ESCUELA POLITÉCNICA
SUPERIOR DE CÓRDOBA
Universidad de Córdoba

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Estudio comparativo de métodos de aprendizaje automático en la detección de *malware* / *Comparative Study of Machine Learning Methods for Malware Detection*

Autor: Manuel Jesús Mariscal Romero
Directores: D. David Guijo Rubio
D. Víctor Manuel Vargas Yun

septiembre, 2025

UNIVERSIDAD
DE
CÓRDOBA

Resumen

Este proyecto se centra en la aplicación de técnicas de aprendizaje automático para la detección de *malware*. El objetivo principal es comparar distintos algoritmos, tanto los estudiados en el plan de estudios de Ingeniería Informática como otros que no forman parte de la formación académica, con el fin de determinar cuáles se adaptan mejor a este problema.

Para ello, se utilizan bases de datos públicas de *malware*, adaptadas mediante técnicas de preprocesamiento, balanceo y reducción de la dimensionalidad. Posteriormente, se implementa un protocolo experimental que incluye la optimización de hiperparámetros, la validación cruzada estratificada y la repetición de experimentos con diferentes semillas para garantizar la reproducibilidad.

Finalmente, se analizan los resultados obtenidos en términos de precisión, eficiencia y viabilidad computacional, destacando las ventajas e inconvenientes de cada enfoque y aportando una visión comparativa que pueda servir de referencia para futuros estudios en la detección automática de *malware*.

Palabras clave: Aprendizaje automático, Detección de *malware*, Clasificación, Ciberseguridad

Abstract

This project focusses on the application of machine learning techniques for malware detection. The main objective is to compare different algorithms, both those studied in the Computer Engineering curriculum and others not included in the formal academic training, to determine which are better suited to this problem.

For this purpose, public malware datasets are used, adapted through preprocessing, data balancing and dimensionality reduction techniques. Afterwards, an experimental protocol including hyperparameter optimization, stratified cross-validation and the repetition of experiments with different random seeds, is implemented to guarantee reproducibility.

Finally, the results are analyzed in terms of accuracy, efficiency and computational viability, with special focus on the advantages and disadvantages of each approach and providing a comparative perspective to serve as a reference for future research on automated malware detection.

Keywords: Machine Learnig, Malware detection, Clasification, Cibersecurity.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

El surgimiento de nuevas herramientas y tecnologías ha hecho posible mejorar las técnicas de ciberseguridad, tanto las técnicas para proteger la información, como las que explotan las brechas de seguridad. Este mismo desarrollo tecnológico hace que se incrementen de forma exponencial los ataques de *malware*, es decir, cualquier tipo de *software* diseñado para dañar, interrumpir, robar o acceder sin autorización a sistemas informáticos. La mejora de las técnicas de ciberseguridad ha provocado que los ciberdelincuentes se esfuercen aún más por conseguir su objetivo.

En 1971, Creeper [1], el primer *malware* de la historia, fue desarrollado por Bob Thomas Morris como un experimento y no causaba daño en los sistemas. Creeper era un gusano que se autorreplicaba y propagaba a través de ARPANET, y mostraba el mensaje «*I'm the creeper, catch me if you can!*». El primer *malware* con impacto mundial, afectando a un 10 % de los 60000 servidores que había en ARPANET [2], fue el gusano Morris [3], desarrollado en 1988 por Robert Tappan Morris. Morris intentaba obtener la contraseña de los equipos en los que se ejecutaba mediante fuerza bruta, es decir, permutaba los nombres de usuarios conocidos y una lista de las contraseñas más comunes. Creeper y el gusano Morris provocaron la aparición de Reaper, el primer antivirus de la historia y la creación del Equipo de Respuesta ante Emergencias Informáticas (CERT, del inglés *Computer Emergency Response Team*) [4].

Uno de los casos recientes más sonados fue WannaCry [5] en 2018, *ransomware*

[6] que bloquea el acceso a partes del sistema y pide un rescate. Este *malware* causó un gran impacto a nivel mundial, afectando en España a empresas como Telefónica o Iberdrola [7].

Este aumento en la complejidad de las técnicas usadas tanto para dañar los sistemas como para evitar su detección, ha hecho que los métodos tradicionales basados en firmas [8] para detectar patrones únicos en el código queden obsoletos. A día de hoy se combina con la heurística y sigue siendo una de las técnicas más usadas, pero son insuficientes para enfrentar vectores desconocidos.

En los últimos años, uno de los campos más estudiados en informática y con mayor avance y previsión de futuro es el aprendizaje automático [9]. Este es un campo de estudio dentro de la inteligencia artificial que se centra en aprender patrones a partir de datos, en lugar de seguir reglas programadas explícitamente. El sistema entrena con ejemplos y luego generaliza para hacer predicciones o tomar decisiones. El aprendizaje automático podría ser una solución eficiente y escalable para la detección y clasificación de *malware*.

A lo largo de este proyecto se tratará de evaluar la efectividad de diferentes modelos de aprendizaje automático para identificar este tipo de programas. Para ello tendremos en cuenta precisión, velocidad y capacidad de adaptación ante nuevas variantes de amenazas tratando de identificar las ventajas y limitaciones de cada método.

Según Purplesec, con datos recogidos hasta 2018, el *malware* sigue siendo uno de los ataques más usados y estima que se crean aproximadamente 230000 muestras nuevas cada día. Además cree que cada ataque con *malware* cuesta de media unos 2,5 millones de dólares a las compañías. En la Figura 1.1 podemos ver como ha evolucionado el número de infecciones por año [10].

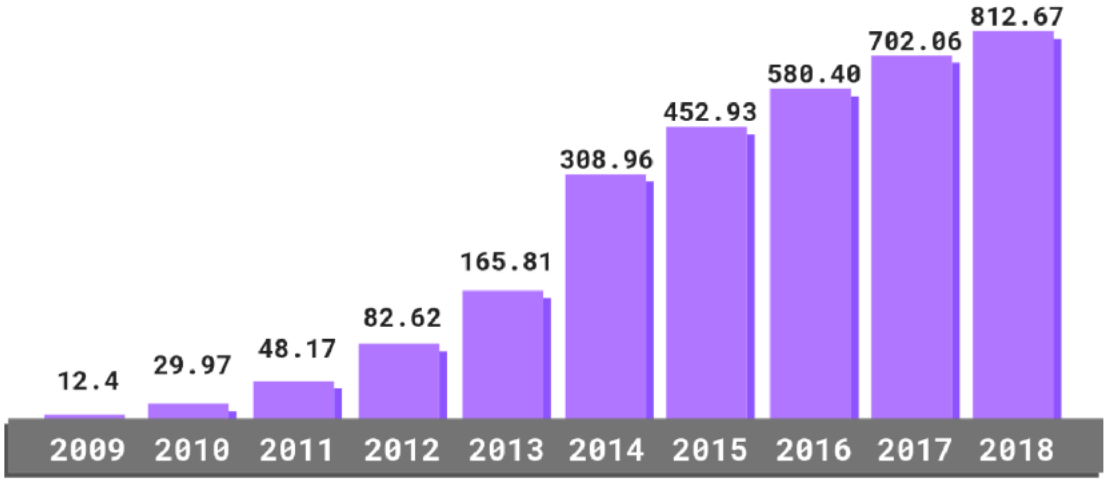


Figura 1.1: Millones de infecciones por año

Capítulo 2

Estado de la técnica

En este capítulo se presentan los fundamentos teóricos más relevantes que sirven como base para el desarrollo de este proyecto. Se revisan conceptos clave relacionados con el aprendizaje automático y su aplicación en la detección de *malware*, así como las nociones generales de ciberseguridad y los enfoques más utilizados en la identificación de *software* malicioso.

2.1. Aprendizaje automático

El aprendizaje automático se puede entender como «la creación de algoritmos y modelos que permiten a los ordenadores aprender y hacer predicciones sin ser específicamente programados» [11].

