (*trojan*)

Un troyano es un tipo de *malware* que se hace pasar por un *software* legítimo o útil para engañar a los usuarios y lograr su ejecución. Una vez instalado, un troyano permite a un atacante acceder o controlar el sistema infectado de manera remota, sin el conocimiento del usuario. Los troyanos no se replican por sí mismos, pero pueden realizar gran variedad de acciones maliciosas, principalmente, robar información confidencial, instalar otros tipos de *malware*, o convertir el sistema en parte de una *botnet*. Su nombre proviene del mito del Caballo de Troya, perteneciente a la mitología griega, ya que, al igual que sucede en la historia, el troyano parece inofensivo en un principio pero oculta una gran amenaza en su interior.

Ransomware

El *ransomware* es un tipo de *malware* diseñado para cifrar los archivos del usuario o bloquear el acceso al sistema, exigiendo un rescate, *ransom*, generalmente en criptomonedas, a cambio de restaurar el acceso. Este tipo de *malware* ha ganado mucha popularidad en los últimos años debido a su impacto devastador en individuos, empresas e incluso instituciones gubernamentales. El *ransomware* suele propagarse a través de correos electrónicos de *phishing*, descargas maliciosas o *exploits* de vulnerabilidades. Una vez este es activado, el comportamiento más típico consiste en mostrar al usuario un mensaje (*ransom note*) con las instrucciones para pagar el rescate. Es importante destacar que, aunque se pague la tasa correspondiente al rescate dentro del tiempo establecido, no hay ninguna garantía de que los atacantes cumplan con su promesa de desbloquear los archivos. Es por ello, por lo que, este tipo de *malware* es extremadamente peligroso puesto que juega con el factor de perder un recurso muy preciado, como puede ser la información, además de utilizar la desesperación de los usuarios en su contra.

Spyware o *Info stealer*

El *spyware* es un tipo de *malware* diseñado para recopilar información del usuario sin su consentimiento. Esta información puede incluir contraseñas, datos bancarios, historiales de navegación, métodos de entrada (como pueden ser las pulsaciones de las teclas de un teclado) o cualquier otro dato sensible que pueda posteriormente ser vendido o utilizado en contra de los usuarios. El *spyware* suele operar de manera sigilosa, sin mostrar signos evidentes de su presencia, lo que dificulta su detección.

En términos generales, existen dos variantes de *spyware*, la primera de ellas se instala en el sistema de forma permanente y siempre está activa, monitorizando la actividad del usuario de manera constante, mandando cualquier información sensible que este use, vea o teclee a un servidor externo para que los atacantes la puedan utilizar o vender, la única ventaja que presenta esta variante es que deja rastro y es más sencilla de detectar. Por otro lado, la segunda variante simplemente se ejecuta una vez, recopila toda la información que pueda y luego se borra a si misma para no dejar ni el más mínimo rastro de su ejecución. Para muchos, esta segunda variante es considerada incluso más peligrosa que la primera puesto que la víctima puede tardar semanas, meses o incluso años en darse cuenta de que su información a sido comprometida.

Además de robar información, algunos tipos de *spyware* pueden modificar la configuración del sistema, instalando *software* adicional o redirigiendo

el tráfico de red, generalmente, para evitar su detección. Este tipo de *malware* es comúnmente distribuido a través de descargas no autorizadas, correos electrónicos de *phishing*, o *software* gratuito (*freeware*) que incluye componentes ocultos.

Adware

El *adware* es un tipo de *software* que muestra publicidad no deseada, a menudo de manera intrusiva, en el dispositivo del usuario. Aunque no siempre es malicioso, el *adware* puede ser molesto y afectar negativamente a la experiencia del usuario. En algunos casos, el *adware* incluye funcionalidades adicionales para rastrear el comportamiento del usuario y mostrar anuncios personalizados, lo que puede considerarse una violación de la privacidad. El *adware* suele distribuirse junto con *software* gratuito, y los usuarios pueden instalarlo sin darse cuenta al aceptar los términos y condiciones sin leerlos detenidamente.

PUP (*Potentially Unwanted Program* o Programa Potencialmente no Deseado)

Un PUP (*Potentially Unwanted Program*, por sus siglas en inglés) es un tipo de *software* que, aunque no es necesariamente malicioso, puede ser considerado no deseado por el usuario. Los PUPs incluyen aplicaciones como barras de herramientas (*toolbars*), optimizadores de sistema, o *software* de publicidad que se instalan sin el consentimiento explícito del usuario. Aunque no siempre son dañinos, los PUPs pueden ralentizar el sistema, mostrar anuncios no deseados, o recopilar información acerca del usuario. Muchos antivirus y soluciones de seguridad clasifican los PUPs como una categoría separada de *malware*, ya que su impacto puede variar desde simplemente molesto hasta potencialmente peligroso.

Rootkit

Un *rootkit* es un conjunto de herramientas o *software* diseñado para otorgar a un atacante acceso privilegiado y persistente a un sistema, mientras oculta su presencia tanto del usuario como de cualquier *software* de seguridad que pueda estar presente en el sistema. Los *rootkits* suelen operar a nivel de *kernel* (conocido como Ring 0 en muchos casos) o del sistema operativo, lo que les permite manipular cualquier funcionalidad del sistema, incluso aquellas de las que el usuario posiblemente desconoce puesto que están ocultas por el sistema operativo para facilitar su uso. Esto permite a los *rootkits* ser uno de los *malware* más peligrosos debido a que pueden evadir

la gran mayoría de soluciones de seguridad y antivirus puesto que operan con los mismos privilegios que estos o incluso más altos. Una vez instalado, un *rootkit* puede ser utilizado para instalar otros tipos de *malware*, robar información, o convertir el sistema en parte de una *botnet*. Debido a su capacidad para ocultarse, los *rootkits* son particularmente difíciles de detectar y eliminar, y a menudo requieren herramientas especializadas o en muchos casos, la reinstalación completa del sistema para poder deshacerse de ellos.

Botnet

Una *botnet* es una red de dispositivos infectados (llamados *bots* o *zombies*) controlados de manera remota por un atacante, conocido como *botmaster*. Los dispositivos infectados pueden incluir computadoras, servidores, dispositivos IoT, y otros equipos conectados a internet. Las *botnets* son utilizadas para realizar una variedad de tareas maliciosas, como pueden ser, ataques de denegación de servicio distribuido (DDoS), envío masivo de correos no deseados (*spam*), minería de criptomonedas, o robo de información masivo. Los dispositivos infectados suelen ser controlados a través de un servidor externo, y los usuarios generalmente no son conscientes de que su dispositivo forma parte de una *botnet*. La creación y gestión de *botnets* es una de las actividades que más dinero genera para los ciberdelincuentes, ya que les permite llevar a cabo ataques a gran escala con un impacto muy significativo.

Análisis estático

Técnica de detección de *malware* que se realiza sin la necesidad de ejecutar el programa en cuestión. Este método se basa en la obtención, inspección y evaluación de las características que se pueden extraer de un archivo binario, tales como su estructura, código fuente (si está disponible), indicios de obfuscación u otras técnicas de ocultación, cadenas de texto incrustadas en este, firmas digitales, huella digital *hash o signature* del archivo, secuencias de bytes concretas, cabeceras del programa, metadatos incrustados, desensamblado del ejecutable y otras propiedades que pueden ser extraídas directamente del archivo. Las ventajas que este enfoque presenta son: su simplicidad, rapidez y bajo coste computacional, ya que no requiere de entornos de ejecución específicos ni de hardware especializado para probar el comportamineto del programa. Sin embargo, el mayor problema de este tipo de análisis es su dificultad para detectar malware que utiliza técnicas avanzadas de ofuscación o cifrado, ya que estas prácticas dificultan la extracción de información útil del binario.

Análisis dinámico

Técnica de detección de *malware* que consiste en evaluar el comportamiento de un programa mediante su ejecución en un entorno controlado, con el objetivo de observar sus interacciones con el sistema operativo, los recursos de este y otros programas. En este enfoque, se monitorizan actividades como la modificación de archivos, el tráfico de red generado, la creación de procesos o la inyección de código en estos, lo cual permite identificar patrones de comportamiento asociados con programas maliciosos. A diferencia del análisis estático, el análisis dinámico ofrece una mayor precisión, ya que puede detectar comportamientos maliciosos que no son evidentes simplemente escaneando el archivo de manera estática, como el uso de técnicas de ofuscación. Sin embargo, este tipo de análisis sigue teniendo sus inconvenientes, por un lado, es más complejo, requiere de más recursos computacionales y es más costoso de implementar, dado que involucra la ejecución real del código en un entorno controlado, generalmente una máquina virtual (*sandbox*). Por otro lado, también es poco eficiente contra casos en los que el *malware* detecta el hecho de que está siendo analizado y oculta su comportamiento malicioso. Además, puede no ser adecuado para dispositivos con recursos limitados, como dispositivos IoT (*Internet Of Things*) o móviles, debido a sus altos requerimientos de hardware y tiempo.

Análisis híbrido

Metodo de detección de *malware* el cual combina las fortalezas tanto del análisis estático como del dinámico. En este método, el programa se ejecuta en un entorno controlado, y durante su ejecución, se realizan *dumps* de memoria de manera periódica o en respuesta a comportamientos sospechosos. Estos volcados de memoria son posteriormente analizados utilizando técnicas de análisis estático para identificar posibles patrones maliciosos, tales como la inyección de código en procesos ajenos, manipulación de memoria que no le pertenece al programa o modificaciones en partes protegidas de la memoria pertenecientes al sistema oeprativo. Este enfoque permite una detección más precisa de *malware* que utiliza técnicas avanzadas de ocultamiento, ya que combina la observación del comportamiento en tiempo real con la inspección detallada del estado de la memoria. Sin embargo, el análisis híbrido es el más complejo y costoso de implementar, ya que requiere tanto de infraestructura de virtualización como de herramientas para realizar un buen análisis de memoria. A pesar de todo, suele ofrecer los mejores resultados en términos de detección.

Huella digital (*fingerprinting*)

El fingerprinting o, la generación de huellas digitales de archivos, es una técnica utilizada para identificar de manera única un archivo mediante la aplicación de funciones criptográficas de *hashing*. Este proceso consiste en calcular un *hash* a partir del contenido completo del archivo utilizando algoritmos como MD5, SHA-1, SHA-256 u otros. El resultado es una cadena de longitud fija que actúa como un identificador único para ese archivo. Cualquier modificación, por mínima que sea, en el contenido del archivo resultará en un *hash* completamente diferente, lo que permite detectar alteraciones o corrupciones en estos.

Esta técnica es muy utilizada en la verificación de la integridad de archivos, la detección de duplicados, y la identificación de malware conocido al comparar el *hash* que este genera con una base de datos de muestras previamente catalogadas. Sin embargo, una limitación importante del *fingerprinting* es su sensibilidad extrema a cambios mínimos, lo que dificulta la identificación de archivos que han sido ligeramente modificados pero que conservan una estructura o funcionalidad. Esto implica que incluso cambiar un bit en el *padding* del archivo, hace que este ya no se detecte como malware al tener una huella digital diferente.

Huella digital difusa (*fuzzy hashing*)

El *fuzzy hashing*, o *hashing* difuso, es una técnica que extiende el concepto del *hashing* tradicional al permitir la comparación de archivos basada en similitudes parciales en lugar de una coincidencia exacta. A diferencia del *hashing* convencional, que opera sobre el archivo completo, el *fuzzy hashing* divide el archivo en bloques o segmentos y calcula un *hash* para cada uno de ellos. Este enfoque por bloques permite identificar similitudes entre archivos incluso cuando solo una porción de su contenido ha sido modificada.

El *fuzzy hashing* es particularmente útil en el análisis forense digital y la detección de *malware*, ya que permite identificar variantes de archivos maliciosos que han sido modificaods para evadir su detección, pero que conservan partes significativas de su código original. Al comparar dos *hashes* difusos, es posible calcular un grado de similitud basado en la cantidad de bloques que coinciden entre ambos. Esto se logra mediante algoritmos especializados como SSDeep o TLSH, que están diseñados para generar *hashes* difusos y medir la similitud entre ellos.

Android y su formato ejecutable (*Android Package Kit* o APK)

Android es el sistema operativo móvil más utilizado del mundo. Originalmente, sus aplicaciones se programaban principalmente en Java, un lenguaje que se ejecuta sobre una máquina virtual conocida como la *Java Virtual Machine* (JVM). Android adoptó este enfoque pero con su propia máquina virtual optimizada para dispositivos móviles, primero llamada Dalvik y ahora ART (*Android Runtime*). Las aplicaciones modernas suelen usar Kotlin, un lenguaje más actual pero que sigue compilando al mismo formato compatible con ART.

El formato ejecutable de una aplicación de Android es un APK (*Android Package Kit*). En esencia, un APK no es más que un archivo comprimido en formato ZIP que contiene todo lo necesario para que la aplicación se pueda instalar y funcionar correctamente. Su estructura incluye elementos clave como el archivo `AndroidManifest.xml`, que sería similar al «DNI» de la aplicación: declara sus componentes, permisos necesarios y características. También contiene los archivos `classes.dex`, que incluyen el código de la aplicación compilado, y carpetas con recursos como imágenes o textos.

Desde el punto de vista de la seguridad, esta estructura tiene una doble cara. Al ser básicamente un ZIP, es relativamente sencillo de descomprimir e inspeccionar, lo que facilita enormemente el análisis estático. Sin embargo, esta misma simplicidad también facilita que un atacante pueda modificarlo (haciendo ingeniería inversa de su contenido). Para dificultar este proceso, los desarrolladores pueden emplear técnicas de ofuscación de código, que lo hacen más difícil de leer y entender, aunque no imposible para alguien que tenga el conocimiento necesario.

3.3. Otros conceptos

Dadas las bases anteriores necesarias para entender el proyecto, se dejan a continuación aquellos conceptos adicionales que pueden ser de gran ayuda para comprender mejor otros aspectos relevantes del proyecto.

Framework web

Un *framework* web es un conjunto de herramientas, bibliotecas y componentes predefinidos que facilitan el desarrollo de aplicaciones y páginas web. Su propósito es ofrecer una estructura básica que los desarrolladores

pueden utilizar para crear aplicaciones de manera más eficiente, sin tener que comenzar desde cero. Los *frameworks* web incluyen funcionalidades que resuelven tareas comunes, como el manejo de rutas (URLs), la conexión a bases de datos, la autenticación de usuarios y la seguridad, lo que ahorra tiempo y esfuerzo durante el desarrollo.

Además, proporcionan una organización estructurada para el código, lo que facilita la colaboración entre desarrolladores y simplifica el mantenimiento de los proyectos a largo plazo. Muchos *frameworks* también incluyen mecanismos de seguridad incorporados para proteger las aplicaciones contra amenazas comunes y herramientas que automatizan tareas repetitivas, como la gestión de dependencias y la ejecución de pruebas. Todo esto permite construir aplicaciones web escalables, seguras y fáciles de mantener.

Algunos *frameworks* web populares son: Django (para Python), Ruby on Rails (para Ruby), Angular y React (para JavaScript), y Laravel (para PHP).

4. Técnicas y herramientas

En este apartado se comentan y analizan las diferentes herramientas y técnicas usadas en la realización de este proyecto.

4.1. Lenguajes y entorno de programación

En esta sección se describe el lenguaje de programación principal y los conceptos y herramientas de entorno que han sido fundamentales para la organización y el desarrollo del proyecto.

Python

Python [19] es un lenguaje de programación alto nivel, interpretado y multipropósito, conocido por su sintaxis clara y sencilla muy similar al inglés. Su filosofía de diseño se centra en torno a la facilidad de uso, lo que lo ha convertido en uno de los lenguajes más populares del mundo para todo tipo de personas que desean tanto adentrarse en el mundo de la programación, como aquellas que ya tienen cierta experiencia en el campo, especialmente si se trata de tareas de automatización, de ciencia de datos o incluso de IA. Especialmente en este último grupo, prácticamente todo el ecosistema de IA moderno se desarrolla sobre Python. Todo esto se debe, principalmente a su enorme comunidad, su facilidad para crear prototipos rápidos y, sobre todo, su vasto repertorio de librerías especializadas y optimizadas para el cálculo científico y el aprendizaje automático.

Entorno virtual

Un entorno virtual es un concepto de programación, especialmente popular en Python, que consiste en crear un directorio aislado que contiene una instalación propia de Python y todas las dependencias específicas de un proyecto dado. Funciona como una «burbuja» que separa los paquetes de un proyecto de los de otros proyectos y de la instalación global del sistema. Esto resuelve el clásico problema de tener diferentes proyectos que requieren versiones distintas de las mismas librerías [16].

La principal ventaja de usar entornos virtuales es que son fáciles de reproducir y de limpiar. Permiten saber con exactitud qué dependencias necesita un proyecto y evitan «contaminar» el sistema con paquetes que solo se usan para una tarea concreta. Cuando un proyecto finaliza, basta con borrar la carpeta del entorno para eliminar todo rastro de este, sin dejar dependencias huérfanas. Sin embargo, su gestión tradicional con herramientas como `venv` [24] y `pip` [12] puede ser manual y algo tediosa, ya que uno debe encargarse de activarlos, desactivarlos y registrar las dependencias explícitamente.

Poetry

Poetry [13] es una herramienta de gestión de dependencias y empaquetado para Python. A diferencia de los métodos tradicionales que combinan herramientas como `pip` [12], `requirements.txt` y `venv` [24], Poetry integra todas estas funcionalidades bajo una única interfaz de comandos y un archivo de configuración llamado `pyproject.toml`. Dicho archivo puede ser modificado para declarar las dependencias del proyecto, características del paquete final, versiones y otro tipo de configuraciones adicionales para la generación de distribuciones y reglas de resolución de paquetes. Poetry se encargará posteriormente de generar a partir de dicho archivo de configuración un fichero llamado `poetry.lock`, el cual contiene todas las versiones exactas de cada paquete y sus dependencias correspondientes, correctamente resueltas y guardadas en un formato que permitirá posteriormente reproducir el entorno de forma simple y directa.

En este caso, Poetry no solo es útil para proyectos grandes sino que es extremadamente cómodo para cualquier proyecto de Python que necesite de un par de dependencias, puesto que permite crear entornos virtuales con solo un par de comandos, permitiendo así no modificar la instalación global de Python en el sistema y permitiendo una gestión más compleja de ciertas dependencias en función de las prestaciones del equipo que uno posee o su

sistema operativo. Por ejemplo, en este proyecto, una de las librerías usadas es PyTorch, la cual cuenta con soporte para aceleración por GPU de sus operaciones, pero, este soporte viene incluido en un paquete diferente al del paquete que solo permite el uso de la CPU. Usando Poetry es posible crear una configuración dinámica que sea capaz de adaptarse al equipo de cada uno, descargando o no la versión con o sin aceleración por hardware en función de si el equipo es compatible con ello.

Conda

Conda [4] es un sistema de gestión de paquetes y entornos de código abierto y multiplataforma. Aunque puede gestionar cualquier lenguaje, es extremadamente popular en la comunidad de Python, especialmente para todo lo relacionado con ciencia de datos e IA. Su distribución principal, Anaconda, viene con un enorme conjunto de librerías científicas preinstaladas, facilitando su uso y dando acceso a todo lo necesario para empezar a realizar pruebas y desarrollar proyectos desde el momento en el que se instala. Por otro lado, existe también una versión mínima, Miniconda [14], que solo incluye el gestor de entornos y permite al usuario instalar únicamente lo que necesita, siendo una opción más ligera y la preferida por muchos simplemente porque, a pesar de requerir un poco más de tiempo y conocimiento en preparar el entorno de trabajo, uno puede instalar específicamente solo lo que necesita.

La gran ventaja de Conda es que gestiona no solo paquetes de Python, sino también dependencias que no son de Python (como librerías de C o compiladores), lo que simplifica mucho la instalación de paquetes complejos como PyTorch. Sin embargo, una de sus desventajas es que utiliza sus propios repositorios de paquetes, que a veces no están tan actualizados como el repositorio oficial de Python (PyPI). Además, compartir y replicar entornos de forma exacta puede ser menos directo que con herramientas más modernas como Poetry.

Jupyter Notebook

Un Jupyter Notebook [17] es un entorno de desarrollo interactivo basado en una aplicación web que permite crear y compartir documentos que contienen código mixto, ecuaciones, visualizaciones y texto. Funciona como un «cuaderno de laboratorio digital», donde se puede escribir y ejecutar código en bloques o «celdas» de forma independiente. Esto es extremadamente útil para el prototipado y el análisis exploratorio de datos.

Su principal ventaja es la interactividad. Permite ejecutar una celda costosa (como cargar un gran dataset o entrenar un modelo) una sola vez, y luego seguir trabajando en otras celdas (como visualizar resultados o probar transformaciones) sin tener que volver a ejecutar todo el script desde el principio. Esto ahorra una cantidad de tiempo enorme, especialmente en este tipo de tareas de experimentación.

Jupyter ha sido una herramienta indispensable a lo largo de todo este proyecto, especialmente en las fases de prototipado y análisis. Se utilizó para experimentar con la extracción de características, para depurar el proceso de creación del *dataset*, para entrenar las primeras versiones del modelo, para visualizar los resultados con Matplotlib e incluso para ejecutar los análisis de interpretabilidad con SHAP. Fue el «banco de pruebas» donde se validaron la mayoría de las ideas antes de integrarlas en el código final del proyecto.

4.2. Librerías

En esta sección se describen las librerías de Python más importantes que se han utilizado a lo largo de todo el proyecto.

PyTorch vs Keras

PyTorch [20] y Keras [55] representan dos de los *frameworks* de *deep learning* más conocidos y utilizados en la actualidad. Si bien ambos facilitan la construcción de redes neuronales, cada uno toma una filosofía de diseño distinta, lo cual los hace más adecuados para diferentes tipos de proyectos y usuarios. Keras funciona como una interfaz de alto nivel, diseñada para la simplicidad y el desarrollo rápido, mientras que PyTorch opera a un nivel más bajo, ofreciendo un control granular y una mayor flexibilidad a la hora de desarrollar modelos personalizados y específicos.

Keras se caracteriza principalmente por su facilidad de uso, debido a que permite construir y entrenar modelos estándar con muy pocas líneas de código. Su API es bastante intuitiva y abstrae gran parte de la complejidad subyacente, lo que lo convierte en una opción excelente para principiantes y para la creación rápida de prototipos y para entornos de producción donde la estandarización es clave. Sin embargo, esta simplicidad conlleva una menor flexibilidad, ya que realizar modificaciones sustanciales en la arquitectura o en el ciclo de entrenamiento de los modelos puede volverse complejo y poco intuitivo debido a no estar diseñado para ello.

Características	Pytoch	Keras
Flexibilidad	Alta, permite crear redes neuronales personalizadas y complejas.	Menos flexible, enfocado en redes estándar.
Facilidad de uso	Requiere más código y tiene una curva de aprendizaje más pronunciada.	Fácil de usar, ideal para desarrolladores principiantes.
Personalización	Excelente para redes personalizadas y modelos avanzados.	Limitada, más orientada a redes convencionales.
Comunidad y soporte	Muy popular en la investigación académica y proyectos avanzados.	Amplio uso en la industria por su simplicidad.
Uso principal	Investigación, redes neuronales complejas.	Desarrollo rápido de modelos estándar.

Tabla 4.1: Comparativa entre PyTorch y Keras

Por el contrario, PyTorch proporciona un set de herramientas mucho más versátil, pensado para la investigación y para proyectos que requieren de arquitecturas personalizadas. En general, permite crear modelos de forma mucho más granular debido a que se basa en proporcionar al usuario con un conjunto de módulos y funciones que pueden ser instanciadas juntas para formar un modelo complejo y personalizado, especializado en la tarea que se desee. A su vez, permite definir de manera más concreta el set de datos a usar, el preprocesado de esos datos y el cómo se entrena el modelo a partir de ellos.

En concreto, para este proyecto, PyTorch fue la elección más lógica. La arquitectura del modelo de clasificación de APKs no es convencional; se procesan un montón de cadenas de caracteres y vectores, los cuales requieren de un preprocesado específico y de la implementación de un *embedder* personalizado para poder usar posteriormente dichas características junto con una lógica y control sobre el proceso de entrenamiento bastante concreto.

NumPy, Pandas y Matplotlib

NumPy, Pandas y Matplotlib constituyen la trilogía fundamental de librerías sobre las que se edifica gran parte del ecosistema de paquetes científicos y de data science en Python.

- **NumPy (*Numerical Python*):** Es la librería base para la computación numérica en Python. Proporciona el concepto de ndarray, una estructura de datos para la creación de arrays N-dimensionales eficientes, y un vasto conjunto de funciones matemáticas para operar sobre ellos. Una de las mayores ventajas de NumPy es su velocidad y eficiencia tanto en tiempo, como en espacio al trabajar con grandes sets de datos. Este rendimiento se debe principalmente a que muchas de sus operaciones están implementadas en C y aprovechan la vectorización, permitiendo ejecutar operaciones complejas en arrays completos sin necesidad de bucles explícitos en Python [15].

- **Pandas:** Construida sobre NumPy, Pandas introduce estructuras de datos de alto nivel, principalmente el DataFrame, una tabla bidimensional heterogénea e indexada, pensada para manejar datos tabulares y series temporales. Facilita enormemente tareas como la lectura y escritura de datos, la limpieza, el filtrado, la agregación y la transformación, siendo una herramienta muy útil estándar para el preprocesamiento de datos [2].

- **Matplotlib:** Es la librería de visualización de datos por excelencia en Python. Permite elegir entre un gran repertorio de gráficos comunes como pueden ser los gráficos de barras o histogramas a la creación de gráficos personalizados, facilitando el trabajo de representar datos complejos y hacerlos agradables a la vista [1].

Estas tres librerías han sido de gran ayuda en diferentes etapas del proyecto. Pandas ha sido la herramienta principal para estructurar las características extraídas de los archivos APK en un DataFrame limpio y manejable, facilitando todo el preprocesamiento. NumPy ha sido utilizado de forma subyacente por Pandas y PyTorch, y directamente para realizar operaciones numéricas eficientes sobre los datos ya procesados antes de introducirlos en el modelo. Finalmente, Matplotlib ha sido usada para la evaluación del modelo, permitiendo visualizar diferentes aspectos del entrenamiento del modelo y la comparación de este con otros de una manera más visual.

scikit-learn

scikit-learn [23] es la librería de referencia para el aprendizaje automático clásico en Python. Proporciona un conjunto enorme y bien documentado de herramientas para prácticamente cualquier tarea de *machine learning*, incluyendo algoritmos de clasificación, regresión, *clustering*, reducción de dimensionalidad y utilidades para el preprocesamiento de datos y la evaluación de modelos. Su mayor ventaja es su API unificada y consistente que facilita en gran medida el trabajo de entrenar modelos y trabajar con ellos: todos sus objetos (modelos, transformadores, etc.) comparten una interfaz común (`.fit()`, `.predict()`, `.transform()`), lo que hace que sea muy fácil de aprender y usar.

A lo largo del proyecto, scikit-learn ha sido de gran ayuda a la hora de comparar la red neuronal con otros modelos clásicos. Se utilizó para entrenar y evaluar dicho modelos (SVM, k-NN, RandomForest, Regresión Logística) con los que poder comprar posteriormente la red. Además, sus funciones de utilidad, como las de división de datos (`train_test_split`) y cálculo de métricas (`accuracy_score()`, `precision_score()`, `roc_curve()`, ...), fueron de gran ayuda para poder obtener cifras acerca del rendimiento de los distintos clasificadores.

Optuna

Optuna [25] es un *framework* de optimización de hiperparámetros moderno que sirve de sustituto a métodos de búsqueda tradicionales basados en fuerza bruta por una estrategia de optimización secuencial e informada. Su diseño «*define-by-run*» lo hace extremadamente flexible y le permite integrarse de forma simple con variedad de *frameworks* de aprendizaje automático como es el caso de PyTorch.

Por defecto, el algoritmo que Optuna utiliza internamente es el *Tree-structured Parzen Estimator* [59] (TPE). Este método funciona de una manera bastante ingeniosa: en lugar de modelar directamente la probabilidad de que unos hiperparámetros den un buen resultado, se modelan dos distribuciones de probabilidad distintas. La primera, $l(x)$, representa la distribución de los hiperparámetros que han dado lugar a buenos resultados (pérdida baja), y la segunda, $g(x)$, la de aquellos que han dado lugar a malos resultados. En cada nuevo paso, Optuna busca un conjunto de hiperparámetros que sea muy probable bajo el modelo «bueno» y muy poco probable bajo el modelo «malo», maximizando así la probabilidad de encontrar una configuración cada vez mejor. Este enfoque dirigido es mucho más eficiente

que una búsqueda aleatoria y más óptimo que una búsqueda exhaustiva porque poda una gran cantidad de casos que no son necesarios probar si se conoce que parte de sus hiperparámetros no generan buenos resultados.

Dado que el rendimiento de una red neuronal depende enormemente de sus hiperparámetros (tasa de aprendizaje, número de capas, etc.), se eligió Optuna para automatizar y optimizar este proceso de búsqueda. Su sencilla integración con PyTorch permitió definir un espacio de búsqueda y dejar que Optuna explorara de forma eficiente cientos de combinaciones, encontrando una configuración casi óptima en mucho menos tiempo de lo que habría llevado un enfoque manual o mediante una búsqueda exhaustiva.

SHAP (*SHapley Additive exPlanations*)

SHAP (*SHapley Additive exPlanations*) es una técnica de interpretabilidad de modelos que responde a una pregunta fundamental: si un modelo ha tomado una decisión, ¿cuánto ha contribuido cada una de las características de entrada a ese resultado final? Para ello, se basa en un concepto de la teoría de juegos cooperativos llamado los valores de Shapley [44]. La analogía es simple: si un equipo gana un premio, ¿cómo se reparte el dinero de forma justa entre sus miembros, teniendo en cuenta la aportación de cada uno? SHAP permite resolver exactamente este problema, pero equiparando a los «jugadores» y al «juego» con las características de un modelo y su predicción respectivamente.

El algoritmo funciona de una manera teóricamente muy elegante. Para calcular la contribución de una característica (por ejemplo, «la aplicación solicita acceso a los contactos»), SHAP considera todas las combinaciones posibles del resto de características (las «coaliciones»). Luego, mide cuánto cambia la predicción del modelo cuando se añade esa característica a cada una de esas coaliciones. Al promediar este «cambio marginal» a través de todas las coaliciones posibles, se obtiene la contribución justa y única de esa característica a la predicción final. El resultado es una explicación aditiva: la suma de las contribuciones de todas las características nos da exactamente la diferencia entre la predicción concreta y la predicción media del modelo.

SHAP se utilizó para analizar e interpretar las predicciones de los distintos modelos entrenados. Permitiendo visualizar qué características (permisos, llamadas a la API del sistema, etc.) eran las más influyentes para cada modelo a la hora de clasificar una aplicación como *malware*. Además, debido a su naturaleza aditiva e individual, SHAP puede ser utilizado tanto para el análisis local de una predicción concreta, como para obtener una explicación

general del rendimiento del modelo y del porque de sus decisiones, convirtiéndolo en una herramienta muy útil y versátil para analizar los distintos modelos que se entrenaron.

Es importante destacar que SHAP es una herramienta muy buena para analizar el funcionamiento interno y las razones por las cuales un modelo concreto decide y sirve muy bien para ver en qué se basa este. SHAP no es una herramienta que se pueda utilizar como fuente fiable para validar relaciones de causalidad [3], es decir, que un modelo prediga un valor para un problema en función de ciertas características, no implica que esas características sean necesariamente la razón por la cual, en la vida real, este sea el factor determinante.

umap-learn

umap-learn es la librería de Python de referencia para aplicar el algoritmo de reducción de dimensionalidad UMAP (*Uniform Manifold Approximation and Projection*) [**?**]. Su principal propósito es proporcionar una herramienta práctica y eficiente para tomar datos de alta dimensionalidad, como pueden ser los vectores de características generados por un *embedder*, y proyectarlos en un espacio de dos o tres dimensiones para su visualización.

Una de las grandes ventajas de esta librería es su diseño, el cual sigue las convenciones de la API de scikit-learn. Esto significa que cualquier persona familiarizada con el ecosistema de ciencia de datos en Python puede utilizarla de forma muy intuitiva, aplicando los métodos `.fit()` y `.transform()` para reducir la dimensionalidad de sus datos en apenas unas pocas líneas de código. Aunque el algoritmo subyacente es matemáticamente complejo, la librería lo abstrae por completo, permitiendo al usuario centrarse en la interpretación de los resultados.

En este proyecto, se ha utilizado la librería umap-learn para visualizar el espacio de características aprendido por el *embedder* de la red neuronal. Su velocidad y su reconocida capacidad para preservar tanto la estructura local como global de los datos la convirtieron en la herramienta ideal para generar gráficos de dispersión 2D y así poder confirmar visualmente si el modelo estaba agrupando de forma coherente las aplicaciones benignas y maliciosas en regiones distintas del espacio.

Streamlit

Streamlit [21] es un *framework* de código abierto para Python, diseñado específicamente para la creación y el despliegue rápido de aplicaciones web

interactivas para proyectos de *data science* y aprendizaje automático. Su filosofía se basa en la simplicidad radical, permitiendo transformar *scripts* convencionales con código de procesamiento de datos, modelos de IA o simples funciones aisladas en aplicaciones web funcionales con un esfuerzo y conocimiento de desarrollo web mínimos.

A diferencia de *frameworks* web más tradicionales y complejos como Django o Flask, que exigen la gestión de rutas, una organización de los ficheros del proyecto específica, sintaxis inusual, plantillas HTML y lógica de servidor, Streamlit permite construir una interfaz de usuario directamente desde un script normal de Python. Con comandos sencillos, se pueden añadir elementos interactivos como botones, deslizadores, gráficos y, fundamentalmente para este caso, campos para la subida de archivos. Esto acelera drásticamente el ciclo de desarrollo puesto que uno puede simplemente centrarse en obtener un modelo o set de funcionalidades que son correctas y dan buenos resultados sin preocuparse mucho de cómo se llevará luego esto a una interfaz gráfica o aplicación de escritorio / web puesto que la conversión es muy sencilla en la mayoría de casos.

La finalidad de la aplicación web en este proyecto es ofrecer una demostración tangible y una especie de *demo* o entorno de prueba para que cualquiera pudiera probar el clasificador de *malware* de forma sencilla. El objetivo era crear una interfaz simple donde un usuario pudiera subir un archivo `.apk` y recibir una predicción de manera inmediata. Streamlit fue la herramienta perfecta para esta tarea, ya que permitió desarrollar esta funcionalidad en cuestión de horas en lugar de días y el resultado es más que suficiente para el alcance deseado.