Inicialmente, los cálculos estadísticos se resolvían con máquinas electromecánicas, como la máquina tabuladora, desarrollada en 1890 por Herman Hollerith [12]. Años más tarde, se presentó la neurona de McCulloch-Pitts, el primer modelo matemático de una neurona biológica, considerado por muchos como el punto de partida para el aprendizaje automático y la base de importantes modelos de cálculo [13]. Hoy en día, el aprendizaje automático es esencial en diferentes ámbitos, como la investigación o los negocios, y emplea algoritmos avanzados capaces de hacer predicciones muy precisas sobre datos desconocidos.

Los algoritmos desarrollados se pueden dividir en aprendizaje supervisado, no

supervisado, semisupervisado y por refuerzo. Si nos centramos en el aprendizaje supervisado, se pueden dividir en técnicas de clasificación y regresión, siendo las primeros las que nos ocupan en este proyecto. Algunas de las principales técnicas de clasificación son: árboles de decisión, redes neuronales artificiales y máquinas de vectores de soporte (del inglés *support-vector machines*, SVM).

Todas estas técnicas se pueden adaptar a las necesidades actuales de la ciberseguridad. Los tipos de ataques, *malware* o vulnerabilidades de los sistemas se están haciendo cada vez más frecuentes, no solo aumentando en cantidad, sino también en complejidad. Estos algoritmos pueden predecir si un *software* es malicioso, si tiene vulnerabilidades o si un correo electrónico puede considerarse un intento de *phishing*.

A continuación se comentarán algunas de las principales técnicas de preprocesamiento, como el balanceo de datos y la reducción de la dimensionalidad, además de estudiar algunas de las métricas más comunes y modelos que podrían llegar a aplicarse en este estudio si fuera necesario.

2.1.1. Balanceo de datos

Es muy habitual que los conjuntos de datos se encuentren desbalanceados. Esto significa que la cantidad de patrones pertenecientes a una o varias clases son significativamente menores a la clase mayoritaria. Por ejemplo, como veremos más adelante en la tabla 5.1, la clase de patrones no maliciosos representa aproximadamente la mitad del total de patrones, en torno a 150000 patrones, mientras que la clase *exploit* solo tiene 12 patrones [14]. Debido a este desbalanceo es probable que la capacidad de generalización del modelo se vea afectada.

Para mitigar estos problemas, las dos principales técnicas son el submuestreo y el sobremuestreo, *undersampling* y *oversampling* por sus términos en inglés.

2.1.1.1. Submuestreo

El *undersampling* engloba las técnicas que tratan de igualar las distribuciones de datos desbalanceados eliminando muestras de las clases mayoritarias respetando la distribución de la clase minoritaria. Las soluciones a este problema pueden cambiar en función del algoritmo que decide los patrones a eliminar [15]. Las principales técnicas de *undersampling* son:

1. ***Random undersampling.***
   Es el método más sencillo, ya que solo se encarga de eliminar patrones de forma aleatoria de las clases mayoritarias. Lo habitual, y la técnica empleada en este estudio para la clasificación binaria, es igualar el número de patrones para cada clase. Su principal punto en contra es que puede eliminar datos útiles [16].

2. ***Condensed nearest neighbours.***
   Es un algoritmo no paramétrico basado en instancias, donde la clasificación se determina a partir de los k casos más cercanos al punto. Es una técnica local que utiliza medidas de distancia para identificar la similitud entre observaciones [17].

3. ***Tomek links.***
   Consiste en identificar pares de instancias pertenecientes a clases distintas que son mutuamente los vecinos más cercanos entre sí. Suelen encontrarse en las zonas de solapamiento entre clases y representan casos ruidosos. Se elimina del conjunto de datos la instancia perteneciente a la clase mayoritaria en cada par, reduciendo así el desbalanceo [18].

4. ***Edited nearest neighbours.***
   Este método se basa en revisar cada instancia del conjunto de datos y clasificarla según la regla de los k vecinos más cercanos. Si la instancia no coincide con la clase mayoritaria de sus vecinos, se considera ruidosa o mal ubicada y se elimina del conjunto [19].

2.1.1.2. Sobremuestreo

Las técnicas de sobremuestreo, *oversampling* en inglés, buscan aumentar la representación de la clase minoritaria o de las clases minoritarias, si son más de una, generando nuevas instancias a partir de los casos existentes. El problema que presenta esta técnica es el riesgo de introducir información no real [15]. Entre los métodos de *oversampling* más destacados se encuentran:

1. ***Random oversampling.***
   Es la técnica más sencilla, ya que copia los patrones de la clase minoritaria o de las clases minoritarias, si hay más de una hasta alcanzar la cantidad deseada. Es habitual que se haga hasta igualar a la clase mayoritaria [15].

2. ***Synthetic Minority Oversampling Technique (SMOTE).***
   Genera instancias sintéticas de las clases minoritarias en lugar de replicar ejemplos existentes. Para ello, selecciona un ejemplo de la clase minoritaria y crea nuevos puntos interpolando con sus vecinos más cercanos [20].

3. ***Adaptative Synthetic Sampling (ADASYN).***
   Es una extensión de SMOTE que genera más instancias sintéticas en las zonas donde la clase minoritaria o de las clases minoritarias, si son más de una, es más difícil de aprender, es decir, donde está menos representada respecto a la mayoritaria [21].

2.1.2. Reducción de la dimensionalidad

En estadística, la reducción de la dimensionalidad es el proceso por el cual se reduce el número de variables aleatorias. Si aplicamos esto al aprendizaje automático, el objetivo a reducir es el número de características del conjunto de datos. Generalmente se aplica antes de la clasificación para evitar los efectos de la maldición de la dimensionalidad. Esta maldición implica que cuando aumenta la dimensionalidad, el volumen del espacio aumenta exponencialmente haciendo que los datos se vuelvan dispersos [22].

La principal ventaja que aportan estas técnicas es reducir el tiempo de entrenamiento y la memoria utilizada en el mismo [23]. A continuación, estudiaremos algunos de los principales métodos.

1. **Análisis de componentes principales**
   El análisis de componentes principales se usa para describir un conjunto de datos en términos de nuevas características no correladas, buscando la proyección donde los datos queden mejor representados según el método de mínimos cuadrados [24].

2. **Análisis factorial**
   Tiene el objetivo es identificar un conjunto reducido de factores latentes que explican la mayor parte de la varianza observada en los datos para descubrir las estructuras que generan las correlaciones entre las variables [25].

3. **Descomposición en valores singulares**
   Es una técnica algebraica que descompone una matriz en tres componentes: U, Σ y V^T [26]. Esta descomposición permite representar los datos en un espacio reducido preservando la mayor parte de la información relevante.

2.1.3. Métricas de evaluación

La elección de métricas de evaluación adecuadas es esencial para valorar de forma precisa el rendimiento de los modelos. No todas las métricas ofrecen la misma información. En esta sección se revisan las métricas más empleadas en la literatura especializada, destacando su utilidad, limitaciones y el tipo de información que aportan para la comparación de modelos.

2.1.3.1. Matriz de confusión

Una matriz de confusión permite visualizar el rendimiento de un algoritmo de clasificación, normalmente supervisado. Cada fila representa las instancias en una clase real, mientras que cada columna representa las instancias en una clase predicha (o viceversa). La diagonal de la matriz representa las instancias correctamente clasificadas [27]. Las matrices de confusión pueden utilizarse con cualquier algoritmo clasificador. En clasificación, las medidas suelen obtenerse de la matriz de confusión. En las Figuras 2.1 y 2.2 podemos ver un ejemplo para clasificación binaria y multiclase respectivamente.