Androguard y otros analizadores

Androguard [7] es una potente herramienta de código abierto y un paquete de Python, diseñada específicamente para el análisis estático y la ingeniería inversa de aplicaciones de Android (archivos APK). Su función principal es el proceso de diseccionar un archivo APK para extraer información detallada sobre su estructura y contenido sin necesidad de ejecutar la aplicación. Una de las mayores ventajas de Androguard es el hecho de que permite además realizar todo este proceso de extracción de características de forma automatizada puesto que, expone una API de Python bastante simple e intuitiva que proporciona acceso a todos los datos que se pueden obtener de analizar las APKs.

A través de Androguard, es posible acceder a componentes como el *manifest* de la aplicación (`AndroidManifest.xml`) para analizar permisos y componentes declarados, desensamblar el código Dalvik (DEX) para inspeccionar las clases y los métodos e, incluso, extraer recursos como cadenas de texto o certificados. Aunque existen otras herramientas de análisis como MobSF [8] (que ofrece un entorno más automatizado y visual) o Jadx [9] (un popular descompilador), Androguard destaca por su granularidad, simplicidad de uso y su naturaleza como librería de Python.

Una de las piedras fundamentales de este proyecto es la posibilidad de extraer características de forma estática de APKs y entrenar un modelo con ellas capaz de discernir entre aplicaciones benignas y malignas. La razón principal de su eso es el hecho de funcionar nativamente en Python y permitir la automatización de la creación del *dataset* de entrenamiento del modelo. A su vez, es el componente que permite analizar muestras nuevas, obteniendo los datos que el modelo espera recibir para realizar una predicción.

4.3. Otras herramientas

Finalmente, en esta sección se detallan otras herramientas de software que, aunque no están directamente relacionadas con la inteligencia artificial, han sido de gran ayuda para la gestión del código, el desarrollo y la documentación del proyecto.

Docker

Docker [11] es una plataforma de código abierto que permite automatizar el despliegue, la ejecución, la distribución y la gestión de aplicaciones mediante el uso de la «containerización». Esta herramienta permite empaquetar una aplicación y todas sus dependencias como bibliotecas, herramientas del sistema, código, comandos de ejecución, configuraciones y todo lo que uno pueda necesitar para ejecutar su aplicación en una unidad independiente y aislada llamada contenedor.

El principal problema que Docker intenta resolver es el clásico «en mi equipo funciona», donde una aplicación se ejecuta correctamente en el entorno de un desarrollador pero falla en otro debido a diferencias en la configuración del sistema operativo o en las versiones de las dependencias, además de posibles errores en la ejecución del proyecto y diversas posibles causas. Un contenedor soluciona esto debido a que, uno puede simplemente distribuir un único archivo que describe el proceso de creación del contenedor llamado

Dockerfile (o la misma imagen del contenedor ya creado) garantizando que cualquiera podrá ejecutar el proyecto exactamente de la misma manera en cualquier máquina que soporte Docker, desde un portátil local hasta un servidor en la nube. Esto asegura la consistencia, la reproducibilidad y simplifica enormemente los flujos de trabajo desarrollo.

El entorno de este proyecto es complejo, requiriendo versiones específicas de Python y múltiples librerías (PyTorch, Androguard, etc.). Para asegurar que la aplicación web y el modelo pudieran ser ejecutados por cualquier persona sin un tedioso proceso de configuración manual, se utilizó Docker para empaquetar todo en un contenedor. Esto garantiza que el proyecto funcione de la misma manera en cualquier sistema, simplificando enormemente su despliegue y asegurando que los resultados fueran reproducibles.

GitHub

GitHub [10] es una plataforma de desarrollo colaborativo basada en la web que utiliza el famosos sistema de control de versiones llamado Git [6]. Permite a los desarrolladores alojar y gestionar sus repositorios de código, que son esencialmente carpetas de un proyecto con un historial completo de todos los cambios realizados. Además de alojar el código, GitHub ofrece herramientas para el seguimiento de errores, la revisión de código y la gestión de proyectos, facilitando el trabajo en equipo.

Todo el código fuente del proyecto ha sido alojado en un repositorio de GitHub. Esto sirvió principalmente para tener un espacio compartido desde el cual poder trabajar desde diferentes equipos manteniendo la coherencia de los archivos. Además, sirvió como copia de seguridad por si acaso algo salía mal y era necesario recuperar los archivos de versiones anteriores.

Visual Studio Code (VSCode)

Visual Studio Code (VSCode) [5] es un editor de código fuente liviano y simple muy utililzado en el mundo del desarrollo gracias a su diseño mínimo, velocidad su carácter modular. El editor en sí es bastante completo y sirve perfectamente para su propósito pero, lo que lo eleva al siguiente nivel es su enorme ecosistema de extensiones que permiten adaptarlo para trabajar con prácticamente cualquier lenguaje o tecnología. Una alternativa muy popular en el mundo de Python es la suite de JetBrains, especialmente su IDE PyCharm [18].

VSCode fue el editor de código principal utilizado para el desarrollo de todo el proyecto. Su flexibilidad, su excelente soporte para Python y sus

extensiones para Docker [11] y Jupyter [17] lo convirtieron en la herramienta ideal para gestionar las diferentes facetas del trabajo desde un único lugar.

TEXstudio

TEXstudio [22] es un Entorno de Desarrollo Integrado (IDE) diseñado específicamente para la creación de documentos con LATEX. LATEX es un sistema de composición de textos de alta calidad, muy utilizado en el mundo académico y científico para la escritura de artículos, tesis y libros, ya que maneja de forma excepcional las fórmulas matemáticas, las referencias cruzadas y la maquetación profesional. TEXstudio facilita el trabajo con LATEX al proporcionar un visor de PDF integrado, autocompletado de comandos, resaltado de sintaxis y otras herramientas que agilizan y simplifican el proceso de escritura.

Toda la documentación de este proyecto, incluyendo esta memoria y sus anexos, ha sido escrita en LATEX utilizando la plantilla proporcionada por la universidad [56]. TEXstudio fue la herramienta elegida para esta tarea, ya que su entorno integrado hizo mucho más cómodo y eficiente el proceso de redactarla.

5. Aspectos relevantes del desarrollo del proyecto

En este capítulo se expone de manera detallada el recorrido completo del proyecto, desde su concepción inicial hasta la evaluación de los resultados finales. Se narrará el proceso de investigación, las decisiones de diseño, los desafíos técnicos encontrados y las soluciones implementadas. El objetivo es ofrecer una visión transparente y cronológica de todo el trabajo realizado, explicando no solo el «qué» se ha hecho, sino también el «porqué» de cada paso. Se comenzará con la fase de exploración y prototipado, para luego profundizar en la creación del conjunto de datos propio, la optimización del modelo y el análisis de su rendimiento e interpretabilidad, culminando con el desarrollo de una aplicación web para su demostración.

5.1. Pruebas iniciales y estado del arte

La primera etapa del proyecto fue de carácter exploratorio. El objetivo era investigar el panorama actual del mundo de la ciberseguridad y cómo se podía aplicar la IA al mismo, validando dicha idea central y construyendo un prototipo inicial que sirviera como base y prueba de concepto para el resto del trabajo.

Estado del arte y base del proyecto

En sus orígenes, la idea principal de este proyecto era considerablemente más más abstracta y se centraba en un área distinta de la ciberseguridad. La intención inicial era explorar si las técnicas de inteligencia artificial (IA) podrían utilizarse para detectar o mitigar vulnerabilidades de *hardware*, como

los ataques de canal lateral (*side-channel*) y de ejecución especulativa, del estilo de Spectre y Meltdown. Sin embargo, tras una primera investigación acerca de la literatura científica disponible, se determinó que este campo era extremadamente complejo y la cantidad de trabajos que aplicaban IA a este problema era muy escasa, concluyendo pues que, este enfoque habría sido demasiado ambicioso y un poco «cogido con pinzas».

Esta falta de base sólida motivó un giro en la investigación hacia un área más consolidada: el análisis de *malware*. Se estudiaron los diferentes enfoques de análisis (estático, dinámico e híbrido) y, de entre ellos, se determinó que el más interesante para indagar en el sería el análisis estático, principalmente debido a su principal ventaja: la capacidad de detectar *malware* sin necesidad de ejecutar el software sospechoso, eliminando así prácticamente cualquier riesgo de infección para el sistema anfitrión. Cabe destacar que, aunque se consideró la idea de un enfoque híbrido, combinando una primera fase estática con un análisis dinámico guiado el cual se ejecutaría dentro de un entorno de *sandbox*, la complejidad de implementar un sistema así excedía el alcance de este trabajo.

Por tanto, la idea del proyecto se estructuró en torno a la construcción de un clasificador de *malware* basado únicamente en características estáticas. Inicialmente, se pensó en trabajar con formatos de ejecutables de escritorio como pueden ser ELF (Linux) o PE (Windows). Sin embargo, la complejidad inherente de estos formatos, especialmente la dificultad de desensamblar o decompilar código máquina nativo para extraer información útil más allá de las cabeceras, presentaba una problemática considerable. Es por ello por lo que, la solución más sencilla y que simplificaba enormemente este proceso sería centrarse en el formato APK de Android. Al ser esencialmente un archivo comprimido, su código fuente es fácilmente accesible y su fichero `AndroidManifest.xml` proporciona una enorme cantidad de información de alto nivel, lo que facilitaba enormemente la tarea de extracción de características.

Con el objetivo claro, la investigación se centró en validar la viabilidad de aplicar redes neuronales profundas a este problema. La revisión de trabajos como los de Wu et al. [61] o Bakour et al. [29] confirmaron que el análisis estático de malware mediante métodos de aprendizaje automático para Android era un campo bastante activo y con potencial. El hallazgo clave fue el *paper* de İbrahim et al. [64], *A Method for Automatic Android Malware Detection Based on Static Analysis and Deep Learning*. Este trabajo demostraba de manera contundente que un modelo de *deep learning* no solo podía aprender a clasificar *malware* con características estáticas, sino que

podía alcanzar resultados extraordinariamente prometedores. Este estudio se convirtió en la principal motivación y en el documento de referencia sobre el que se construiría todo el proyecto de aquí en adelante.

Búsqueda de un *dataset* de pruebas (Drebin)

Una vez que la dirección del proyecto estuvo clara, el siguiente paso lógico era intentar replicar las ideas del *paper* de referencia [64] y construir un prototipo. Este paso serviría para confirmar de manera práctica sus conclusiones y para empezar a entender los desafíos de implementación del modelo, ya que el artículo, si bien era detallado, omitía ciertos detalles importantes acerca tanto de la estructura del modelo, sus hiperparámetros, configuraciones específicas de su *embedder* o detalles acerca del mismo *pipeline* de datos empleado. Pero, sin duda alguna, el mayor obstáculo fue que los autores no proporcionaban acceso al *dataset* que utilizaron ni detallaban su proceso de creación, lo que dificultaba en gran medida replicar exactamente sus descubrimientos.

Por ello, se comenzó la búsqueda de un *dataset* público que fuera lo más similar posible en cuanto a las características extraídas. Se consideraron varias opciones, como era el caso de CICMalDroid2020 [45] o de Drebin [28], eligiendo finalmente la segunda de estas opciones. La razón principal fue que, de entre los disponibles, era el que presentaba un conjunto de características más parecidas al descrito en el *paper* de referencia, además de ser un *dataset* bastante grande, conteniendo cerca de 170 000 muestras y era ampliamente utilizado por la comunidad. Otro de los motivos principales de buscas un *dataset* preexistente de forma tan temprana en el proyecto era para aislar el problema de la recolección de datos y centrarse principalmente en conseguir construir un prototipo cuanto antes.

El formato original del *dataset* de Drebin, sin embargo, no era directamente utilizable. Consistía de un archivo CSV con los identificadores (hashes SHA256) de las muestras maliciosas y un directorio con miles de ficheros de texto, uno por cada aplicación (tanto benigna como maliciosa). Cada uno de estos ficheros contenía las características extraídas en un formato de tipo `CLAVE=VALOR` por línea. El primer paso para poder usar dicho *dataset* fue, por tanto, procesar toda esta estructura y condensarla en un único archivo CSV fácilmente manipulable. Se creó un *script* de Python que interaba sobre todos los ficheros, leía las características de cada aplicación y las agrupaba en un formato tabular. El resultado fue un archivo `.csv` donde cada fila representaba una aplicación, identificada por su *hash*, y cada columna correspondía a una categoría de características (permisos, intenciones, etc.),

conteniendo una lista con todos los valores encontrados para esa categoría. Este CSV fue la base sobre la que se construyó el prototipo.

Creación de un modelo prototipo

Con los datos de Drebin ya en un formato manejable, se procedió al diseño del modelo. Inicialmente, se podría haber planteado un modelo monolítico donde todos los módulos estuvieran internamente acoplados. Sin embargo, uno de los objetivos del proyecto era comparar el rendimiento de la red neuronal con el de modelos clásicos de aprendizaje automático (como RandomForest, SVM, etc.). Estos modelos clásicos, en su mayoría, no son capaces de trabajar directamente con listas de cadenas de texto *strings*, ya que operan exclusivamente con datos numéricos. Esto suponía un problema a considerar puesto que la gran mayoría del *dataset* estaba compuesto por ellas.

Esta limitación motivó la primera gran decisión arquitectónica, la cual fue dividir el modelo en dos componentes principales y desacoplados entre si:

1. **Un *embedder* (o preprocesador / «incrustador»):** Un módulo encargado de tomar los datos en bruto de cada muestra (las listas de *strings*) y transformarlos en una representación numérica densa y de longitud fija.
2. **Una cabeza clasificadora:** Un perceptrón multicapa (MLP) que tomaría como entrada el vector numérico generado por el *embedder* y, a través de varias capas «ocultas», reduciría su dimensionalidad hasta obtener la predicción final (una probabilidad para cada una de las clases).

Esta estructura modular ofrecía una ventaja fundamental: la salida del *embedder* podría servir como entrada no solo para la cabeza MLP de la red neuronal, sino también para cualquiera de los modelos clásicos de ML, simplificando enormemente el proceso de comparación.

Durante esta fase de prototipado, el *embedder* se diseñó para realizar todo el preprocesamiento de los datos de entrada de forma interna. Primero, se creaban vocabularios para cada una de las columnas de aquellas características que no fueran de naturaleza numérica. Cuando el modelo recibía una nueva muestra, el *embedder* convertía cada una de estas características (las listas de *strings*) en su correspondiente lista de índices numéricos según

su vocabulario asociado. A continuación, aplicaba un *padding* a cada vector para que todas las secuencias dentro de un mismo lote de datos tuvieran la misma longitud. Finalmente, estos vectores de índices se pasaban a través de una capa de *embedding* que los convertía en vectores densos, los cuales se concatenaban para formar la representación final de la muestra. Desde el principio, el modelo se diseñó pensando en la parametrización de este, permitiendo modificar aspectos como las dimensiones del *embedding* o la estructura de la MLP sin necesidad de alterar el código interno.

Entrenamiento del modelo

Para el entrenamiento del prototipo se empleó una metodología de validación cruzada, concretamente una validación cruzada k-fold estratificada y repetida (*Repeated Stratified K-Fold Cross-Validation*). En lugar de una simple división, la estratificación asegura que en cada «pliegue» (*fold*) del *dataset* se mantenga la misma proporción de muestras benignas y malignas que en el conjunto completo, lo cual es crucial en *datasets* desbalanceados, como es el caso de Drebin. Además, el proceso se repitió varias veces (concretamente, 5 repeticiones de 2 *folds* cada una) para garantizar que los resultados no fueran fruto de una partición afortunada, sino que fueran estadísticamente robustos.

Una decisión importante a lo largo del entrenamiento fue definir la métrica principal a optimizar. Se eligió la sensibilidad o *recall* debido principalmente a que, en un problema de detección de *malware*, el coste de un falso negativo (clasificar una aplicación maliciosa como benigna) es infinitamente mayor que el de un falso positivo (clasificar una aplicación benigna como maliciosa). El *recall* mide precisamente la capacidad del modelo para encontrar todos los positivos reales. Por tanto, todos los modelos fueron entrenados y optimizados para maximizar esta métrica, aceptando cometer más falsos positivos a cambio de minimizar el número de amenazas que pasaban desapercibidas.

Para entrenar los modelos clásicos (RandomForest, XGBoost, k-NN y la Regresión Logística), se utilizó la salida del *embedder* de la red neuronal como entrada. Esto no solo simplificaba el proceso de adaptar los datos para que estos modelos los pudieran entender, sino que además hacía todo el proceso más justo, puesto que todos entrenaban en base al mismo conjunto de datos. Además, usar dicha salida como entrada también les proporcionaba a los modelos los datos ya preprocesados y en un formato numérico que sí podían manejar.

Análisis de datos preliminar

El entrenamiento de los modelos sobre el *dataset* completo de Drebin (más de 127.000 muestras) obtuvo unos resultados iniciales muy interesantes y que afectaron en gran medida al rumbo del resto del proyecto.