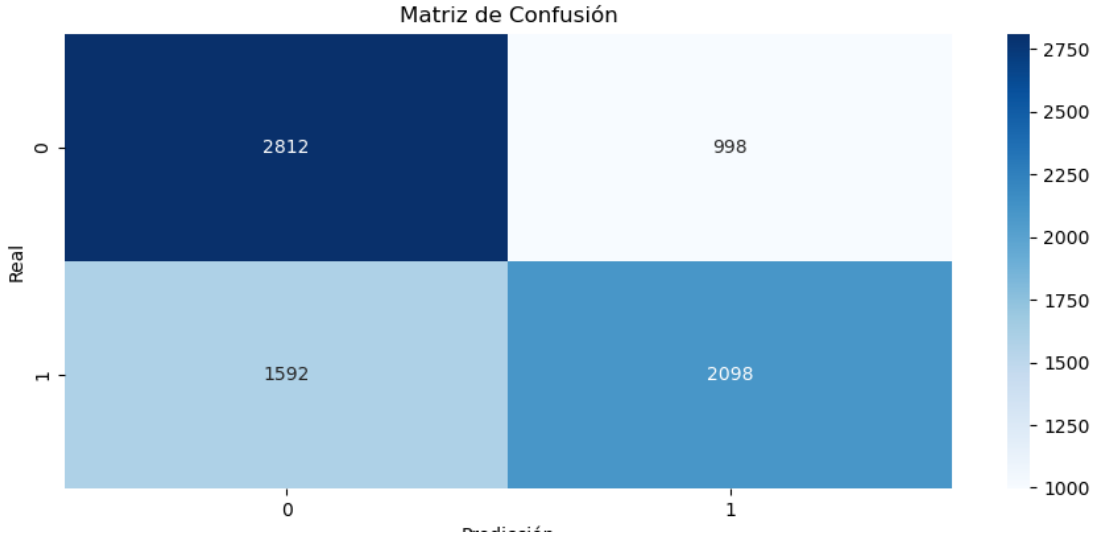


Figura 2.1: Ejemplo de matriz de confusión para clasificación binaria

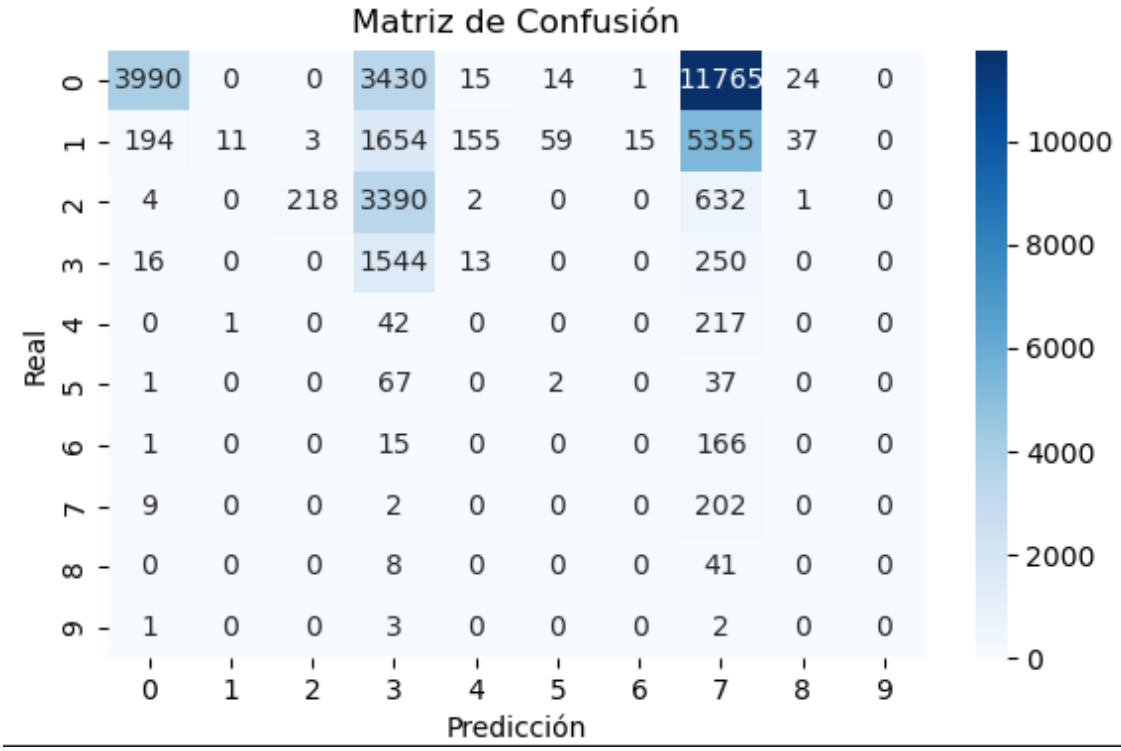


Figura 2.2: Ejemplo de matriz de confusión para clasificación multiclase

2.1.3.2. Exactitud

La exactitud o *Accuracy* (CCR) se corresponde con el porcentaje de aciertos que se han producido, es decir, los patrones clasificados correctamente respecto al total. Se calcula como la suma de verdaderos positivos (TP) y verdaderos negativos (TN) respecto al número total de patrones de entrada (N) [28].

$$\text{CCR} = \frac{TP + TN}{N} \tag{2.1}$$

2.1.3.3. Precisión

La precisión es una métrica que evalúa la proporción de patrones clasificados como positivas que realmente pertenecen a la clase positiva, es decir, mide como de confiable es el modelo cuando predice un positivo. Es muy relevante cuando el coste de clasificar erróneamente un negativo como positivo es alto.

$$\text{Precisión} = \frac{TP}{TP + FP} \tag{2.2}$$

Donde TP representa el número de verdaderos positivos, y FP corresponde al número de falsos positivos.

2.1.3.4. Sensibilidad

También conocida como exhaustividad o *recall* en inglés, mide la capacidad del modelo para detectar correctamente los positivos de un conjunto de datos. Como se muestra en la ecuación 2.3, se calcula como la proporción entre el número de verdaderos positivos (TP) y la suma de verdaderos positivos y falsos negativos (FN) [28]. Un valor alto de sensibilidad indica que se han obtenido pocos falsos negativos.

$$\text{Sensibilidad} = \frac{TP}{TP + FN} \tag{2.3}$$

2.1.3.5. Mínima sensibilidad

La mínima sensibilidad mide cómo de bien se clasifica la clase peor clasificada. Es útil en clasificación multiclase o con conjuntos de datos desbalanceados, ya que permite identificar si existe alguna clase que el modelo no está clasificando correctamente. Un valor alto indica que el modelo mantiene un buen rendimiento en todas las clases, mientras que un valor bajo revela que, al menos, una de ellas presenta un bajo grado de acierto. Si el modelo se deja una clase sin clasificar, el valor será 0.

Sea S_i la sensibilidad de la clase i, con n el número total de clases, la mínima sensibilidad se calcula como se muestra en la ecuación 2.4.

$$MS = \min_{i \in \{1,2,\ldots,n\}} S_i \tag{2.4}$$

Donde la sensibilidad de cada clase S_i se obtiene mediante la ecuación 2.3

2.1.3.6. Valor-F1

El valor-F1 o *F1-score* mide el equilibrio entre la precisión y la sensibilidad [28]. Se calcula como la media armónica entre ambas, lo que penaliza de forma más severa los valores extremos y proporciona una medida equilibrada del rendimiento del modelo. Es especialmente útil en problemas con clases desbalanceadas, ya que evita que un alto rendimiento en una sola métrica distorsione la evaluación global. Su ecuación se describe como:

$$F_1 = \frac{2 \cdot TP}{2 \cdot TP + FP + FN} \tag{2.5}$$

2.1.4. Validación cruzada

La validación es esencial en la experimentación con modelos de aprendizaje automático, ya que permite evaluar de manera fiable la capacidad de generalización del modelo, reduciendo el sesgo que podría aparecer al hacer una única partición entrenamiento/prueba.

Se usa para para evaluar los resultados de un análisis estadístico y garantizar que son independientes de la partición entre datos de entrenamiento y prueba. Para ello hay que calcular la media aritmética obtenida de las medidas de evaluación sobre diferentes particiones. Este método es habitual cuando el objetivo es la predicción y se quiere estimar la precisión de los modelos que se van a utilizar [29]. A continuación, se estudian las principales estrategias de validación cruzada.

1. **Validación cruzada de K iteraciones**
   Los datos en k subconjuntos, usando en cada iteración uno como prueba y los restantes $(k-1)$ como entrenamiento. El proceso se repite k veces y los resultados se promedian para obtener una única estimación del rendimiento. Ofrece mucha precisión, pero es muy costoso computacionalmente [30].

2. **Validación cruzada aleatoria**
   En este caso se eligen aleatoriamente los datos que pertenecen a cada partición de entrenamiento y prueba. El resultado es la media aritmética de los valores obtenidos para las métricas en cada iteración [31].

3. **Validación cruzada dejando uno fuera**
   En este tipo de validación se separan los datos de forma que tenemos una sola muestra para los datos de prueba, mientras que para los de entrenamiento tenemos el resto de patrones del conjunto de datos. Se realizan tantas iteraciones como muestras (N) tenga el conjunto de datos, calculando el error para cada una de ellas como evaluación y se obtiene la media aritmética de todas ellas [32].

$$E = \frac{1}{N}\sum_{i=1}^{N} E_i$$

2.1.5. Algoritmos de clasificación

En esta sección se presentan algunos algoritmos de clasificación más comunes, entre ellos, los que se van a usar en el estudio. Los clasificadores son modelos de aprendizaje automático capaces de asignar una etiqueta a cada instancia de datos en función de sus características. Se describen tanto modelos simples y rápidos, que facilitan la interpretación de los resultados, como modelos más complejos, que ofrecen un mayor potencial de generalización y precisión.

2.1.5.1. Árboles de decisión

Son modelos de aprendizaje supervisado que representan las decisiones mediante una estructura jerárquica de nodos y ramas. Cada nodo interno evalúa un atributo de los datos y realiza una bifurcación según el valor de dicho atributo, mientras que los nodos hoja asignan la clase final. Normalmente usando criterios como la ganancia de información o el índice Gini [33, 34].

Ventajas.

- La estructura del árbol es visual y fácil de entender.
- Una vez construido, clasificar nuevas instancias es muy rápido.
- Puede manejar datos categóricos y continuos sin necesidad de gran preprocesamiento.
- No asume ninguna distribución particular de los datos, adaptándose a distintas formas de patrones.

Inconvenientes.

- Los árboles pueden crecer demasiado y memorizar el conjunto de entrenamiento, perdiendo capacidad de generalización.
- Cambios leves en los datos pueden alterar la estructura del árbol de manera significativa.
- Limitada capacidad predictiva para relaciones complejas.

2.1.5.2. *Random forest*

Random forest es un método de ensamblado que construye múltiples árboles de decisión sobre distintas muestras del conjunto de datos y combinando aleatoriamente subconjuntos de características. Cada árbol individual realiza su predicción, y el bosque final decide por mayoría de votos (en clasificación) o promedio (en regresión). Esta estrategia reduce la varianza y mejora la capacidad de generalización respecto a un único árbol de decisión, manteniendo robustez frente al ruido y al sobreajuste [35, 36, 37].

Ventajas.

- Es uno de los clasificadores más certeros.
- Suele obtener mejores resultados que un árbol individual.
- Maneja variables categóricas y continuas y tolera datos faltantes.
- Permite identificar qué atributos contribuyen más a la predicción.

Inconvenientes.

- Al estar compuesto por muchos árboles, es difícil interpretar.
- Entrenar y almacenar múltiples árboles requiere más recursos que un árbol único.
- Puede sobreajustar en ciertas tareas de clasificación o regresión.

2.1.5.3. *k-Nearest Neighbors*

El método de los k vecinos más cercanos es un método de clasificación supervisada no paramétrico que estima la probabilidad de que un patrón pertenezca a una clase. Es un tipo de aprendizaje vago, donde la función se aproxima localmente y el cálculo se hace en la clasificación [38].

Funciona asignando a un archivo o programa nuevo la categoría de sus vecinos más cercanos en un conjunto de datos previamente etiquetado. Es decir, compara el nuevo caso con las muestras conocidas y lo clasifica según la mayoría de etiquetas entre los más similares. La cercanía se mide normalmente usando la distancia euclidiana [39].

2.1.5.4. *Ridge*

El clasificador *Ridge* se basa en la regresión lineal, pero incorpora un término de penalización que evita que los coeficientes del modelo se vuelvan demasiado grandes. De esta forma, se controla el sobreajuste y se mejora la capacidad de generalización en datos nuevos. En la práctica, *Ridge* busca un equilibrio entre ajustar bien los datos de entrenamiento y mantener la complejidad del modelo bajo control [40].

2.1.5.5. Perceptrón multicapa

Un perceptrón es el modelo matemático más simple para representar una neurona, una célula especializada que posee una cantidad indefinida de canales de entrada y un canal de salida. Los canales de entrada recogen información del entorno y la envían a la neurona, que reacciona y envía una respuesta al cerebro. La labor de las neuronas se vuelve valiosa al asociarse a otras, donde el canal de salida se asocia al canal de entrada de otra.

Esto permite construir una red, donde la información se transmite y se procesa de manera jerárquica entre múltiples capas de neuronas. En esta red llamada perceptrón multicapa, ada neurona realiza operaciones simples, pero al combinarse en capas y mediante funciones de activación no lineales, la red es capaz de modelar relaciones complejas y patrones en los datos [41].

2.1.5.6. Máquinas de vectores de soporte

Las Máquinas de vectores de soporte permiten optimizar los márgenes de separación entre clases y funciona muy bien en problemas de clasificación complejos. Su fortaleza reside en el uso de funciones *kernel*, que permiten transformar los datos en espacios de mayor dimensión y separar clases que no son linealmente separables. En la detección de *malware*, donde las fronteras entre software benigno y malicioso pueden ser difusas, esto resulta especialmente útil. El principal inconveniente es que su coste computacional puede ser elevado en conjuntos de datos grandes.

2.1.5.7. *Light Gradient-Boosting Machine*

El clasificador *Light Gradient-Boosting Machine*, es muy eficiente en problemas de clasificación con grandes volúmenes de datos y un número elevado de características. Se basa en el método de *gradient boosting*, pero introduce optimizaciones como el uso de histogramas y técnicas de reducción de memoria que lo hacen más rápido y escalable que otros métodos similares. *LightGBM* ha mostrado resultados competitivos estudios recientes [42].

2.2. Ciberseguridad

La ciberseguridad es la protección de la infraestructura informática y la información que hay en ella, abarcando *software*, *hardware* y redes. Para garantizar la seguridad, es esencial combinar estrategias de prevención con métodos de protección efectivos. Las estrategias de prevención, como el uso de *firewalls*, *software* antivirus actualizado y educación en ciberseguridad para los usuarios, se centra en identificar y mitigar posibles amenazas antes de que ocurran.

Por otro lado, la protección se enfoca en responder a los incidentes y minimizar sus efectos, mediante herramientas como los sistemas de detección de intrusiones. Con esto, podemos llegar a la conclusión de que el objetivo de la seguridad es minimizar los riesgos de recibir un ataque y reducir el impacto en caso recibirlo [43]. En esta sección nos centraremos en la ciberseguridad *software*, concretamente en los aspectos relacionados con la detección y clasificación de *malware*.

2.2.1. Conceptos generales

La ciberseguridad constituye un pilar esencial en la sociedad digital actual, donde cada vez más actividades cotidianas, económicas y sociales dependen de sistemas informáticos y redes de comunicación. Su importancia radica en garantizar que la información y los servicios digitales sean fiables y estén protegidos frente a accesos indebidos o manipulaciones maliciosas.

Los principales objetivos de la ciberseguridad se articulan en torno a la denominada triada CIA (*Confidenciality, Integrity and Availability*, por sus siglas en inglés.) [44]:

- **Confidencialidad**: asegura que los datos solo sean accesibles por usuarios autorizados.
- **Integridad**: garantiza que la información no sea alterada de forma no autorizada.
- **Disponibilidad**: busca que los sistemas y servicios estén siempre accesibles.

En este contexto, las organizaciones y usuarios se enfrentan a amenazas comunes como que se comentarán superficialmente por su falta de relación con el proyecto. Algunas de las amenazas más frecuentes son:

1. ***Phishing***: Correos electrónicos, mensajes o enlaces fraudulentos que buscan engañar al usuario para que revele información confidencial [45].

2. ***Malware***: Software malicioso que diseñado para dañar sistemas, robar información o tomar control de dispositivos. Se explicará en profundidad en la sección 2.2.2.

3. **Denegación de servicio**: Inundación de sistemas con tráfico para interrumpir su funcionamiento y dejar servicios inaccesibles [46].

4. **Robo de credenciales/fuerza bruta**: Intentos de acceder a cuentas mediante contraseñas robadas, adivinación sistemática o explotación de vulnerabilidades de autenticación.