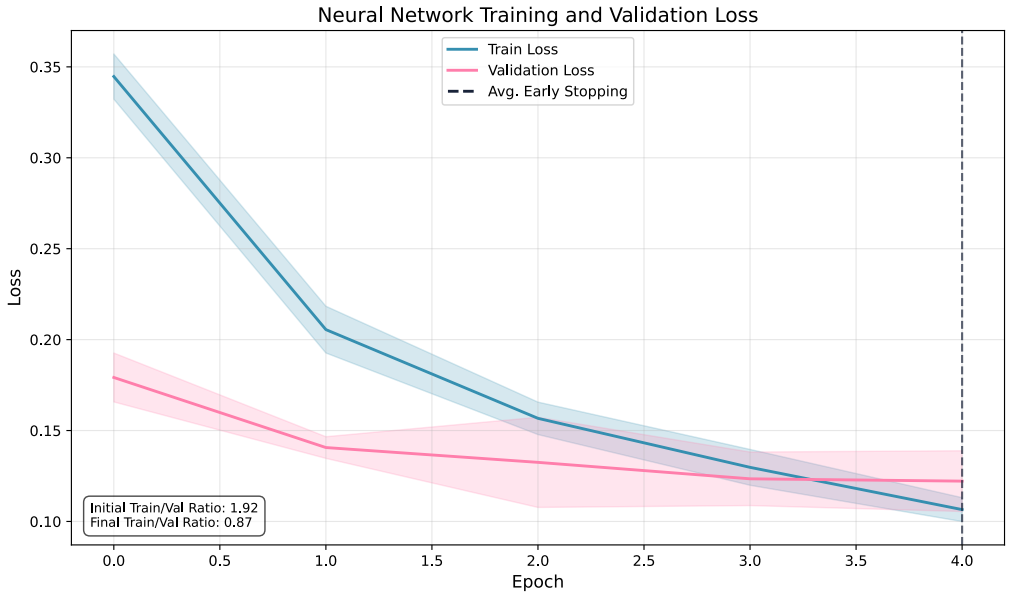


Figura 5.1: Curva de pérdida del entrenamiento del prototipo entrenado con el *dataset* Drebin

La curva de entrenamiento (Figura 5.1) de la red neuronal mostró un comportamiento muy positivo. Se observó una convergencia extremadamente rápida, con la pérdida tanto de entrenamiento como de validación disminuyendo drásticamente y estabilizándose en apenas 4 épocas. Esto indica que el modelo era capaz de aprender los patrones del *dataset* de forma muy eficiente. Además, la pequeña brecha entre ambas curvas sugería que el modelo generalizaba bien y no sufría de sobreajuste alguno. Para disminuir el tiempo de entrenamiento se implementó la famosa técnica de *early stopping*, la cual se activaba de forma temprana en la mayoría de los pliegues, deteniendo el entrenamiento cuando la pérdida de validación dejaba de mejorar, lo que optimizó considerablemente los tiempos.

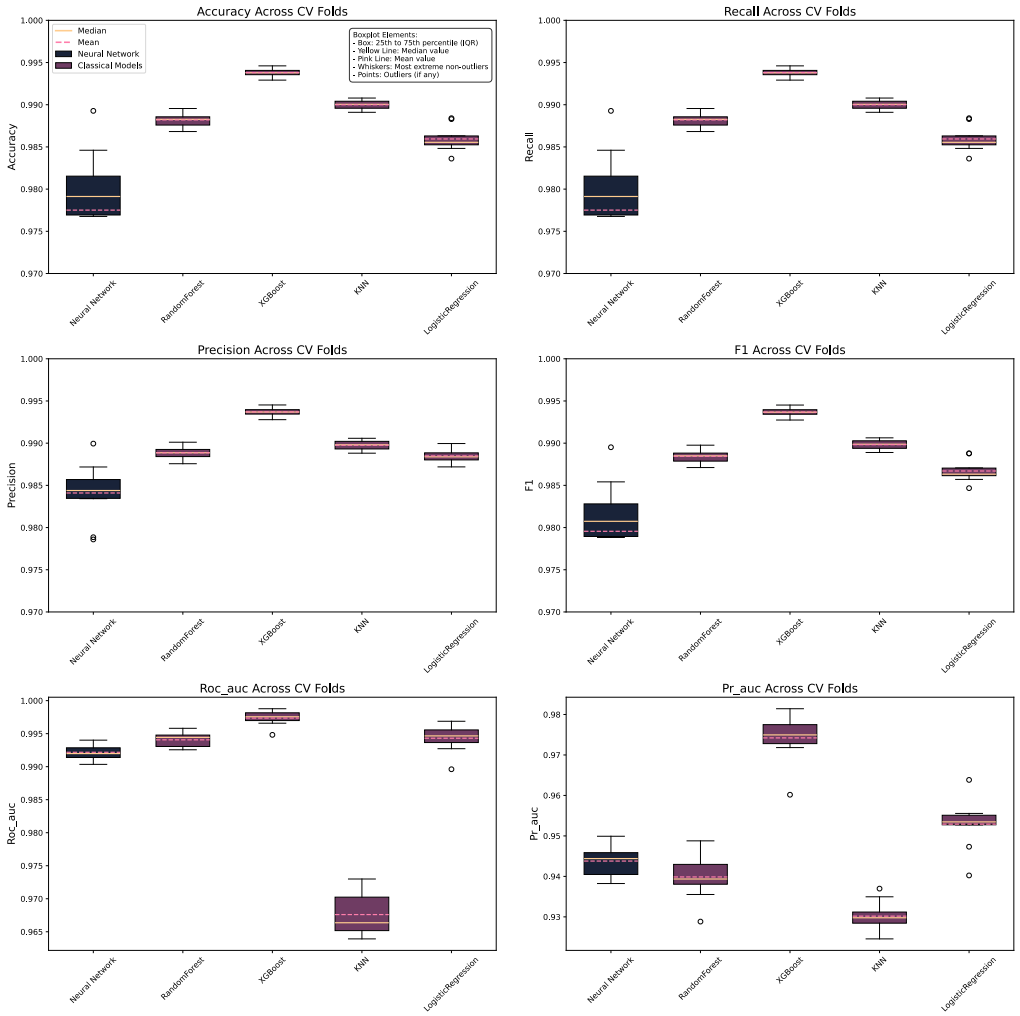


Figura 5.2: Distribuciones de las métricas de evaluación para cada modelo en la validación cruzada inicial.

Al analizar las métricas de rendimiento de todos los modelos (Figura 5.2), la red neuronal obtuvo un *recall* muy respetable, cercano al 97.8 %, pero no fue el mejor modelo. El XGBoost se posicionó como el claro ganador con un 99.25 % de *recall*, seguido de cerca por el k-NN (99 %) y el RandomForest (98.75 %). Cabe destacar también que, la Regresión Logística, a pesar de su simplicidad, alcanzó un admirable 98.5 % y demostró ser el modelo con la menor varianza en sus resultados entre los diferentes *folds*, lo que indica una gran estabilidad.

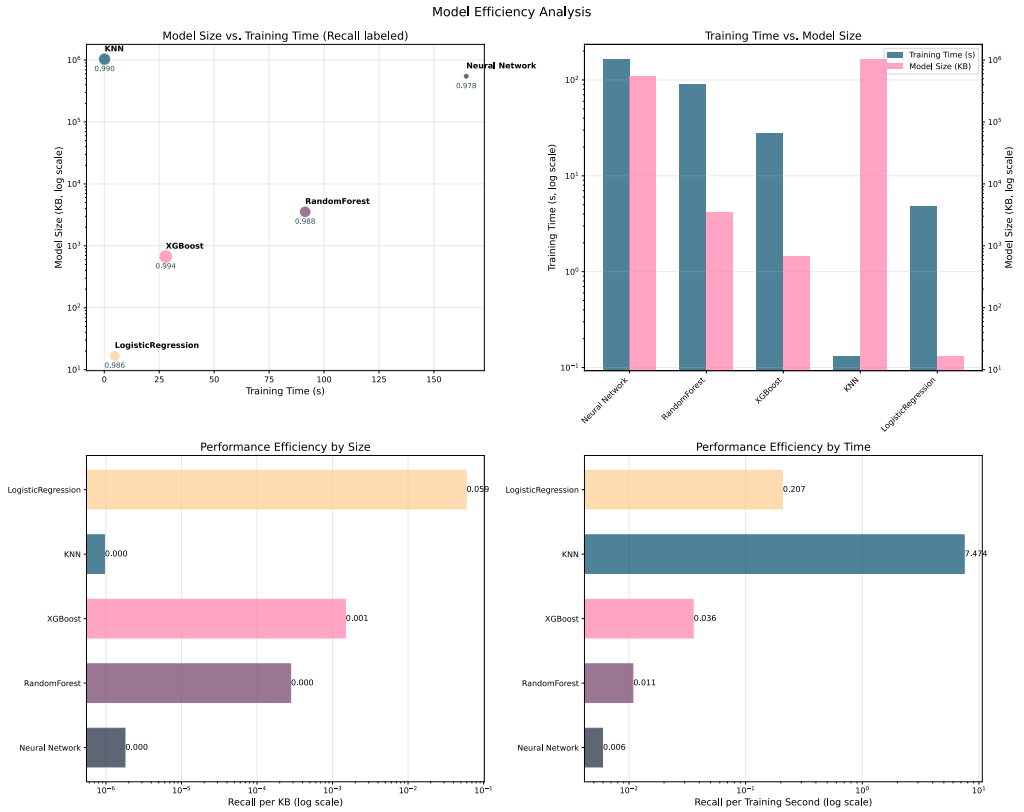


Figura 5.3: Análisis comparativo de la eficiencia en tiempo y espacio de los modelos iniciales.

En términos de eficiencia, el análisis (Figura 5.3) reveló un panorama interesante. La red neuronal fue, con diferencia, el modelo que más tiempo requirió para entrenar (cerca de 3 minutos en media para cada uno de los *folds*). En el otro extremo, la Regresión Logística fue extremadamente rápida y ligera. El modelo k-NN presentó un caso curioso: su tiempo de «entrenamiento» fue casi nulo, ya que su método consiste simplemente en memorizar el *dataset*, pero esto lo convierte en el modelo más pesado en cuanto a uso de disco se refiere. Los gráficos de eficiencia (rendimiento por KB y por segundo) confirmaron estas observaciones, mostrando a la Regresión Logística como el modelo más eficiente en espacio y al k-NN como el más eficiente en tiempo de entrenamiento.

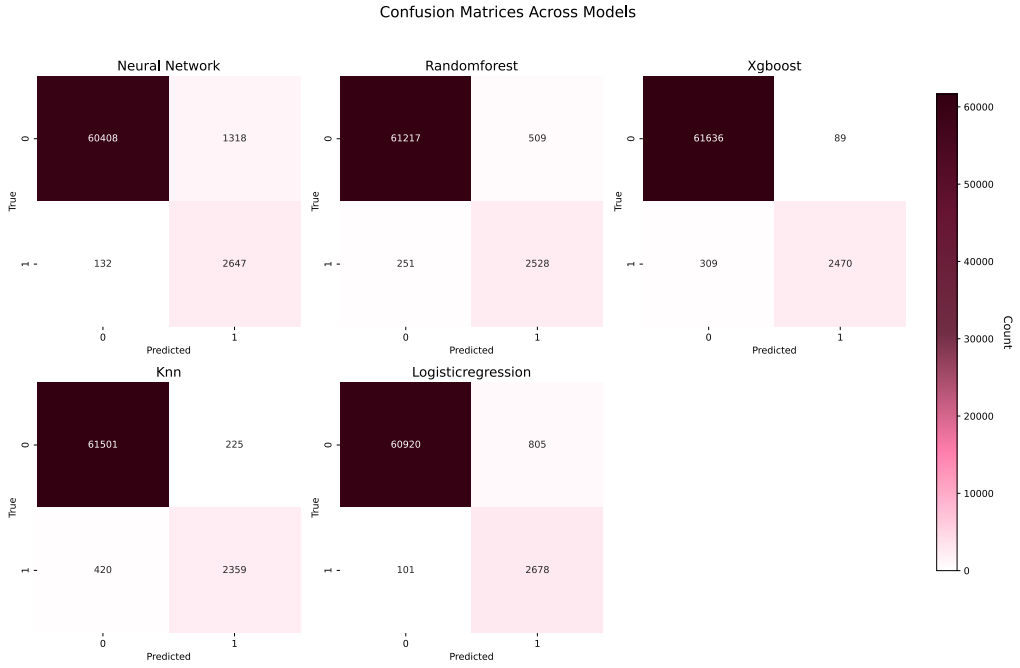


Figura 5.4: Matrices de confusión para cada uno de los modelos entrenados con el *dataset* Drebin.

Las matrices de confusión (Figura 5.4) reflejaron el éxito de la estrategia de optimización basada en el recall. Para la mayoría de los modelos, el número de falsos negativos (esquina inferior izquierda) fue notablemente inferior al de falsos positivos (esquina superior derecha), indicando que los modelos estaban, efectivamente, priorizando la detección de todas las muestras maliciosas. A su vez, en las matrices es posible apreciar el enorme desbalanceo de clases que presenta el *dataset*. A pesar de ello, todos los modelos fueron capaces de obtener resultados muy buenos en todas sus predicciones.

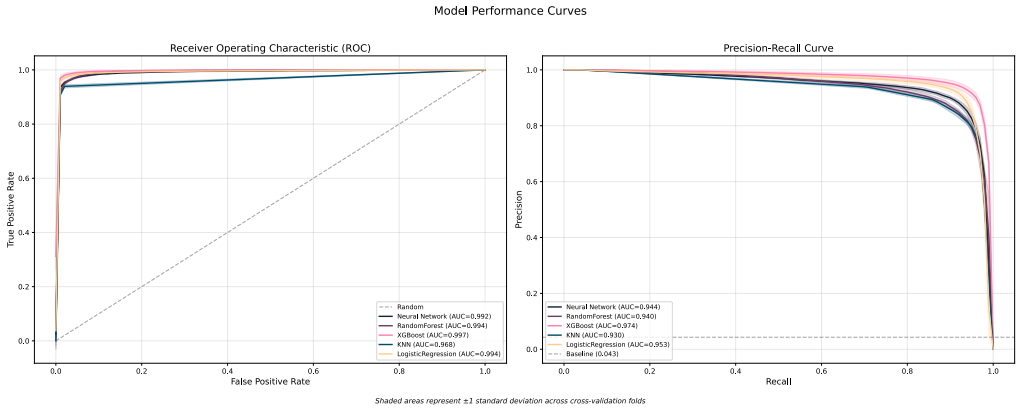


Figura 5.5: Curvas ROC y PR de los modelos iniciales.

Finalmente, las curvas ROC y PR (Figura 5.5) confirmaron el excelente poder discriminativo de todos los modelos, con valores de AUC cercanos al máximo en casi todos los casos. Las curvas PR son bastante más relevantes en aquellos *datasets* fuertemente desbalanceados, pues no se ven tan afectadas por esta medida, mostraron ligeras diferencias, señalando que el modelo XGBoost y a la Regresión Logística, nuevamente, son los modelos más robustos a lo largo de todos los umbrales de decisión posibles.

Inicialmente, estos resultados podrían parecer un poco decepcionantes para la red neuronal, al ser superada por modelos mucho más simples. Sin embargo, esto es ser muy pesimista. Un *recall* del 98 % es un resultado excelente. Pero lo más importante es que, realmente quien permitía que los otros modelos más simple obtuvieran estos resultados altísimos era precisamente la misma red, en concreto el *embedder*. Esto indicaba que el verdadero poder de nuestro enfoque no residía necesariamente en la cabeza clasificadora de la red, sino en su capacidad para aprender una representación de características de alta calidad.

5.2. Extracción de características y *dataset* propio

Los prometedores resultados de la fase de prototipado confirmaron que el enfoque era perfectamente viable. Sin embargo, el modelo seguía siendo un «juguete» de laboratorio, ya que era imposible aplicarlo a una APK real por no conocer el proceso de extracción de características del *dataset* Drebin. Este fue el punto de inflexión del proyecto, donde nació la necesidad de

construir un *dataset* propio con un *pipeline* de extracción transparente y replicable, para así poder crear una herramienta verdaderamente funcional.

Obtención de las aplicaciones maliciosas

El primer paso para crear un *dataset* propio era obtener una gran cantidad de archivos APK que poder analizar, tanto benignos como maliciosos. Tras una breve investigación, se descubrió la existencia del proyecto AndroZoo [26], un inmenso repositorio de aplicaciones de Android destinado a la comunidad científica. AndroZoo proporciona acceso a millones de APKs a través de una sencilla API web, lo que lo convertía en el recurso ideal. Tras solicitar y obtener una clave de API para acceder al mismo, se procedió a interactuar con su catálogo para ver que podía ofrecer este.

AndroZoo ofrece un fichero CSV con metadatos de toda su colección, incluyendo el origen de la aplicación y su número de detecciones en VirusTotal. Se utilizó esta información para realizar un filtrado. Para las aplicaciones benignas, se seleccionaron aquellas provenientes de la Google Play Store, más recientes de 2022 y con 0 detecciones en VirusTotal. Para las maliciosas, se seleccionaron aquellas cuya fuente era VirusShare, un repositorio específico de *malware*. Un *script* de Python fue desarrollado para descargar de forma aleatoria un número igual de aplicaciones de cada grupo, culminando en una colección de 20 000 APKs (10 000 para cada clase), ocupando aproximadamente 200 GB de espacio, una cantidad bastante sólida para crear un buen *dataset*, moderno y balanceado.