De todas estas opciones, el objeto de estudio de este proyecto será el *malware*.

2.2.2. *Malware*

El *software* malicioso o *malware* es cualquier tipo de *software* que se introduce sin el consentimiento del usuario con el objetivo de dañar o comprometer la confidencialidad, integridad o disponibilidad de la información o el sistema [47]. Su objetivo puede ir desde la interrupción del funcionamiento de sistemas hasta el robo de información sensible o la obtención de control remoto sobre dispositivos.

Los primeros ejemplos de *malware* surgieron en las décadas de 1970 y 1980 como experimentos de laboratorio o programas con fines demostrativos. Actualmente se ha convertido en una de las amenazas externas más relevantes debido al daño que puede llegar a causar y afectando tanto a usuarios individuales como a grandes corporaciones y entidades gubernamentales. Su relevancia se ha incrementado con la expansión de la digitalización y el uso masivo de Internet, donde ataques automatizados y campañas de *malware* buscan constantemente vulnerabilidades en sistemas y redes, además de posibles descuidos de los usuarios.

2.2.2.1. Tipos de *Malware*

Podemos clasificar el *malware* en diferentes categorías [48] según su propósito:

- **Virus**. Tienen como objetivo infectar archivos y sistemas informáticos. Se propagan cuando los usuarios comparten archivos o ejecutan programas infectados.
- **Gusanos**. Se propagan a través de las redes sin que tenga que intervenir el usuario.
- **Troyanos**. Se presentan como un *software* legítimo. De esta forma intentan engañar al usuario para que lo descargue, instale y ejecute.
- ***Adware***. Muestra anuncios de forma intrusiva. Puede ser incrustada en una página web mediante gráficos, carteles, ventanas flotantes, o durante la instalación de algún programa al usuario, con el fin de generar lucro a sus autores.
- ***Spyware***. Trata de conseguir información de un equipo sin conocimiento ni consentimiento del usuario. Después transmite esta información a una entidad externa.
- ***Ransomware***. Conocido como secuestro de datos en español. Está diseñado para restringir el acceso a archivos o partes de un sistema y pedir un rescate para quitar la restricción.
- ***Rootkit***. Es un conjunto de *software* que permite al atacante un acceso de privilegio a un ordenador, manteniendo presencia inicialmente oculta al control de los administradores.
- ***Keylogger***. Se encarga de registrar las pulsaciones que se realizan en el teclado, para memorizarlas en un fichero o enviarlas a través de Internet.
- ***Exploit***. Aprovecha un error o una vulnerabilidad de una aplicación o sistema para provocar un comportamiento involuntario.
- ***Backdoor***. Puerta trasera en español. Este tipo de *software* permite un acceso no autorizado al sistema, evitando pasar por los métodos de autenticación.

2.2.3. Técnicas de detección de *malware*

Ningún método de detección es infalible y los principales antivirus comerciales pueden combinar distintas técnicas en función de las necesidades. La detección basada en firmas siguen siendo el método más usado en términos absolutos porque son rápidas, eficientes y fáciles de implementar. Este método consiste en comparar archivos con una base de datos de patrones conocidos. Otros mecanismos son: la detección heurística, por comportamiento, *sandbox* e inteligencia artificial [49].

Existen varias limitaciones de los métodos tradicionales frente a nuevas amenazas. Por ejemplo, para evadir la detección basada en firmas se generaba una cadena de bits única cada vez que se codificaba. Esto se denomina polimorfismo. Gracias a la heurística no era necesaria una coincidencia exacta con las firmas almacenadas, pero debido a la gran cantidad de variaciones que surgen a diario, su efectividad y la de otros mecanismos se ve comprometida [50]. A continuación se estudiarán algunas de las técnicas más usadas.

2.2.3.1. Detección basada en firmas

La detección de intrusiones basada en firmas compara la actividad del sistema con patrones conocidos de ataques para identificar comportamientos maliciosos. El código del *Malware* se compila como cualquier programa, y este código se puede cifrar para crear una firma única. Otras formas de crear una firma son sus acciones en memoria o los archivos específicos que generan en ubicaciones específicas. En los casos más modernos se comunican a través de Internet hacia direcciones IP o dominios, lo que permite a los sistemas de defensa y prevención detectarlas y alertar [51].

Las firmas habitualmente son generadas y almacenadas por los proveedores comerciales de estos servicios, como pueden ser *ESTET* [52] o *Bitdefender* [53], aunque algunas pueden ser abiertas o colaborativas, como es el caso de *VirusTotal* [54].

Ventajas. Las detección basada en firmas es un sistema muy preciso, ya que si el autor del *Malware* no realiza cambios en el código, la firma no cambia y la detección es casi inmediata. Esto implica que el sistema de detección proporcione muy pocos falsos positivos [51].

Inconvenientes. El principal inconveniente de la detección basada en firmas es que para poder actuar, la amenaza debe ser conocida. Esto significa que no tienen forma de detectar las amenazas de tipo *zero day*, es decir, que acaban de ser descubiertas y que aún no tienen una firma [55].
Como se ha comentado en las ventajas, es casi infalible si no se modifica el código del programa, pero la firma queda completamente inservible con un pequeño cambio que se introduzca [8].

2.2.3.2. Detección heurística

En la detección de *Malware*, se conoce como heurística al conjunto de técnicas que se usan para identificar aplicaciones maliciosas que no se encuentran en la base de datos de firmas [56]. Surge de la necesidad de combatir las nuevas amenazas y es uno de los pocos métodos capaces de combatir los *Malware* polimórficos, que cambian su código constantemente [57]

Ventajas. La principal ventaja de la detección heurística es su capacidad para identificar nuevas amenazas desconocidas. Al no depender exclusivamente de firmas previas, resulta eficaz para reconocer modificaciones de *Malware* existente o ataques que todavía no han sido documentados. Esto amplía la cobertura de protección frente a amenazas emergentes [58].

Inconvenientes. Este tipo de análisis trata de detectar el comportamiento conocido, por lo que si un programa no realiza ninguna acción conocida, es probable que no sea detectado [58].

2.2.3.3. Detección basada en comportamiento

Esta técnica es clave frente a amenazas que pueden evitar ser detectadas por los métodos tradicionales. Los atacantes, para evitar ser detectados, ofuscan el código, práctica que "consiste en transformar el código fuente de un programa en una forma más compleja y difícil de comprender, sin alterar su funcionalidad" [59].

La detección basada en comportamiento analiza las actividades y patrones de ejecución de programas para identificar posibles amenazas y descubrir comportamientos anómalos que puedan indicar una intrusión, incluso cuando se trata de ataques desconocidos o de tipo *zero day*. Es capaz de identificar amenazas desconocidas, ya que no depende de bases de datos de firmas previas pero suele generar un mayor número de falsos positivos.

2.2.3.4. Detección mediante aprendizaje automático

Los modelos de aprendizaje automático tienen la capacidad de extraer conocimiento a partir de grandes cantidades de información y reconocer tanto patrones complejos como inusuales. Examina entradas como registros del sistema, tráfico de red, procesos en ejecución o secuencias de instrucciones para localizar conductas anómalas. Además, pueden ajustarse de forma continua incorporando nuevas muestras, lo que incrementa su efectividad frente a variantes no identificadas [60].

En este caso, si un atacante realiza cambios en el código, lo ofusca o es un programa no conocido, no tiene por qué engañar al modelo. Esto hace que sea uno de los métodos de detección de *malware* más potentes y con proyección de futuro [61].

Durante la realización de este proyecto vamos a centrarnos en la detección de *malware* mediante métodos de aprendizaje automático, concretamente de clasificación nominal.

Capítulo 3

Formulación del problema y objetivos

En este capítulo se describirá el contexto en de la investigación y los retos que plantea, así como los problemas que surgen de esta situación. Después, definiremos los objetivos específicos que orientan el estudio, estableciendo las metas a alcanzar y el alcance del proyecto. Al mismo tiempo, se abordarán con más detalle las situaciones que han provocado estos problemas y se expondrán distintos objetivos con la intención de mitigarlos.

3.1. Contexto y motivación

La detección de *malware* es uno de los grandes desafíos de la ciberseguridad por su rápida y constante evolución. Cada día aparecen nuevas amenazas capaces de evadir las técnicas tradicionales explicadas en la sección 2.2.3 y no es posible depender de reglas predefinidas. Esto ha puesto de manifiesto la necesidad de evolucionar al mismo ritmo las técnicas de detección y ha hecho que las técnicas tradicionales resulten insuficientes por si solas. Con el rápido crecimiento que está teniendo el aprendizaje automático, podría ser un gran aliado si se usa de la forma adecuada, ya que es una herramienta muy potente capaz de detectar amenazas aún no conocidas.

El uso de algoritmos de aprendizaje automático permite automatizar el análisis

de grandes volúmenes de datos y adaptarse a la evolución de las amenazas. Además, existe una gran variedad de modelos, lo que posibilita evaluar diferentes estrategias e identificar los métodos más precisos, eficientes y escalables. Por otro lado, es posible volver a entrenar usando nuevos datos y adaptarse rápidamente a los cambios que surjan [62].

3.2. Definición del problema

A continuación, se definen los principales problemas y preguntas que nos hemos encontrado:

- Detectar nuevas amenazas sin necesidad de conocerlas previamente.
- Dificultad de seleccionar el algoritmo más adecuado.
- Limitaciones de recursos computacionales.
- ¿Cómo afectan las características de cada conjunto al rendimiento de los modelos?
- ¿Qué variables son más influyentes en la clasificación?

3.3. Objetivos

A partir de los problemas presentados en la sección 3.2, podemos establecer una serie de objetivos que definirán el desarrollo del estudio del que trata este proyecto. Los objetivos se pueden dividir en dos tipos: generales y específicos. El primero es la columna vertebral del proyecto, el tema central sobre el que gira el estudio que se realizará. Los objetivos específicos dividen el objetivo principal en otros más concretos y deseables. A continuación, se expondrán ambos tipos.

3.3.1. Objetivo general

El objetivo principal para este estudio es comparar distintos algoritmos de aprendizaje automático haciendo uso de conjuntos de datos de *malware*. Este objetivo se centra en los dos primeros problemas comentados en la sección 3.2, tratando de conseguir detectar nuevas amenazas y tener una idea general de qué algoritmos se adaptan mejor a este propósito. Para ello, evaluaremos su eficiencia, precisión y viabilidad computacional.

3.3.2. Objetivos específicos

A partir del objetivo principal y del resto de problemas planteados, podemos concretar una serie propósitos más concretos:

- Estudio teórico de distintos algoritmos en la detección de *malware*.
- Obtención y análisis de bases de datos públicas de *malware*.
- Implementación de metodologías de detección de *malware* y su adaptación para uso en las bases de datos anteriores.
- Evaluar el rendimiento, eficiencia, precisión y viabilidad computacional de estos algoritmos.
- Identificar y analizar los métodos que se adaptan mejor al problema, destacando las ventajas e inconvenientes de cada uno de los algoritmos.
- Identificación de las variables e información más influyentes en la detección de *malware*, particularmente para cada base de datos.

Capítulo 4

Metodología de trabajo

Este capítulo describe la metodología seguida para el desarrollo del proyecto. Se explican los enfoques, técnicas y herramientas utilizadas para alcanzar los objetivos. Además, se expone el tipo de estudio realizado y la selección del conjunto de datos y los modelos. El propósito es ofrecer una guía clara del proceso seguido.

4.1. Enfoque metodológico

Existen varios enfoques aplicables a este tipo de proyecto, pero dado el carácter planteado inicialmente en los objetivos, se centrará en un estudio comparativo y experimental de distintos algoritmos de aprendizaje automático en la detección de *malware*. Se combina la experimentación práctica sobre conjuntos de datos reales con análisis estadísticos sobre los resultado obtenidos en las distintas pruebas realizadas. Cada modelo se somete a pruebas controladas en escenarios de clasificación binaria y multiclase, utilizando conjuntos de datos públicos y representativos.

Esta metodología permite identificar los algoritmos con mejor equilibrio y adaptación a nuevos patrones. También se podrán detectar posibles limitaciones y áreas de mejora para futuras investigaciones. Este enfoque proporciona un marco sistemático para el análisis comparativo de modelos, facilitando la interpretación de resultados y la toma de decisiones fundamentadas sobre el rendimiento de cada algoritmo.

4.2. Preparación del entorno

En esta sección se describe el entorno de trabajo utilizado por el alumno para la implementación de los modelos y la realización de las pruebas. El entorno se debe preparar de forma correcta, ya que puede afectar a la ejecución de los algoritmos y a la reproducibilidad de los experimentos. A continuación, se explican elementos del entorno como el lenguaje de programación, las bibliotecas y las características del equipo.

4.2.1. Herramientas y bibliotecas

El desarrollo y la experimentación de este proyecto se han llevado a cabo empleando un conjunto de herramientas y bibliotecas muy utilizadas en la ciencia de datos. *Python* ha sido el lenguaje de programación de este trabajo, ya que ofrece una fácil implementación de modelos, manipulación de datos y visualización de resultados. Su popularidad en se debe a su sintaxis sencilla, escalabilidad y amplia variedad de herramientas y bibliotecas [63].

En este proyecto se ha utilizado la versión 3.12 de *Python*, elegida principalmente por su compatibilidad con las bibliotecas empleadas, en particular, con *GridSearchCV*, que aprovechan la paralelización de procesos para mejorar el rendimiento. El problema encontrado es que los hilos no se cierran correctamente, es un comportamiento típico asociado a lo que en programación concurrente se denomina *thread leakage* o hilos huérfanos. provoca que la memoria *RAM* y la *CPU* sigan siendo consumidas incluso después de que la ejecución haya terminado. En teoría, este problema no afecta al rendimiento de los clasificadores, pero puede afectar al tiempo de ejecución.

4.2.1.1. `Scikit-learn`

`Scikit-learn` es un paquete de código abierto en `Python` que ofrece una gran variedad de métodos de aprendizaje automático rápidos y eficientes, gracias a que usan bibliotecas compiladas en lenguajes como `C++`, `C` o `Fortran`. Tiene detrás una comunidad activa que mantiene la documentación, corrige errores y asegura la calidad. Aunque no incluye todos los algoritmos usados en este proyecto, es una herramienta muy recomendable si necesitamos: transformación de datos, aprendizaje supervisado o evaluación de modelos [64].

4.2.1.2. `DLOrdinal`

La biblioteca `dlordinal` incluye muchas de las metodologías más recientes de clasificación ordinal usando técnicas avanzadas de aprendizaje profundo. El enfoque ordinal de esta herramienta tiene el objetivo de aprovechar la información de orden presente en la variable objetivo usando funciones de pérdida, diversas capas de salida y otras estrategias [65]. El módulo de `dlordinal` que nos ha resultado de utilidad para este proyecto ha sido la el conjunto de métricas que incluye para evaluar los modelos utilizados, ya que cuenta con una de las métricas que finalmente hemos usado: mínima sensibilidad.

4.2.1.3. `Matplotlib`

Para una mejor visualización de los datos obtenidos en los modelos utilizados, se ha usado la biblioteca `Matplotlib`, ya que incluye una gran cantidad de recursos para la representación gráfica de la información [66]. Se ha usado, en combinación con `Seaborn`, descrita en la sección 4.2.1.7.

4.2.1.4. `NumPy`

La biblioteca `NumPy` tiene como objetivo principal dar soporte a la creación de vectores y matrices de grandes dimensiones, junto con una colección de funciones matemáticas con las que operar [67]. Ha sido de gran utilidad en el desarrollo del proyecto, ya que el conjunto de datos con el que se ha trabajado es de un tamaño considerable, aunque no ha sido necesario hacer uso de las funciones que proporciona porque la mayor parte de los cálculos necesarios se hacen de manera interna en los modelos utilizados.

4.2.1.5. `Pandas`

`Pandas` es una herramienta muy potente para el manejo, análisis y manipulación de datos. Incluye una amplia variedad de herramientas para: leer y escribir datos, reestructuración y segmentación, inserción y eliminación de columnas, mezcla y unión de datos y muchas funcionalidades más [68]. Varias de ellas se han utilizado durante el desarrollo y la preparación del conjunto de datos.