Pruebas con Androguard y extracción de características

Con las APKs descargadas, comenzó la fase de experimentación para extraer características relevantes de forma automatizada. La herramienta elegida fue Androguard, por su flexibilidad y su integración nativa con Python. Se buscaron extraer las características que la literatura, especialmente el paper de referencia, señalaba como más importantes. Las características finales seleccionadas fueron:

- **`FILE_SIZE:`** El tamaño del archivo en bytes.
- **`FUZZY_HASH:`** Una huella digital difusa (calculado con la librería `ppdeep`) que permite comparar la similitud entre archivos.

- `ACTIVITIES, SERVICES, RECEIVERS, PERMISSIONS:` Listas de los componentes y permisos declarados en el `AndroidManifest.xml`.
- `API_CALLS:` Una lista de las llamadas a la API que realiza la aplicación, filtradas para incluir únicamente llamadas a librerías externas. Útil para identificar comportamientos potencialmente peligrosos.
- `OPCODE_COUNTS:` Un vector con el recuento de cada uno de los *opcodes* (instrucciones de bajo nivel) de la máquina virtual Dalvik presentes en el código de la aplicación. Útil para determinar que tipo de operaciones se realizan en el código.

Se desarrolló un *script* que iteraba sobre cada una de las 20 000 APKs, aplicaba Androguard para extraer este conjunto de características, y las guardaba de forma incremental en un fichero CSV. El resultado fue un *dataset* propio, moderno y, lo más importante, con un proceso de construcción totalmente conocido y replicable.

5.3. Adaptación y optimización del modelo final

Con el nuevo y mucho más completo *dataset* preparado, se procedió a adaptar el modelo prototipo. Sin embargo, esta etapa reveló una serie de problemas de rendimiento y diseño que no habían sido aparentes con los datos más simples del *dataset* Drebin, obligando a realizar cambios importantes en el modelo y a implementar distintas optimizaciones al mismo.

Adaptación del modelo y problemas imprevistos

El prototipo inicial, aunque diseñado para ser extensible, no estaba para nada preparado para la escala y la diversidad del nuevo *dataset*. Los primeros problemas fueron evidentes: el modelo no era capaz de procesar los nuevos tipos de características, como los vectores de enteros de los `OPCODE_COUNTS` o la secuencia de caracteres del `FUZZY_HASH`. Se tuvo que rediseñar el embedder para manejar estas nuevas entradas, añadiendo una capa GRU para procesar el *fuzzy hash* a nivel de carácter y módulos específicos para los otros tipos de datos.

Tras estas modificaciones, el modelo «funcionaba», pero de manera desastrosa. Los tiempos de entrenamiento se dispararon (de 3 minutos por *fold* a más de 20 minutos para una sola época), el consumo de memoria

RAM también aumentó de manera drástica, requiriendo más de 256 GB, y el uso de la GPU había caido en picado (2-3 %), mientras que la CPU estaba constantemente al 100 %. Y lo más grave: el modelo no aprendía nada. La función de pérdida explotaba a infinito tras un par de iteraciones. Estaba claro que había problemas de diseño fundamentales que la simplicidad del *dataset* Drebin había ocultado.

Optimizaciones y soluciones de los problemas

Para solucionar todos estos problemas fue necesario investigar profundamente el comportamiento del modelo y de todo lo que intervenía en este. El primer culpable identificado fue el manejo de los datos. El nuevo *dataset*, a pesar de tener muchas menos muestras, era casi 20 veces más pesado que el *dataset* Drebin (5 GB ahora frente a unos 200 MB), esto se debía principalmente a que las diferentes listas de características ahora eran mucho más largas, pasando de unos 20 elementos por cada lista a cerca de 19 000 en algunos casos. Se descubrió que el preprocesamiento «al vuelo» de estas listas dentro del modelo era un cuello de botella enorme. Esto hacía que para cada muestra se recalcularan una gran cantidad de datos, resultando en un proceso altamente ineficiente. La solución a esto fue sacar todo el preprocesamiento fuera del modelo y darle los datos ya procesados al modelo directamente, saltando este paso completamente. Además, una interacción inesperada entre Pandas y NumPy estaba también empeorando el rendimiento de forma drástica. Pandas trabaja con NumPy internamente, pero, las columnas del *DataFrame* que contenían listas de números se estaban tratando como listas de objetos de Python normales, no como *arrays* de memoria contigua de NumPy. Esto impedía que NumPy vectorizara las operaciones sobre dichas listas y causaba un cuello de botella enorme en la CPU. La solución fue asegurarse de convertir explícitamente todas las listas internas en *arrays* de NumPy y luego en matrices.

Este cambio redujo drásticamente los tiempos de entrenamiento, pero los problemas de memoria y de aprendizaje persistían. La causa del uso desmesurado de memoria se encontraba en cómo PyTorch manejaba la paralelización de la carga de datos en Windows. Debido a cómo funciona el sistema operativo, cuanado un nuevo proceso se lanza a partir de un proceso padre, este tiene que copiar todos sus datos al hijo, lo que estaba obligando a copiar el *dataset* entero en cada uno de los procesos hijos. La solución fue optimizar el objeto `Dataset` de PyTorch para que solo manejara las matrices de NumPy y reducir a su vez el número de procesos paralelos que se necesitaban. Finalmente, el problema del «gradiente explosivo» que impedía

al modelo aprender se solucionó con un paso que había sido obviado: el escalado de las características numéricas. El enorme rango de valores nuevos que tomaban algunas de las características, como el `FILE_SIZE`, estaban desestabilizando por completo el entrenamiento.

Con estos grandes problemas resueltos, se añadieron un par de optimizaciones finales: un planificador de la tasa de aprendizaje (*scheduler*) para ajustar el *learning rate* dinámicamente y el uso de una técnica conocida como precisión mixta (*mixed precision*), la cual permitía a la GPU realizar ciertas operaciones con tipos de datos de menor precisión para acelerar el cómputo. Tras todo este proceso de optimización y solución de problemas, el tiempo de entrenamiento para todo el *dataset* se redujo a unos impresionantes 3 minutos y ahora el modelo empleaba correctamente todos los recursos del sistema.

Búsqueda de hiperparámetros óptimos

Para exprimir al máximo el rendimiento del modelo, se utilizó la librería Optuna para realizar una búsqueda automática y eficiente de los mejores hiperparámetros. En lugar de una búsqueda a ciegas, Optuna emplea algoritmos de búsqueda bayesiana para explorar de forma más inteligente el espacio de hiperparámetros, mejorando la calidad del conjunto de hiperparámetros que se encuentra y reduciendo el tiempo que se emplea en ello. Se definieron los rangos de búsqueda para los hiperparámetros más importantes, siendo estos: la tasa de aprendizaje, la estructura de las capas ocultas, la dimensión del *embedding*, el tamaño del lote de datos a procesar en cada iteración (*batch size*), el *dropout* y el *weight decay*.

Tras el proceso de optimización, se identificaron dos configuraciones bastante prometedoras. La primera era un modelo más pequeño y rápido (*learning rate* de 10^{-3}, dimensión de *embedding* de 64 y una capa oculta de 64 neuronas con un *batch size* de 64), que entrenaba en unos 3 minutos y alcanzaba un *recall* del 98 % pesando cerca de 1 GB. La segunda era un modelo mucho más grande y lento (learning rate de 10^{-3}, dimensión de *embedding* de 256, 2 capas de 256 y 16 neuronas respectivamente y un *batch size* de 16), pero con una mayor capacidad para aprender. Dado que su rendimiento era similar al del modelo pequeño con el *dataset* actual, y por la comodidad y rapidez del entrenamiento, se eligió la primera configuración como el modelo final para el resto de análisis. Aunque cabe destacar que la segunda configuración podría ser de gran utilidad si se la entrenara con más datos.

Cuantización del modelo

Como experimento adicional, se investigó la posibilidad de reducir el tamaño del modelo final mediante técnicas de cuantización post-entrenamiento, con la idea de hacerlo viable para dispositivos móviles y con recursos limitados. Se probaron los métodos de cuantización dinámica y estática que ofrece PyTorch. Desafortunadamente, la cuantización dinámica apenas redujo el tamaño, ya que el 99 % del peso del modelo reside en las capas de *embedding*, que esta técnica no puede comprimir por defecto. Por otro lado, la cuantización estática sí lograba reducir el tamaño del modelo en un factor de 4, pero a costa de destruir por completo su rendimiento predictivo. Por tanto, se concluyó que la cuantización simple no era una vía factible para este modelo, y que enfoques más avanzados quedarían como una posible línea de trabajo futura.

Análisis de los resultados

Una vez optimizado el modelo de red neuronal y entrenado con la mejor configuración de hiperparámetros, se volvió a realizar una evaluación comparativa completa contra los modelos clásicos, incluyendo esta vez un modelo SVM.

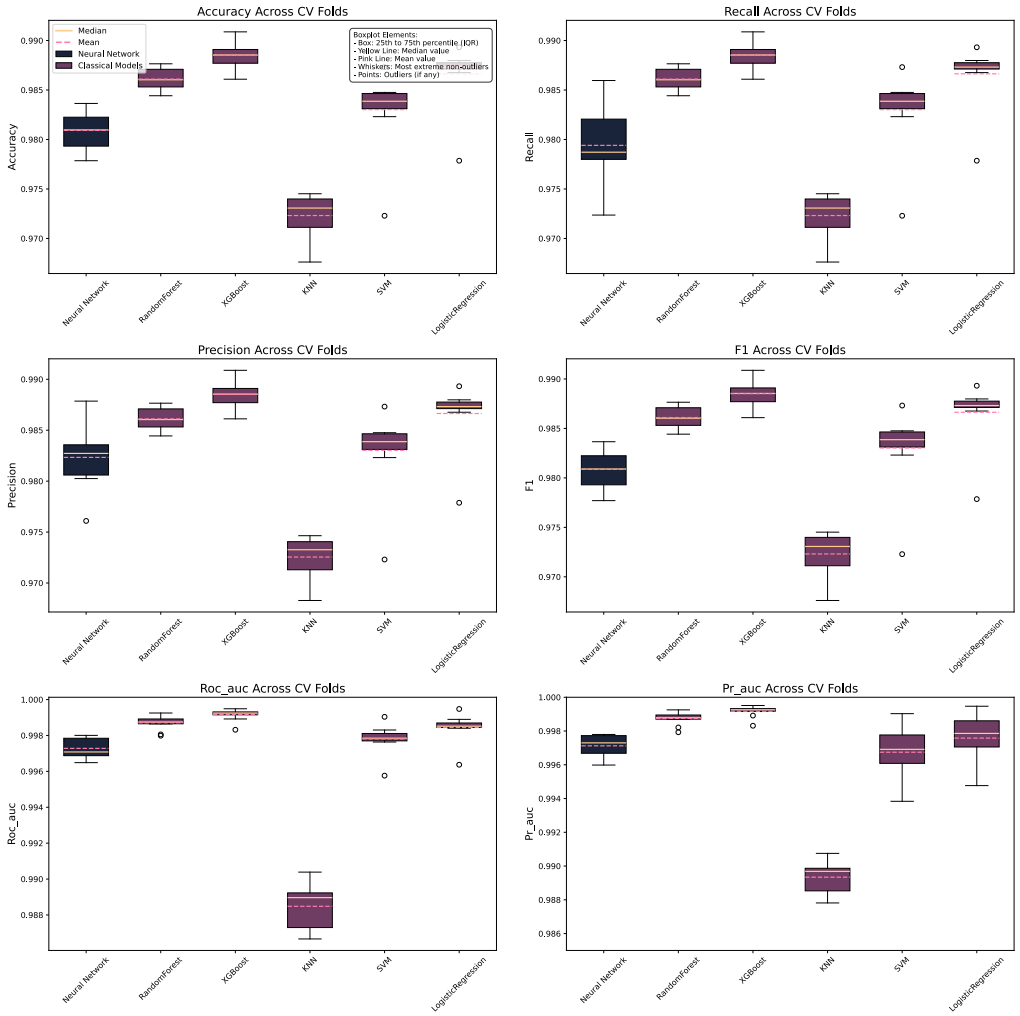


Figura 5.6: Distribuciones de las métricas de evaluación para cada modelo en la validación cruzada final.

Los resultados, mostrados en la Figura 5.6, revelan una mejora sustancial en todos los modelos gracias al nuevo *dataset* y al *embedder* optimizado. Esta vez, la red neuronal muestra un rendimiento mucho más alto y estable, con todas sus métricas rondando el 98-99 %. El mejor modelo en términos de *recall* y F1-Score vuelve a ser XGBoost, seguido muy de cerca por la Regresión Logística y RandomForest, demostrando la increíble eficacia de los modelos clásicos cuando se les alimenta con características de alta calidad.

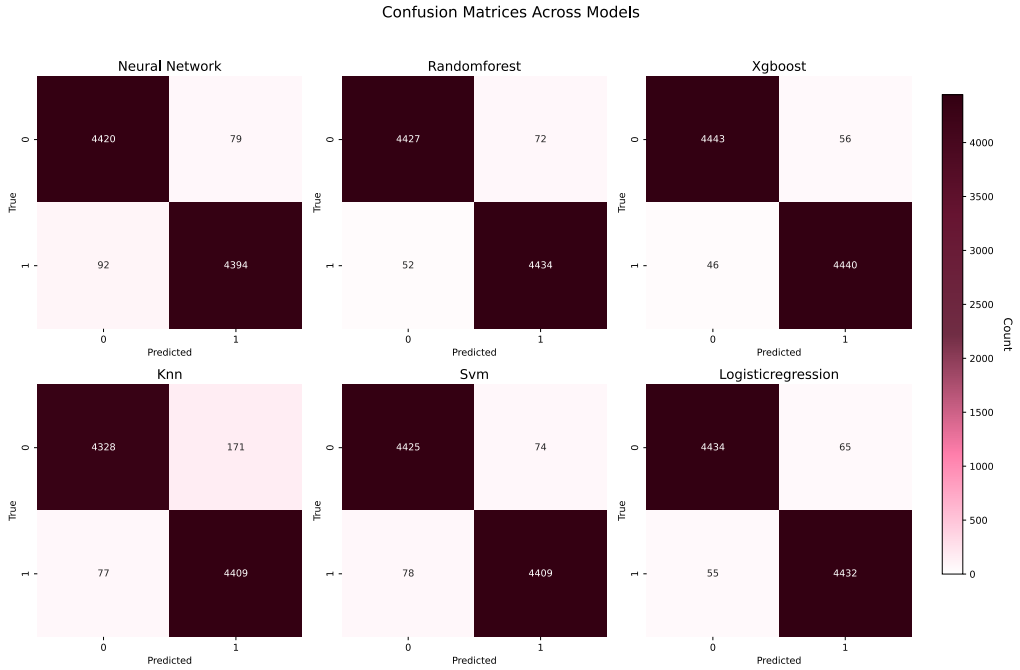


Figura 5.7: Matrices de confusión para cada uno de los modelos entrenados con el dataset final.

Las matrices de confusión (Figura 5.7) confirman visualmente este excelente rendimiento. Todos los modelos son capaces de diferenciar casi perfectamente entre las dos clases, y se mantiene la tendencia a cometer menos falsos negativos que falsos positivos, validando el enfoque en el *recall* durante el entrenamiento.

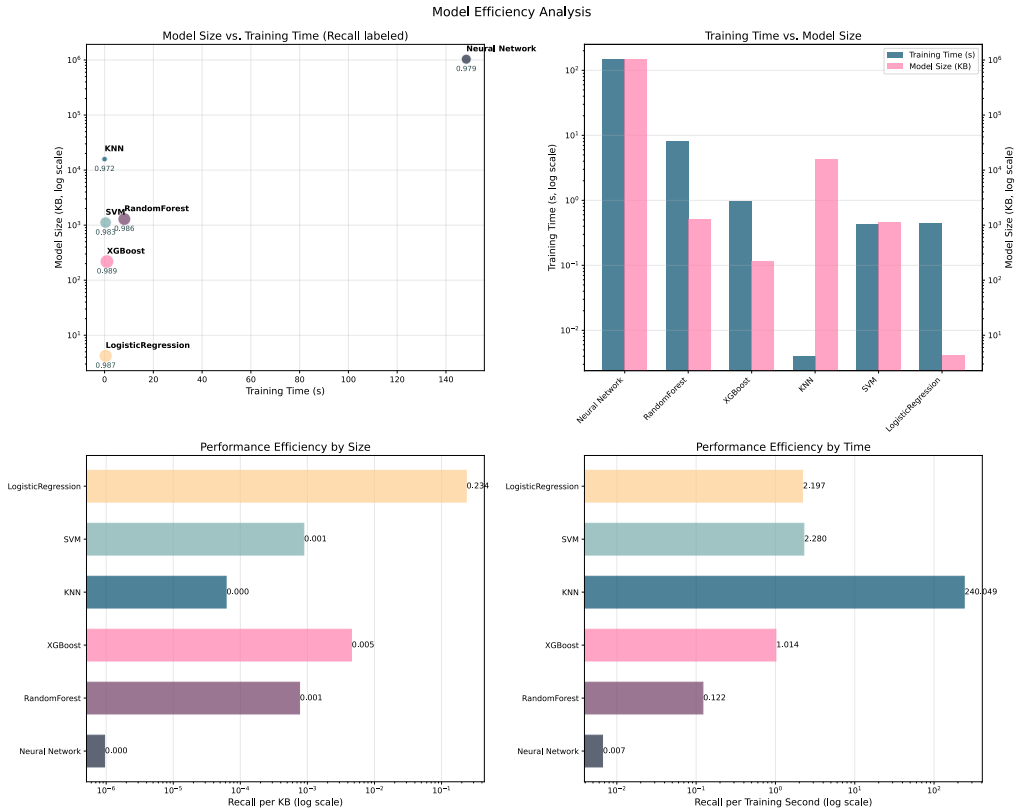


Figura 5.8: Análisis comparativo de la eficiencia en tiempo y espacio de los modelos finales.

En cuanto a la eficiencia (Figura 5.8), el panorama se mantiene similar al visto durante la fase de prototipado, con el añadido de que ahora las diferencias son un poco más marcadas. Los modelos clásicos ahora son capaces de entrenar en menos de 10 segundos gracias a recibir los datos ya procesados. Finalmente, la red neuronal se consolida como el modelo más pesado y lento de entrenar, ocupando la esquina superior derecha del gráfico de dispersión. El resto de gráficos son levemente diferentes a los ya vistos en la fase inicial pero su significado se mantiene igual. La Regresión Logística es el modelo más liviano y por ello, el más eficiente en cuanto a rendimiento por KB. Por otro lado, el k-NN se mantiene como el modelo más eficiente en cuanto a rendimiento por segundo de entrenamiento debido a lo simple que es este.

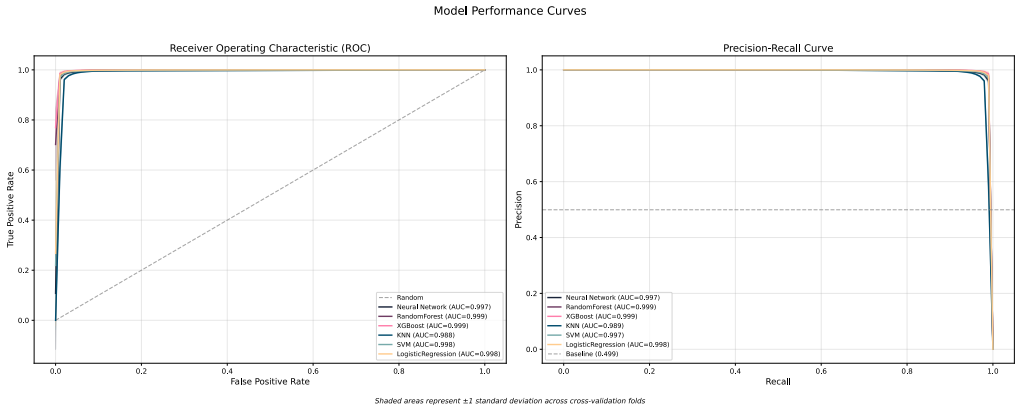


Figura 5.9: Curvas ROC y PR de los modelos finales.

Finalmente, las curvas ROC y PR de la Figura 5.9 son casi perfectas para todos los modelos. Los valores de AUC, tanto para ROC como para PR, son altísimos, lo que indica que todos los clasificadores son extremadamente buenos distinguiendo las clases y mantienen una alta precisión a lo largo de todos los posibles umbrales de decisión. En conclusión, el modelo final y el *pipeline* de datos han sido un éxito rotundo.

5.4. Interpretabilidad del modelo

Con unas métricas de evaluación tan buenas, queda claro que los modelos son muy eficaces, pero no sabemos por qué. Funcionan como «cajas negras». Esta sección busca «abrir» esas cajas para entender en qué se fijan los modelos para tomar sus decisiones, utilizando para ello el framework SHAP.

Análisis de las predicciones mediante SHAP

El análisis de interpretabilidad se realizó con la librería SHAP, que permite calcular la contribución de cada característica a una predicción concreta. Se optó por un enfoque agnóstico al modelo (KernelExplainer), lo que permitió analizar todos los clasificadores (la MLP de la red y los modelos clásicos) con la misma metodología, utilizando siempre como entrada la salida del embedder. Dado que el embedder transforma las características originales en un vector, se agregaron los valores SHAP de las dimensiones correspondientes a cada característica original para obtener una explicación más intuitiva.

Antes de proceder a la explicación es importante entender el siguiente detalle, dado que nuestro datos de entrada provienen del *embedder*, su significado real se ha perdido al pasar por este, dicho de otra manera, en el análisis de características no numéricas, un valor SHAP «alto» o «bajo» no tiene una interpretación directa (no significa que una lista sea más larga o corta o, contenga elementos mayores o menores), sino que indica que el valor de esa característica se mapea a una región u otra del espacio de *embeddings* que el modelo asocia con una clase. Además, los análisis SHAP describen en qué se fija el modelo, lo cual no implica necesariamente una relación causal en el mundo real [3].

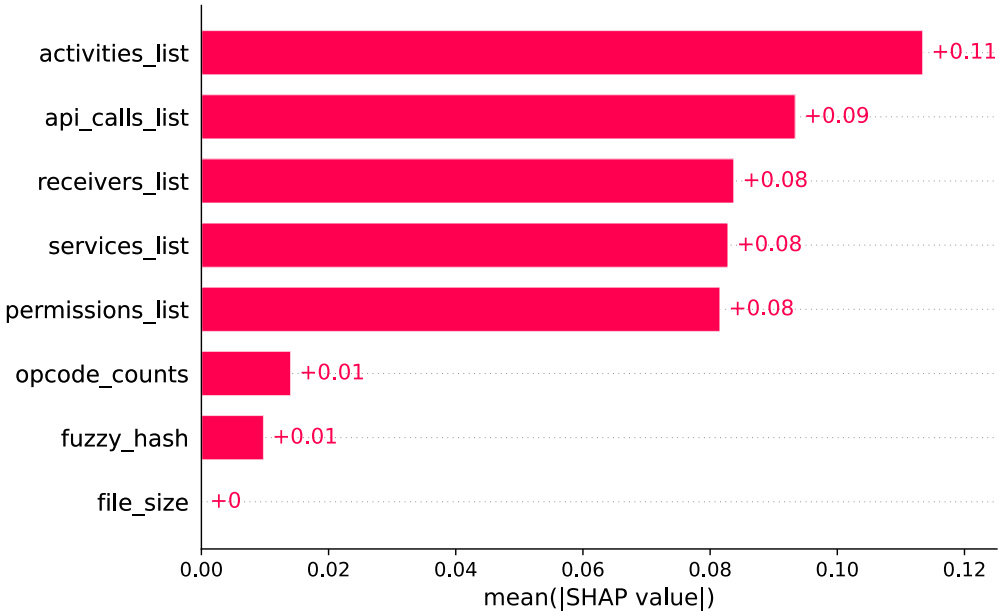


Figura 5.10: Importancia global de las características para el modelo de Red Neuronal, según el valor medio de SHAP.

La Figura 5.10 muestra la importancia global de las características para la red neuronal. Claramente, la característica más influyente es `activities_list`, seguida de cerca por `api_calls_list`, `receivers_list`, `services_list` y `permissions_list`. Esto tiene mucho sentido, ya que los componentes y las API que usa una aplicación son los mejores indicadores de su comportamiento real. Características como el `file_size` o el `fuzzy_hash` tienen un impacto casi nulo, lo cual es un hallazgo interesante principalmente porque estas características eran las que el *paper* de referencia indicaba como las más influyentes en el modelo que ellos entrenaron. Una de las posibles explicaciones del porqué de este comportamiento diferente podría ser el hecho

de que ninguna de estas dos características pasa por el embedder, lo cual implica que estas características no estarán necesariamente agrupadas con el resto. Esto junto con el hecho de que su contribución es mucho menor (tienen menos valores en el vector de salida del *embedder* que el resto de características) pueden ser una de las razones detrás de esta discrepancia.

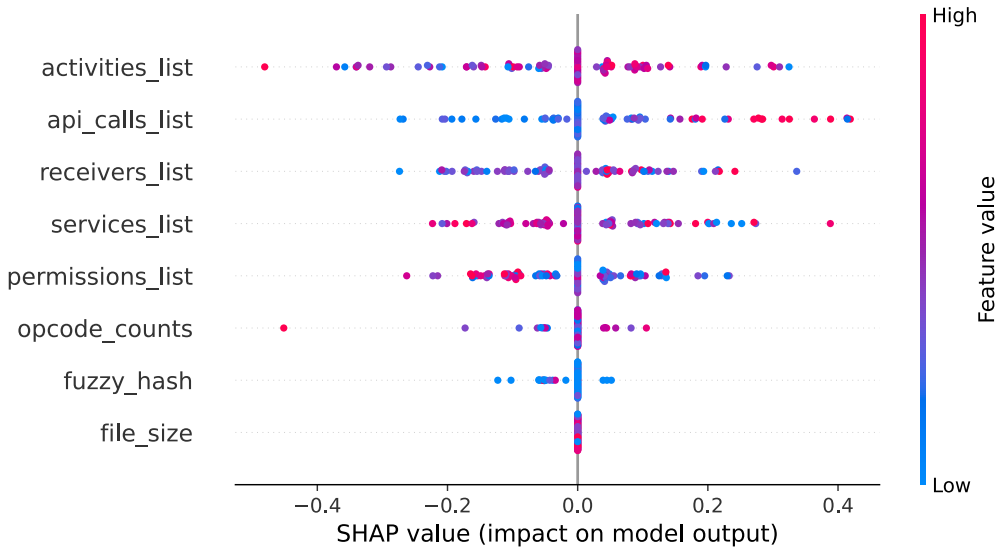


Figura 5.11: Gráfico beeswarm de SHAP para la Red Neuronal, mostrando el impacto de cada característica en las predicciones individuales.

El gráfico *beeswarm* (Figura 5.11) ofrece una visión más detallada. Cada punto es una predicción, y su posición en el eje X indica cómo ha afectado esa característica a la probabilidad de ser *malware*. El color indica si el valor de la característica era «alto» o «bajo» (en el espacio de *embeddings*). Podemos observar una clara separación: para las características más importantes, los valores «altos» (rojos) tienden a empujar la predicción hacia la derecha (mayor probabilidad de ser *malware*), mientras que los «bajos» (azules) la empujan hacia la izquierda. Esto sugiere que el *embedder* ha aprendido a mapear las características asociadas al *malware* a una región específica del espacio.

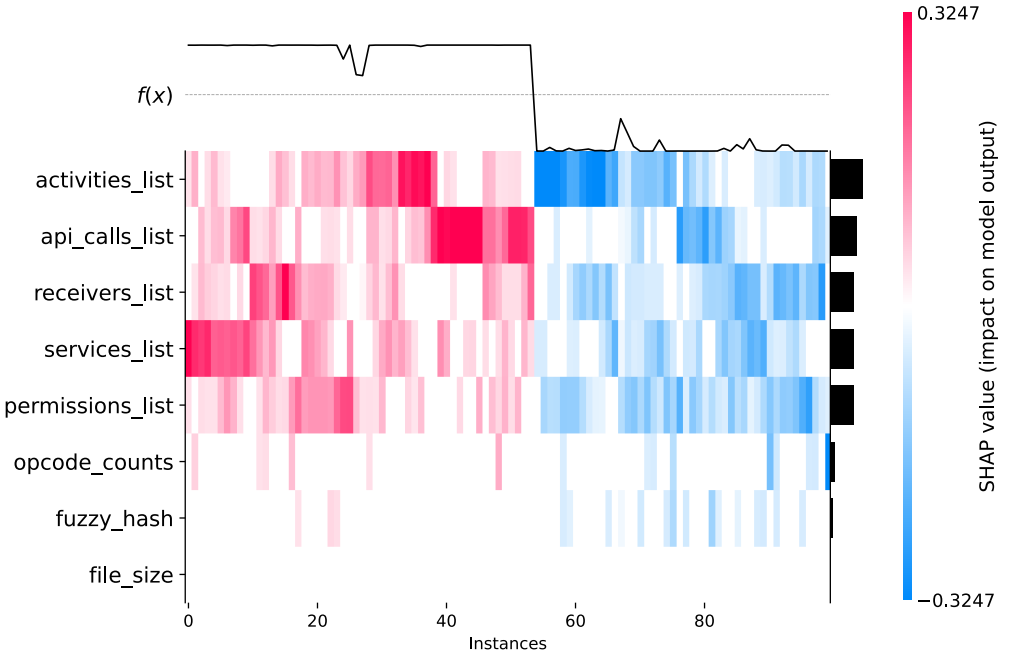


Figura 5.12: Gráfico heatmap de SHAP, mostrando las contribuciones de las características para un conjunto de instancias.

Finalmente, el gráfico de calor (Figura 5.12) muestra un análisis local para un lote de muestras. Cada columna es una instancia, y los colores indican el impacto de cada característica en esa predicción concreta. Se puede observar cómo, para las muestras de la izquierda (clasificadas como *malware*), múltiples características (en rojo) contribuyen positivamente a esa decisión, mientras que para las de la derecha (benignas), son las contribuciones negativas (en azul) las que dominan.

Al realizar este mismo análisis para la Regresión Logística y XGBoost, se observaron patrones muy similares en la importancia de las características, lo que indica que todos los modelos están aprendiendo a fijarse en las mismas «pistas». Esto refuerza la idea de que el *embedder* está haciendo gran parte del trabajo pesado, proporcionando a todos los clasificadores un conjunto de características muy bien segregado y del cual es fácil aprender en que fijarse.

Análisis del *embedder* y el espacio de características

Dado que el *embedder* es la base de la que beben todos nuestros modelos, es importante comprobar si está haciendo bien su trabajo. Si la hipótesis es correcta, el *embedder* debería estar aprendiendo a proyectar las distintas

aplicaciones en un espacio de alta dimensionalidad donde las muestras maliciosas y benignas estuvieran en regiones distintas de dicho espacio, formando cúmulos claramente separables. Para visualizar este espacio, se utilizó la técnica de reducción de dimensionalidad UMAP.

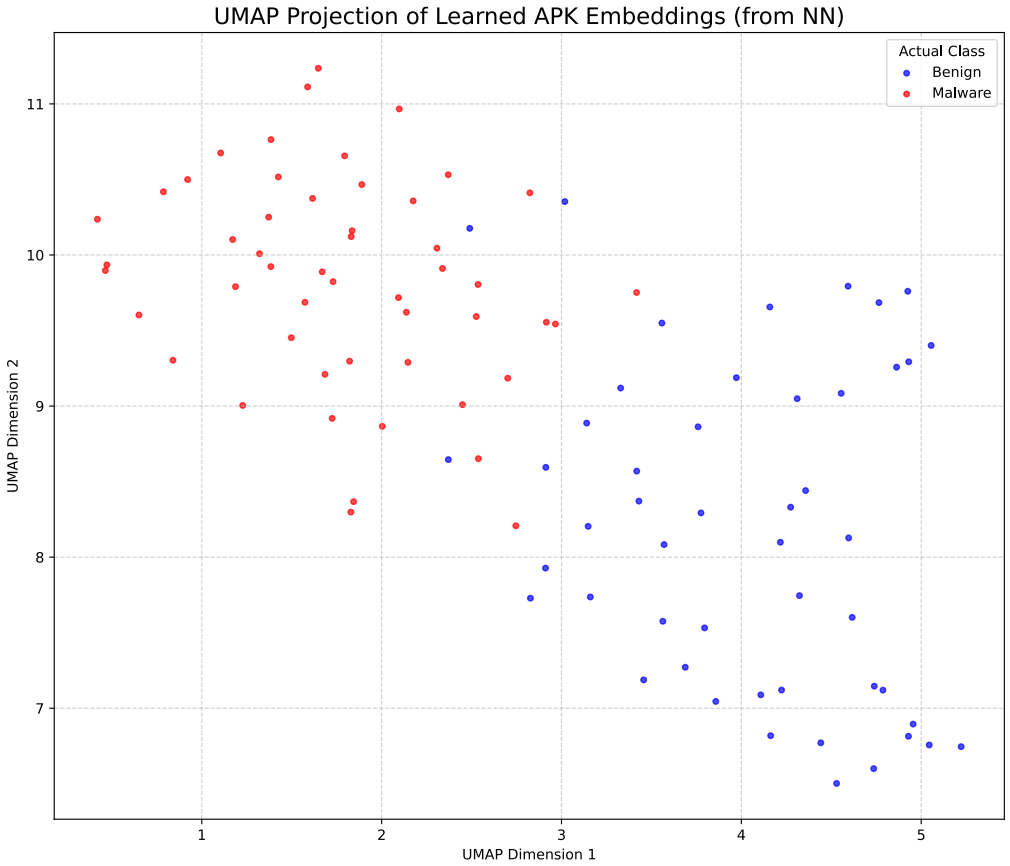


Figura 5.13: Proyección 2D con UMAP del espacio de características aprendido por el *embedder*.

El resultado, como se puede observar claramente en la Figura 5.13, es impresionante. La proyección 2D del espacio de *embeddings* muestra una separación casi perfecta entre las dos clases. Las aplicaciones de *malware* (en rojo) forman un cúmulo denso y bien definido en una región del espacio, mientras que las aplicaciones benignas (en azul) se agrupan en otra región completamente distinta. Esta visualización confirma que el *embedder* ha aprendido a transformar correctamente las relaciones complejas que presentan las distintas características estáticas en una representación donde la clasificación se vuelve una tarea mucho más sencilla. En esencia, el *embedder* está haciendo la mayor parte del trabajo, lo que explica por qué incluso

modelos lineales simples como la Regresión Logística pueden alcanzar un rendimiento tan alto.