4.2.1.6. `LightGBM`

La biblioteca *Light Gradient-Boosting Machine* por su nombre en inglés, es una infraestructura de aprendizaje automático basada en modelos de árboles de decisión [69]. Se puede usar en diferentes tareas, pero la importante para el análisis realizado el la de clasificación. Los principales algoritmos soportados son: *Gradient Boosting Decision Trees (GBDT)*, el cual utiliza `LGBMClassifier`, clasificador usado durante la experimentación, *Dropouts meet Multiple Additive Regression Trees (Dart)* y *Gradient-based One-Side Sampling (Goss)* [70].

4.2.1.7. `Seaborn`

Basada en `Matplotlib`, `Seaborn` proporciona una interfaz de alto nivel para generar gráficos estadísticos [71]. Es posible usar ambas bibliotecas de forma combinada para una mayor capacidad de visualización. Mientras que `Matplotlib` ofrece un control detallado sobre cada elemento de la figura, `Seaborn` simplifica la creación de visualizaciones complejas, incorporando estilos predefinidos y funciones específicas para el análisis de datos.

4.2.2. *Hardware*

El entrenamiento y evaluación de los modelos se ha realizado en el equipo del estudiante con las siguientes características: procesador *Intel Core i7-4712MQ*, tarjeta gráfica *NVIDIA GeForce 920M*, 16 GB de *RAM* DDR3 y almacenamiento compuesto por un *SSD Crucial MX500* de 250 GB y un *HDD* de 1 TB. Este hardware permite la paralelización de los algoritmos en múltiples núcleos del procesador, lo que reduce significativamente los tiempos de entrenamiento, pero se encuentra muy limitado respecto al conjunto utilizado para clasificación multiclase y modelos más costosos como puede ser *SVM*.

4.2.3. Conjunto de datos

En lo que a *malware* se refiere, *BODMAS* [72] es uno de los conjuntos de datos más completos en la actualidad, con la ventaja para este proyecto de ya estar procesado y tener una amplia bibliografía. Otra opción interesante puede ser *VirusShare* [73], ya que cuenta con más de 99 millones de muestras de *malware* actualizadas pero tiene varios inconvenientes para este proyecto. El primero, es que no incluye muestras de *software* no malicioso y el segundo, que necesita un procesamiento previo para extraer las características. Todo esto conlleva un aumento de tiempo considerable para la realización del proyecto. Otra de las opciones estudiadas ha sido *theZoo* [74]. En cuanto a este repositorio hemos podido observar que tiene los mismos inconvenientes que *VirusShare* y no tiene sus ventajas. Por último tenemos *Microsoft Malware Classification* [75]. En este caso tenemos un conjunto de datos muy amplio con casi medio *terabyte* de información, pero además de los inconve-

nientes ya comentados en los anteriores conjuntos, solo incluye *malware* que afecta a equipos *Windows*, lo que limitaría considerablemente el alcance del estudio.

Teniendo en cuenta todo lo comentado hasta ahora sobre los distintos conjuntos de datos considerados, hemos decidido usar *BODMAS*, ya que es el que mejor se adapta a las necesidades del estudio.

4.2.4. Modelos

Existe una gran variedad de modelos de aprendizaje automático implementados en las diferentes bibliotecas de *Python*. Aunque habría sido interesante hacer una comparación con el mayor número posible de ellos, por limitaciones de equipo y tiempo se ha hecho una pequeña selección siguiendo los siguientes criterios:

- Diversidad en los enfoques de aprendizaje: Modelos rápidos como los árboles, otros más robustos, modelos lineales como referencia, métodos basados en distancia, redes neuronales y máquinas de vectores soporte por su capacidad de trabajar la optimización de márgenes.
- Equilibrio entre interpretabilidad y complejidad, usando modelos simples y algunos complejos pero que suelen ofrecer mejores resultados.
- Se han incluido tanto modelos que toleran bien el desbalanceo como otros más sensibles.
- Escalabilidad y coste computacional, usando desde algunos modelos ligeros a otros más costosos. Esto permite evaluar la viabilidad práctica de cada modelo en escenarios reales

En este proyecto se han empleado diversos algoritmos de aprendizaje automático, seleccionados en función de los criterios mencionados en el capitulo 4 y con el objetivo de representar diferentes enfoques. Se han utilizado los siguientes modelos implementados en *scikit-learn* y *LightGBM*:

- *DecisionTreeClassifier*
- *RandomForestClassifier*
- *KNeighborsClassifier*
- *RidgeClassifier*
- *MLPClassifier*
- *SVC*
- *LGBMClassifier* (de *LightGBM*)

Todos los modelos se han ajustado y evaluado utilizando *GridSearchCV*, lo que permite explorar sistemáticamente distintas combinaciones de hiperparámetros y asegurar comparaciones consistentes entre los distintos métodos de clasificación. La descripción teórica de estos modelos se presenta en el capítulo 2.

4.2.5. Criterios de evaluación

Para evaluar la efectividad de los modelos implementados, de las métricas comentadas en la sección 2.1.3, se han utilizado las siguientes métricas:

- ***Accuracy***: proporción de predicciones correctas sobre el total de patrones.
- **Mínima Sensibilidad**: sensibilidad de la clase peor clasificada.
- ***F1-score***: media armónica entre precisión y sensibilidad. En este documento se ha hecho referencia a ella como valor-F.

Estas métricas han sido seleccionadas porque permiten evaluar de forma equilibrada tanto el rendimiento global del modelo como su capacidad para identificar correctamente las clases menos representadas, evitando que los resultados se vean sesgados por un posible desbalanceo en los datos. Esta combinación ofrece una visión complementaria que facilita la comparación objetiva entre diferentes enfoques.

Capítulo 5

Desarrollo y experimentación

En esta fase se lleva a cabo la implementación práctica del estudio, haciendo uso de los modelos de aprendizaje automático implementados principalmente en la biblioteca *Scikit-Learn* de *python*. Para ello se realiza un procesamiento de los datos, necesario para obtener un conjunto reducido y otro apto para la clasificación multiclase. Además, se configuran los entornos necesarios para su entrenamiento y evaluación, se establecen las métricas de rendimiento, los procedimientos de prueba y los escenarios de experimentación que permitirán obtener resultados consistentes y comparables. El objetivo es verificar, mediante pruebas controladas, la efectividad de cada método en la detección de *malware*.

La parte experimental se aborda desde dos perspectivas complementarias. En primer lugar, se evalúa la capacidad de los modelos para la detección de *malware* mediante pruebas de clasificación binaria, determinando si un patrón corresponde a software malicioso o legítimo. En segundo lugar, se analiza la viabilidad de realizar una clasificación multiclase sobre esos mismos patrones, identificando el tipo específico de *malware* al que pertenecen, lo que permite un análisis más detallado y aplicable a entornos de ciberseguridad avanzada.

5.1. Procesamiento del conjunto de datos

Dadas las limitaciones *hardware* y la cantidad de datos, aproximadamente 135000 patrones y 2400 atributos, es necesario hacer un procesamiento previo del conjunto de datos. Para ello hemos tenido en cuenta varios enfoques. Por un lado, *BODMAS* nos permite hacer una distinción entre clasificación binaria y clasificación multiclase, pero para ello es necesario reordenar los datos, ya que se encuentran distribuidos en varios archivos. Por otro lado, es necesario reducir la cantidad de datos. A continuación veremos los distintos enfoques.

Las pruebas que se realizaran en las secciones 5.1.3.1 y 5.1.3.2 tienen el objetivo de comprobar como se comportan los métodos de *undersamplig* y *PCA*. Teniendo esto en cuenta, solo se han realizado unas pruebas simples con los métodos más sencillos para poder elegir una configuración adecuada para los conjuntos de datos. El resto de métodos de clasificación, así como los de validación, se usarán en el capítulo 6.

5.1.1. Etiquetado de los datos

En esta sección se describe el tratamiento necesario de los archivos disponibles para poder etiquetar cada patrón correctamente.

5.1.1.1. Clasificación binaria

El conjunto de datos para la clasificación binaria no necesita ningún tratamiento previo como pasa en el que usaremos para la clasificación multiclase. En el caso de la clasificación binaria, las clases que usaremos ya se encuentran etiquetadas y almacenadas junto con los patrones en el archivo *bodmas.npz*, por lo que el procesamiento se explicará en la sección 5.1.2.