5.5. Desarrollo de la aplicación web

Como objetivo final del proyecto, se desarrolló una demostración técnica en forma de aplicación web. El propósito de esta aplicación es permitir a un usuario interactuar con los clasificadores entrenados, subir sus propios archivos APK y obtener no solo una predicción, sino también una explicación de cómo se ha llegado a ella, aumentando así la transparencia y la credibilidad del sistema.

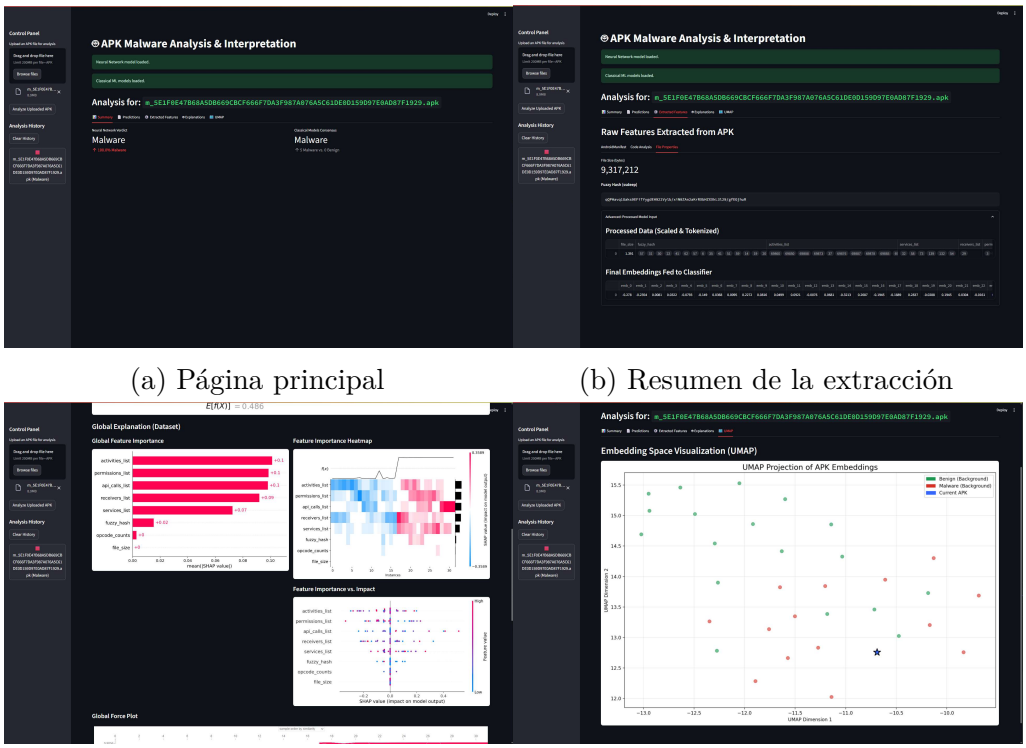
Creación de la web

Para facilitar el desarrollo y la integración con el código Python existente, se eligió Streamlit como *framework* web. A diferencia de otros *frameworks* más complejos, Streamlit está diseñado para transformar *scripts* de datos y modelos de IA en aplicaciones interactivas con un mínimo esfuerzo, lo que lo convertía en la herramienta perfecta para este propósito.

La aplicación se diseñó con una interfaz sencilla y minimalista. Un menú lateral permite al usuario subir un archivo APK y ver un historial de los análisis realizados durante la sesión. La página principal se organiza en pestañas que muestran:

- Un resumen de la predicción actual (Figura 5.14a).
- Los detalles de las características extraídas de la APK (Figura 5.14b).
- Un análisis de interpretabilidad con gráficos de SHAP para el modelo seleccionado (Figura 5.14c).
- La visualización del espacio de *embedding* con UMAP, mostrando dónde se sitúa la aplicación analizada (Figura 5.14d).

De esta forma, la aplicación no solo da un resultado, sino que también educa al usuario sobre el proceso interno del modelo.



(a) Página principal (b) Resumen de la extracción

(c) Análisis de interpretabilidad (d) Análisis del espacio de *embeddings*

Figura 5.14: Aplicación de demostración del modelo

Dockerización y despliegue

Para facilitar la prueba y la distribución de la aplicación, se decidió empaquetarla en un contenedor de Docker. Esto permite que cualquier persona con Docker instalado pueda ejecutar la aplicación con un único comando, sin preocuparse por instalar Python o cualquiera de las numerosas dependencias del proyecto. El `Dockerfile` se encargó de preparar el entorno utilizando Poetry para instalar los paquetes, y de iniciar el servidor de Streamlit, exponiendo el puerto correspondiente.

El despliegue final se realizó en un servidor de la universidad. El proceso consistió en conectarse a la red del centro mediante una VPN, acceder a la máquina remota por SSH, transferir los archivos del proyecto mediante SCP y, finalmente, ejecutar el contenedor de Docker. Esto permitió que la aplicación fuera accesible desde dentro de la red de la universidad[1], completando así el ciclo de vida del proyecto desde la investigación hasta el despliegue de una herramienta funcional.

[1]Disponible en este dirección: http://10.168.168.34:8501/

6. Trabajos relacionados

Este capítulo presenta una revisión de la literatura y los trabajos existentes que son relevantes para el proyecto. El objetivo de este estado del arte es doble: por un lado, fundamentar las decisiones tomadas durante el desarrollo, mostrando que se basan en estudios y técnicas usadas actualmente en el campo; y por otro, posicionar este trabajo dentro del panorama de la investigación en detección de *malware*, destacando tanto sus similitudes con otros enfoques como sus aportaciones propias. La revisión se ha estructurado en tres áreas: una visión general acerca del *malware*, un análisis de otras investigaciones y papers que han contemplado métodos de detección estática de *malware* Android y un último apartado hablando acerca de algunos *datasets* bastante comunes en este ámbito.

6.1. Primer contacto con el *malware*

Para poder abordar un problema tan complejo como la detección de *malware*, la primera fase del proyecto consistió en una investigación para asentar una buena base teórica acerca del mundillo del *malware* y todo el panorama que este cubre. Los trabajos presentados en esta sección sirvieron como una introducción al campo de la ciberseguridad, explicando qué es el *malware*, sus diferentes tipos y las técnicas que se emplean para su análisis.

A review on malware analysis for IoT and Android system

El trabajo de Yadav y Gupta (2022) [63] ofrece una revisión exhaustiva y muy accesible del panorama del *malware*, con un enfoque particular en los sistemas que usan Android y los dispositivos IoT. El artículo comienza

introduciendo los conceptos básicos de la seguridad informática y explora las vulnerabilidades comunes que los atacantes explotan en estos entornos. Una de sus aportaciones más valiosas es la descripción detallada del «plan de explotación» que suelen seguir los atacantes, lo que ayuda a comprender la motivación y las fases de un ciberataque.

Los autores evalúan tanto el análisis estático como el dinámico, concluyendo que una estrategia de detección óptima debería, idealmente, combinar ambos enfoques. Además, discuten otras tecnologías de defensa como los Honeypots o los Sistemas de Detección de Intrusos (IDS).

Este artículo fue uno de los más importantes durante la fase inicial de investigación. Puesto que, proporcionó una visión amplia y general sobre el *malware* y sus métodos de análisis, sirviendo como una excelente introducción al tema.

A Static Approach for Malware Analysis: A Guide to Analysis Tools and Techniques

En este artículo, Nair et al. (2023) [49] se centran exclusivamente en el análisis estático, presentándolo como la primera línea de defensa contra el *malware*. La publicación funciona como una guía práctica que repasa las diferentes técnicas y herramientas disponibles para inspeccionar archivos sospechosos sin necesidad de ejecutarlos. Se detallan los distintos tipos de *malware* (troyanos, gusanos, *rootkits*, etc.) y abordan algunos de los desafíos y problemas a los que se enfrentan hoy en día los analizadores de *malware*, como por ejemplo, las técnicas de ofuscación de código, las cuales permiten «cifrar» el código de un programa de tal manera que dificulte su análisis.

El estudio subraya la importancia de los analizadores, el análisis de cadenas de texto y el *pattern matching* como métodos fundamentales del análisis estático. También destaca la necesidad de que los analistas sepan cómo manejar archivos empaquetados (*packed*) para poder examinar su contenido real, esto sería equiparable a cómo funcionan las APK en Android.

Este trabajo fue de gran utilidad para profundizar en la metodología central del proyecto. Mientras que otros estudios hablan del análisis estático de forma general, este artículo proporciona una visión más detallada de las técnicas específicas y los problemas prácticos, como la ofuscación, que se han de tener en cuenta a la hora de diseñar un sistema que sea capaz de extraer características de archivos maliciosos.

6.2. Análisis estático de *malware* en Android

Una vez sentadas las bases, la investigación se centró en el nicho específico de este proyecto: la aplicación de técnicas de análisis estático y, más concretamente, de inteligencia artificial, para la detección de *malware* en el ecosistema de Android. Los trabajos de esta sección fueron los que más influenciaron la dirección que se ha tomado en el trabajo y son principalmente la razón por la cual se siguió adelante con el mismo.

A Method for Automatic Android Malware Detection Based on Static Analysis and Deep Learning

El trabajo de İbrahim et al. (2022) [64] ha sido la principal fuente de inspiración y el punto de partida de este proyecto. El artículo aborda exactamente el mismo problema que el que se intenta resolver en este trabajo: la creación de modelo de aprendizaje automático capaz de detectar *malware* en Android basado exclusivamente en el análisis estático de dichos archivos. Los autores proponen un método que consiste en recolectar un gran número de características estáticas, incluyendo dos propuestas por ellos mismos, para luego alimentar un modelo de red neuronal construido con la API de Keras.

Para su evaluación, crearon un *dataset* propio con más de 14 000 muestras y realizaron dos experimentos: uno de clasificación binaria (*malware* vs. benigno) y otro de clasificación multiclase (diferenciando entre distintas familias de *malware*). Sus resultados fueron muy prometedores, alcanzando un F_1-*Score* del 99,5 % en la detección binaria, superando a otros trabajos relacionados.

Este *paper* fue la raíz del proyecto. Sirvió para demostrar que era posible alcanzar una precisión alta utilizando únicamente características estáticas. La metodología que describen, incluyendo el uso de Androguard para la extracción de características, fue la base sobre la que se empezó a construir el prototipo planteado. Sin embargo, hay que destacar que, a pesar de ser un muy buen *paper*, el artículo omite ciertos detalles cruciales de implementación, como la arquitectura exacta de la red neuronal o cómo se entrenaron los modelos clásicos con los que se comparan. Esta falta de detalle motivó no solo a replicar su idea, sino a profundizar en estos aspectos, realizando pruebas propias de optimización de hiperparámetros y un análisis comparativo más

transparente y detallado. Además de realizar un análisis de interpretabilidad a los diferentes modelos para aumentar la confianza de estos.

MAPAS: a practical deep learning-based android malware detection system

Kim et al. (2022) [42] proponen en su trabajo MAPAS, un sistema de detección de *malware* con un enfoque muy interesante y orientado a la eficiencia. Al igual que en este caso, utilizan un modelo de *deep learning*, pero, con la diferencia de que en su caso usan una Red Neuronal Convolucional (CNN), para analizar características estáticas, concretamente grafos de llamadas a la API. Sin embargo, la gran diferencia es que no utilizan la CNN como el clasificador final. En su lugar, la emplean únicamente como un extractor de características para descubrir patrones comunes en el *malware*.

La clasificación final la realiza un algoritmo mucho más ligero, que simplemente calcula la similitud entre los patrones de una nueva aplicación y los patrones de *malware* obtenidos mediante la CNN. Gracias a este diseño, afirman que su sistema es mucho más rápido y consume hasta diez veces menos memoria que otras aproximaciones, lo que lo haría viable para ser ejecutado directamente en un dispositivo móvil.

Este trabajo ofreció una perspectiva alternativa muy valiosa sobre otra línea posible de investigación del proyecto. Mientras que en este caso se emplea el *embedder* de la red neuronal para potenciar a modelos clásicos, MAPAS utiliza una CNN para alimentar a un clasificador basado en similitud. Esta filosofía de usar el *deep learning* para el aprendizaje de representaciones y delegar la clasificación final a un modelo más simple es una idea muy poderosa que resuena con las conclusiones propias, reforzando la idea de que una solución mixta parece ser la mejor forma de atacar el problema, sobretodo en el caso en el que se deseara llevar este tipo de analizadores a dispositivos móviles donde la cantidad de recursos disponible es mucho más limitada.

Android mobile malware detection using machine learning: A systematic review

Este artículo de Senanayake et al. (2021) [53] es una revisión sistemática de la literatura que analiza cómo se ha aplicado el aprendizaje automático, y en especial el *deep learning* (DL), a la defensa contra el *malware* en Android. Tras revisar 132 estudios publicados entre 2014 y 2021, los autores concluyen

que hay una tendencia clara a abandonar las reglas manuales y el ML tradicional en favor de los modelos de DL, debido a la capacidad de estos últimos para abstraer características de forma más potente, siendo capaces de combatir mejor contra técnicas de evasión modernas.

El estudio no solo se centra en la detección, sino que también discute las tendencias de la investigación, los principales desafíos y las futuras líneas de trabajo en el campo de la defensa contra el *malware* en Android basada en DL.

The Android Malware Static Analysis: Techniques, Limitations, and Open Challenges

Aunque es un trabajo de 2018, el estudio de Bakour et al. [29] resultó ser una fuente de motivación muy importante. El artículo realiza una revisión exhaustiva de más de 80 *frameworks* de análisis estático para Android, identificando sus técnicas, pero sobre todo, sus limitaciones y los desafíos que quedaban por resolver en aquel momento. Una de sus contribuciones más interesantes es la categorización de las características estáticas en cuatro grupos: basadas en el manifiesto, en el código, en la semántica y en los metadatos de la aplicación.

El estudio concluye con un caso práctico en el que se demuestra que los antivirus comerciales y las herramientas académicas de la época tenían serios problemas para detectar *malware* que utilizaba técnicas de ofuscación. Los autores concluyen que existía una «necesidad urgente» de herramientas más precisas y robustas.

Este *paper* fue bastante útil como motivación para la realización del proyecto pues que, se confirma que, incluso hace pocos años, el campo del análisis estático tenía carencias significativas, validando la necesidad de explorar nuevos enfoques como el presentado en este caso.

Droidmat: Android malware detection through manifest and api calls tracing

El trabajo de Wu et al. (2012) [61] es uno de los primeros ejemplos de un sistema que aplica aprendizaje automático al análisis estático en Android. Los autores presentan DroidMat, una herramienta que extrae características del manifiesto, los mensajes Intent y las llamadas a la API para caracterizar el comportamiento de una aplicación. Posteriormente, utiliza algoritmos de

clustering (k-Means) y clasificación (k-NN) para distinguir entre aplicaciones benignas y maliciosas.

En su evaluación, los autores afirman que DroidMat no solo obtenía una tasa de *recall* superior a la de Androguard (en su versión de 2011), sino que también era significativamente más rápido en su análisis.

Este estudio fue una referencia muy valiosa porque demostraba, ya en 2012, que la idea de combinar la extracción de características estáticas con el aprendizaje automático era un camino viable y prometedor.

6.3. *Datasets* de *malware* para Android

Al igual que todo proyecto de aprendizaje automático, es necesario tener datos de calidad para que estos salgan adelante. Esta sección final revisa los trabajos que crean algunos de los *datasets* más relevantes del campo y las herramientas que se usan para llevar esto a cabo.

Drebin: Effective and explainable detection of android malware in your pocket

El trabajo de Arp et al. (2014) [28] no solo propone un método de detección, sino que también introduce uno de los *datasets* más utilizados en la investigación de *malware* en Android: el *dataset* Drebin. Este conjunto de datos contiene 5.560 muestras de *malware* y más de 123.000 aplicaciones benignas, junto con un conjunto de características estáticas extraídas de cada una de ellas, como permisos, llamadas a API o componentes del manifiesto.

El objetivo original de Drebin era crear un sistema lo suficientemente ligero como para poder ejecutarse directamente en un teléfono. Aunque el artículo no detalla con precisión el proceso exacto para extraer las características de una nueva aplicación, el *dataset* que liberaron ha sido una referencia para la comunidad durante bastante tiempo.

El *dataset* Drebin fue de gran ayuda en este proyecto debido a que su conjunto de características era bastante similar al del *paper* que se utilizó de referencia. Este detalle junto con el hecho de que el *dataset* está fuertemente desbalanceado en cuanto a la clase negativa fueron dos factores muy útiles que permitieron empezar el desarrollo temprano de un prototipo bastante bueno que asentarías las bases del modelo final que se desarrolla a lo largo del trabajo. Cabe destacar que, dado la falta de información acerca de su proceso de creación, también inspiró a la creación de un *dataset* propio.

Dynamic android malware category classification using semi-supervised deep learning

En este trabajo, Mahdavifar et al. (2020) [45] presentan un nuevo *dataset* llamado CICMalDroid2020. Este conjunto de datos es bastante interesante porque incluye más de 17.000 muestras recientes y, a diferencia de Drebin, contiene tanto características estáticas como dinámicas (obtenidas de la ejecución de las aplicaciones). Además, el artículo propone un enfoque de aprendizaje semisupervisado para clasificar el *malware* en diferentes categorías (Adware, Banking, etc.), una técnica muy útil cuando se dispone de pocos datos etiquetados.

Deep ground truth analysis of current android malware

El trabajo de Wei et al. (2017) [60] tiene un objetivo distinto a los demás. En lugar de crear un *dataset* para entrenar modelos, su meta fue crear un conjunto de datos de «verdad fundamental» (*ground truth*) a través de un análisis manual exhaustivo y profundo. Analizaron casi 25.000 muestras de *malware* y las clasificaron manualmente en 71 familias y 135 variedades distintas, documentando en detalle el comportamiento específico de cada una.

El resultado es un recurso de un valor incalculable para entender el ecosistema del *malware* en Android, no desde un punto de vista estadístico, sino cualitativo.

AndroZoo: Collecting Millions of Android Apps for the Research Community

Allix et al. (2016) [26] presentan AndroZoo, que no es un *dataset* en sí mismo, sino un inmenso repositorio de aplicaciones para Android. Se trata de un proyecto en continuo desarrollo que recolecta miles de archivos APK de diversas fuentes, incluyendo la tienda oficial de Google Play, mercados de terceros y colecciones de *malware* como VirusShare. El proyecto pone toda esta colección a disposición de la comunidad investigadora a través de una API propia.

Para cada aplicación, AndroZoo también proporciona los resultados de su análisis por parte de decenas de antivirus comerciales, mediante el uso de VirusTotal lo que permite que uno pueda clasificar cada APK con una etiqueta de «malicioso» o «benigno» con un cierto grado de confianza.

AndroZoo fue, sin duda, una de las herramientas más importantes a la hora de construir el *dataset* propio de este trabajo. Todo el conjunto de datos sobre el que se ha entrenado y evaluado el modelo final de este proyecto se ha creado a partir de miles de muestras, tanto benignas como maliciosas, descargadas directamente del repositorio de AndroZoo. Sin el acceso a esta increíble colección, la creación de dicho *dataset* habría sido muchísimo más difícil sino prácticamente imposible.

7. Conclusiones y Líneas de trabajo futuras

Este capítulo final tiene como objetivo recapitular los hallazgos y resultados obtenidos a lo largo del desarrollo del proyecto, ofreciendo una reflexión crítica sobre el cumplimiento de los objetivos y las lecciones aprendidas. Asimismo, se propondrán diversas líneas de trabajo futuras que podrían servir como continuación de esta investigación, explorando nuevas vías para mejorar, optimizar y ampliar la solución desarrollada.

7.1. Conclusiones

El desarrollo de este proyecto ha culminado en la creación de un sistema completo y funcional para la detección de *malware* en Android mediante análisis estático e inteligencia artificial. A lo largo de este recorrido, se han obtenido una serie de conclusiones técnicas y conceptuales de gran relevancia.

La primera y más evidente conclusión es que **el enfoque propuesto es altamente eficaz**. Todos los modelos entrenados, tanto la red neuronal como los clasificadores clásicos, han demostrado una capacidad de detección muy buena, alcanzando métricas de *recall*, precisión y AUC superiores al 98 % en la mayoría de los casos. Esto confirma de manera rotunda la hipótesis central del proyecto: es perfectamente viable construir un clasificador de alto rendimiento basándose únicamente en las características estáticas extraídas de un archivo APK, sin necesidad de recurrir a técnicas de análisis dinámico, que son más lentas y costosas. El objetivo de obtener un modelo útil y con buen rendimiento se ha cumplido con creces.

Sin embargo, la conclusión más importante y reveladora de este trabajo es el **papel fundamental del *embedder* como motor de ingeniería de características**. Si bien la red neuronal por sí sola es un clasificador muy competente, su verdadero valor radica en su capacidad para aprender una representación numérica densa y significativa a partir de datos de entrada complejos y heterogéneos. La visualización del espacio de *embeddings* mediante UMAP demostró de forma sorprendente que este componente era capaz de separar casi perfectamente las aplicaciones benignas de las maliciosas en un espacio de alta dimensionalidad. Este preprocesamiento inteligente es la razón por la cual modelos mucho más simples, como la Regresión Logística o XGBoost, pudieron alcanzar un rendimiento tan espectacular. El proyecto demuestra en la práctica un principio fundamental del aprendizaje automático: la calidad de los datos y de sus representaciones es, a menudo, más importante que la complejidad del modelo clasificador final. Esto valida la estrategia híbrida como un enfoque extremadamente poderoso.

En tercer lugar, el análisis de interpretabilidad mediante SHAP ha permitido **abrir la «caja negra» de los modelos y validar su proceso de razonamiento**. Los resultados mostraron que todos los clasificadores, de manera consistente, basaban sus decisiones en las características más lógicas: los permisos solicitados, las llamadas a la API y los componentes declarados en la aplicación (actividades, servicios y receptores). Esto no solo aumenta la confianza en las predicciones, sino que también cumple el objetivo de crear un modelo interpretable, demostrando que no estaba aprendiendo de artefactos o sesgos extraños en los datos, sino de indicadores de comportamiento reales.

Desde una perspectiva de ingeniería, otra conclusión importante es que la **optimización y el diseño del *pipeline* de datos son tan críticos como el propio modelo**. La transición del prototipo al modelo final reveló graves cuellos de botella relacionados con el manejo de la memoria y el preprocesamiento de datos, cuya solución requirió bastante trabajo de optimización. Esto subraya la importancia de entender no solo los algoritmos, sino también las herramientas de software (Pandas, NumPy, PyTorch) a un nivel más bajo para construir sistemas eficientes y escalables.

Finalmente, la exploración de la **cuantización del modelo arrojó resultados limitados para esta arquitectura específica**. Se concluyó que las técnicas de cuantización post-entrenamiento no son efectivas cuando la mayor parte del peso de un modelo reside en las capas de *embedding*. Esto supone una conclusión técnica valiosa: para llevar un modelo de estas

características a entornos con recursos limitados, como un dispositivo móvil, se requerirían enfoques más avanzados, como el entrenamiento consciente de la cuantización y con ello, el posible rediseño de la arquitectura.

En resumen, se han cumplido todos los objetivos principales del proyecto. Se ha realizado una investigación del estado del arte, se ha construido un *dataset* propio con un *pipeline* replicable, se ha entrenado y optimizado una red neuronal de alto rendimiento y se ha realizado un análisis comparativo e interpretativo exhaustivo, culminando en una aplicación web funcional. A nivel personal, el proyecto ha permitido adquirir una profunda experiencia práctica en todo el ciclo de vida de un proyecto de IA, desde la investigación y la gestión de datos hasta el despliegue.

7.2. Líneas de trabajo futuras

Todo proyecto de investigación y desarrollo abre nuevas puertas y plantea nuevas preguntas. Este trabajo no es una excepción. A continuación, se detallan varias líneas de trabajo futuras que podrían dar continuidad y expandir las ideas y soluciones aquí presentadas.

La primera y más obvia línea de futuro sería el **despliegue y la evaluación del modelo en un dispositivo de Android real**. Aunque la aplicación web es una excelente herramienta de demostración, el objetivo final de un sistema de este tipo sería poder analizar las aplicaciones directamente en el teléfono del usuario. Esto presenta importantes desafíos de optimización, puesto que, como se concluyó en el análisis de cuantización, el modelo actual es demasiado grande (cerca de 1 GB). Una línea de investigación futura muy interesante sería explorar técnicas avanzadas para reducir su tamaño, como el *pruning* (poda de conexiones neuronales), el entrenamiento consciente de la cuantización (*Quantization-Aware Training*) o incluso el rediseño del modelo utilizando arquitecturas específicamente pensadas para dispositivos móviles, como las inspiradas en MobileNet, adaptándolas para procesar las características estáticas en lugar de imágenes.

Una segunda vía de expansión natural sería la de **escalar el modelo y el *dataset***. Durante la búsqueda de hiperparámetros, se encontró una configuración de modelo mucho más grande y con mayor capacidad teórica de aprendizaje, pero que no ofrecía una mejora de rendimiento significativa con el *dataset* actual de 20 000 aplicaciones. Una línea de trabajo futura consistiría en ampliar de forma masiva el *dataset*, utilizando el *pipeline* ya creado para procesar muchas más APKs de AndroZoo. Con un volumen de datos mucho mayor, sería posible que el modelo más grande sí fuera capaz

de aprender patrones más sutiles y complejos, superando el rendimiento del modelo actual y de los clasificadores clásicos.

En tercer lugar, se podría **enriquecer el conjunto de características extraídas**. Aunque el análisis estático ha demostrado ser muy potente, la incorporación de otros tipos de información podría mejorar aún más la precisión. Se podría explorar la inclusión de características basadas en grafos, como los grafos de llamadas a la API o los grafos de flujo de control, y utilizar modelos de Redes Neuronales de Grafos (GNN) para analizarlos, siguiendo la línea de trabajos como el de MAPAS [42]. Otra opción aún más ambiciosa sería evolucionar hacia un «modelo híbrido», incorporando un número limitado de características dinámicas obtenidas de la ejecución de la aplicación en un *sandbox*, para complementar la visión estática con información acerca del comportamiento real de las aplicaciones.

Finalmente, la **aplicación web de demostración podría ser mejorada y expandida**. Se podrían añadir funcionalidades como un historial de análisis persistente para cada usuario, al igual que un sistema de usuarios y de gestión que permitiera añadir y modificar los modelos que estarían disponibles en la aplicación. Otro posible enfoque sería reestructurar la aplicación y usar un *framework* web más completo como puede ser Django. Esto convertiría la herramienta de una simple demostración a una plataforma de análisis educativo y de diagnóstico mucho más completa.

Bibliografía

[1] API Reference; Matplotlib 3.10.3 documentation — matplotlib.org. https://matplotlib.org/stable/api/index.html.

[2] API reference; pandas 2.3.0 documentation — pandas.pydata.org. https://pandas.pydata.org/docs/reference/index.html#api.

[3] Be careful when interpreting predictive models in search of causal insights; SHAP latest documentation — shap.readthedocs.io. https://shap.readthedocs.io/en/latest/example_notebooks/overviews/Be%20careful%20when%20interpreting%20predictive%20models%20in%20search%20of%20causal%20insights.html#When-predictive-models-cannot-answer-causal-questions-but-causal-inference-methods-can%C2%A0help.

[4] Conda Documentation; conda-docs documentation — docs.conda.io. https://docs.conda.io/en/latest/.

[5] Documentation for Visual Studio Code — code.visualstudio.com. https://code.visualstudio.com/docs.

[6] Git — git-scm.com. https://git-scm.com/.

[7] GitHub - androguard/androguard: Reverse engineering and pentesting for Android applications — github.com. https://github.com/androguard/androguard.

[8] GitHub - MobSF/Mobile-Security-Framework-MobSF: Mobile Security Framework (MobSF) is an automated, all-in-one mobile application (Android/iOS/Windows) pen-testing, malware analysis and security

assessment framework capable of performing static and dynamic analysis. — github.com. https://github.com/MobSF/Mobile-Security-Framework-MobSF.

[9] GitHub - skylot/jadx: Dex to Java decompiler — github.com. https://github.com/skylot/jadx.

[10] GitHub.com Help Documentation — docs.github.com. https://docs.github.com/en.

[11] Home — docs.docker.com. https://docs.docker.com/.

[12] Installing Packages - Python Packaging User Guide — packaging.python.org. https://packaging.python.org/en/latest/tutorials/installing-packages/#creating-virtual-environments.

[13] Introduction | Documentation | Poetry - Python dependency management and packaging made easy — python-poetry.org. https://python-poetry.org/docs/.

[14] Miniconda - Anaconda — anaconda.com. https://www.anaconda.com/docs/getting-started/miniconda/main.

[15] NumPy reference; NumPy v2.3 Manual — numpy.org. https://numpy.org/doc/stable/reference/index.html#reference.

[16] PEP 405 – Python Virtual Environments — peps.python.org. https://peps.python.org/pep-0405/.

[17] Project Jupyter — jupyter.org. https://jupyter.org/. [Accessed 01-07-2025].

[18] PyCharm: The only Python IDE you need — jetbrains.com. https://www.jetbrains.com/pycharm/.

[19] Python 3.11.12 Documentation — docs.python.org. https://docs.python.org/3.11/.

[20] PyTorch documentation; PyTorch 2.7 documentation — docs.pytorch.org. https://docs.pytorch.org/docs/stable/index.html.

[21] Streamlit Docs — docs.streamlit.io. https://docs.streamlit.io/.

[22] TeXstudio - A LaTeX editor — texstudio.org. https://www.texstudio.org/.

[23] User Guide — scikit-learn.org. https://scikit-learn.org/stable/user_guide.html.

[24] venv — Creation of virtual environments — docs.python.org. https://docs.python.org/3.11/library/venv.html.

[25] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[26] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA, 2016. ACM.

[27] Shunichi Amari. A theory of adaptive pattern classifiers. *IEEE Transactions on Electronic Computers*, (3):299–307, 2006.

[28] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26, 2014.

[29] Khaled Bakour, H. Murat Ünver, and Razan Ghanem. The android malware static analysis: Techniques, limitations, and open challenges. In *2018 3rd International Conference on Computer Science and Engineering (UBMK)*, pages 586–593, 2018.

[30] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

[31] Leo Breiman. Bagging predictors. *Machine learning*, 24:123–140, 1996.

[32] Leo Breiman. Random forests. *Machine learning*, 45:5–32, 2001.

[33] Leo Breiman, Jerome Friedman, Richard A Olshen, and Charles J Stone. *Classification and regression trees*. Routledge, 2017.

[34] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[35] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20:273–297, 1995.

[36] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.

[37] Kristopher De Asis, J Hernandez-Garcia, G Holland, and Richard Sutton. Multi-step reinforcement learning: A unifying algorithm. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[38] Jerome H Friedman. Stochastic gradient boosting. *Computational statistics & data analysis*, 38(4):367–378, 2002.

[39] Shivangi Gheewala, Shuxiang Xu, and Soonja Yeom. In-depth survey: deep learning in recommender systems—exploring prediction and ranking models, datasets, feature analysis, and emerging trends. *Neural Computing and Applications*, pages 1–73, 2025.

[40] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of field robotics*, 37(3):362–386, 2020.

[41] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[42] Jinsung Kim, Younghoon Ban, Eunbyeol Ko, Haehyun Cho, and Jeong Hyun Yi. Mapas: a practical deep learning-based android malware detection system. *International Journal of Information Security*, 21(4):725–738, 2022.

[43] Julián Luengo, Diego García-Gil, Sergio Ramírez-Gallego, Salvador García, and Francisco Herrera. Big data preprocessing. *Cham: Springer*, 1:1–186, 2020.

[44] Scott M Lundberg and Su-In Lee. A unified approach to interpreting model predictions. *Advances in neural information processing systems*, 30, 2017.

[45] Samaneh Mahdavifar, Andi Fitriah Abdul Kadir, Rasool Fatemi, Dima Alhadidi, and Ali A Ghorbani. Dynamic android malware category classification using semi-supervised deep learning. In *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*, pages 515–522. IEEE, 2020.

[46] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5:115–133, 1943.

[47] Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*, 2018.

[48] Marvin Minsky and Seymour A Papert. *Perceptrons, reissue of the 1988 expanded edition with a new foreword by Léon Bottou: an introduction to computational geometry*. MIT press, 2017.

[49] Riya Nair, Kiranbhai R Dodiya, and Parth Lakhalani. A Static Approach for Malware Analysis: A Guide to Analysis Tools and Techniques. *International Journal for Research in Applied Science and Engineering Technology*, 11(12):1451–1474, dec 31 2023.

[50] Muhammad Imran Razzak, Saeeda Naz, and Ahmad Zaib. Deep learning for medical image processing: Overview, challenges and the future. *Classification in BioApps: Automation of decision making*, pages 323–350, 2017.

[51] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[52] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[53] Janaka Senanayake, Harsha Kalutarage, and Mhd Omar Al-Kadri. Android mobile malware detection using machine learning: A systematic review. *Electronics*, 10(13):1606, 2021.

[54] Statcounter. Mobile Operating System Market Share Worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide, Julio 2025.

[55] Keras Team. Keras documentation: Keras 3 API documentation — keras.io. https://keras.io/api/.

[56] TFGM_GII UBU, cgosorio, Carlos López Nozal, David Miguel Lozano, and Álvar Arnaiz-González. ubutfgm/plantillaLatex. https://github.com/ubutfgm/plantillaLatex, dec 20 2024.

[57] Jesper E Van Engelen and Holger H Hoos. A survey on semi-supervised learning. *Machine learning*, 109(2):373–440, 2020.

[58] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[59] Shuhei Watanabe. Tree-structured parzen estimator: Understanding its algorithm components and their roles for better empirical performance. *arXiv preprint arXiv:2304.11127*, 2023.

[60] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep ground truth analysis of current android malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings 14*, pages 252–276. Springer, 2017.

[61] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *2012 Seventh Asia joint conference on information security*, pages 62–69. IEEE, 2012.

[62] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[63] Chandra Shekhar Yadav and Sangeeta Gupta. A review on malware analysis for iot and android system. *SN Computer Science*, 4(2):118, 2022.

[64] Mülhem İbrahim, Bayan Issa, and Muhammed Basheer Jasser. A method for automatic android malware detection based on static analysis and deep learning. *IEEE Access*, 10:117334–117352, 2022.