5.1.1.2. Clasificación multiclase

El conjunto de datos seleccionado se divide en varios archivos:

- *bodmas.npz*: incluye la matriz de patrones de entrada en formato de matriz de *python* y la matriz de salidas deseadas.
- *bodmas_metadata.csv*: la información relevante para nuestro problema es la columna *sha* que contiene la función *hash* de todo el conjunto de datos.
- *bodmas_malware_category.csv*: contiene la función *hash* del *malware* y la categoría a la que pertenece.

Dado que las distintas categorías se encuentran en formato texto, es necesario codificarlas para poder trabajar con ellas. La codificación elegida ha sido la representada en la tabla 5.1.

Tabla 5.1: Codificación de las clases *malware*.

Categoría	Codificación	Nº de patrones	Porcentaje
benign	0	77142	53.26 %
trojan	1	29972	20.69 %
worm	2	16697	11.53 %
backdoor	3	7331	5.06 %
downloader	4	1031	0.71 %
informationstealer	5	448	0.31 %
dropper	6	715	0.49 %
ransomware	7	821	0.57 %
rootkit	8	3	0.00 %
cryptominer	9	20	0.01 %
pua	10	29	0.02 %
exploit	11	12	0.01 %
virus	12	192	0.13 %
p2p-worm	13	16	0.01 %
trojan-gamethief	14	6	0.00 %

Para poder trabajar con las categorías de *malware* en nuestros datos, primero combinamos dos conjuntos de información: uno que contiene detalles sobre cada muestra y otro que indica a qué tipo de *malware* pertenece cada una. Para unir esta información, usamos una herramienta que nos permite combinar ambas fuentes de datos en función de una columna común. Así logramos que, para cada muestra, se indique también su categoría si la tiene.

Sin embargo, hay algunas muestras que no tienen asignada ninguna categoría de *malware*. Esto significa que simplemente esas muestras no son maliciosas. Para dejar esto claro en los datos, rellenamos esos espacios vacíos con la etiqueta *benign*. Además, eliminamos cualquier otra información que no fuera relevante para el análisis, y nos quedamos solo con la categoría de cada muestra.

Finalmente, para poder trabajar de manera más cómoda con estas categorías, las transformamos en números. Esto facilita el procesamiento posterior, ya que los modelos de aprendizaje automático trabajan mejor con valores numéricos que con texto.

El código utilizado para esta tarea se encuentra en el Anexo A.1.

5.1.2. Reducción del conjunto de datos

Esto se hace con el objetivo de disminuir el tiempo que los algoritmos van a necesitar para procesar la información sin perjudicar la integridad de los datos, ya que los resultados del estudio podrían verse afectados y llevar a unas conclusiones erróneas. Esta tarea se puede enfrentar desde dos planteamientos distintos: condensar el número de patrones o el número de características. Ambos planteamientos se han estudiado de forma teórica en esta memoria en las secciones 2.1.1 y 2.1.2 respectivamente. Las técnicas elegidas son *undersampling* por simplicidad y *PCA* porque según el estudio *A Low Complexity ML-Based Methods for Malware Classification* [76] se obtienen unos resultados algo más precisos que con otros métodos.

El código utilizado se encuentra en el anexo A.2. A continuación se explicarán los pasos seguidos.

5.1.2.1. Balanceo de datos: submuestreo

Como ya hemos estudiado en la sección 2.1.1.1, el submuestreo o *undersampling* en inglés, es una técnica para abordar el desbalance de clases en un conjunto de datos, especialmente cuando una de las clases tiene muchos más patrones que la otra. Por simplicidad, este tipo de procesamiento se ha aplicado solo al conjunto de datos utilizado para la clasificación binaria. En nuestro caso, el desbalance no es demasiado grande ya que *BODMAS* contiene 57293 muestras *malware* y 77142 muestras benignas.

El método *RandomUnderSampler* [77] de la biblioteca *Imbalanced learn* nos permite varias formas de actuar, siendo la que nos interesa para este estudio la que nos permite elegir manualmente el número de patrones de cada clase. Hemos elegido una cantidad de 15000 patrones en por clase.

5.1.2.2. Reducción de la dimensionalidad: *PCA*

Este método, consiste en reducir el número de variables de las que consta el problema. Para aplicar el método matemático-estadístico de análisis de componentes principales, *PCA* por sus siglas en inglés, usamos la clase *PCA* [78]. Esta clase nos permite entrenar el modelo y transformar el conjunto de datos tanto para el conjunto de entrenamiento como para el de test. Para ello será necesario separar previamente los datos, ya que *BODMAS* no cuenta con esta división.

5.1.3. Elección final del nuevo conjunto de datos

Para poder decidir como será el conjunto de entrenamiento final se han hecho distintos conjuntos de datos sobre los que se probarán algunos algoritmos. Los conjuntos son los siguientes:

- Clasificación binaria con *PCA*.
- Clasificación binaria con *PCA* y *Undersampling* con 15000 patrones por clase.
- Clasificación multiclase con *PCA*.

5.1.3.1. Elección del conjunto final para clasificación binaria

Los resultados obtenidos se reflejan en las tablas 5.2 y 5.3.

Tabla 5.2: Clasificación binaria con *PCA*.

Clasificador	Tiempo (s)	Entrenamiento			Generalización		
		Acc	MS	F1	Acc	MS	F1
Decision tree	*0.885*	1.000	1.000	1.000	0.972	0.971	1.000
Random forest	25.91	1.000	1.000	1.000	0.984	0.976	1.000
K-NN	**0.095**	0.973	0.970	1.000	0.963	0.963	1.000

Tabla 5.3: Clasificación binaria con *PCA* y *undersampling*.

Clasificador	Tiempo (s)	Entrenamiento			Generalización		
		Acc	MS	F1	Acc	MS	F1
Decision tree	*0.184*	1.000	1.000	1.000	0.945	0.936	1.000
Random forest	4.926	1.000	1.000	1.000	0.963	0.957	1.000
K-NN	**0.016**	0.954	0.948	1.000	0.938	0.931	1.000

Hemos decidido usar el conjunto de datos en el que se ha aplicado tanto *PCA* como *undersampling*, ya que, aunque los resultados son similares en ambos conjuntos, el tiempo es considerablemente más bajo y dadas las limitaciones del equipo disponible puede ser beneficioso a la hora de probar algoritmos más complejos.

5.1.3.2. Elección del conjunto final para clasificación multiclase

Los resultados obtenidos se reflejan en la tabla 5.4.

Tabla 5.4: Clasificación multiclase con *PCA*

Clasificador	Tiempo (s)	Entrenamiento			Generalización		
		Acc	MS	F1	Acc	MS	F1
Decision tree	*1.059*	0.999	0.895	0.999	0.939	0.000	0.976
Random forest	30.04	0.999	0.895	0.999	0.955	0.000	0.981
K-NN	**0.088**	0.951	0.000	0.981	0.936	0.000	0.975

Hay varios métodos que podemos usar para reducir el tamaño del conjunto de datos, como el *clustering* o variantes del método de *undersampling* ya utilizado en

clasificación binaria. A pesar de ello, estos métodos tienen una mayor complejidad de aplicación y la reducción de las dimensiones no es el objeto de este estudio. Por otro lado, esta decisión puede suponer algunos problemas al usar técnicas como *GridSearchCV* o la validación cruzada, ya que incrementan considerablemente el tiempo de entrenamiento.

Podemos ver en la tabla 5.4 que la métrica de mínima sensibilidad es 0 para todos los casos de test. Como ya se ha explicado en esta memoria, mide cómo de bien se clasifica la clase peor clasificada y un valor de 0 indica que alguna de las clases no se ha clasificado bien.

Como podemos ver en la matriz de confusión representada en la imagen 5.1, algunas de las clases con menos patrones tienen dificultades para obtener una buena clasificación debido a la falta de información en el entrenamiento. Algunos clasificadores tienen la opción de asignar un peso a los patrones de cada clase inversamente proporcional al número de patrones de la clase, de manera que todas las clases tengan el mismo peso en el entrenamiento, pero no se consiguen mejores resultados.

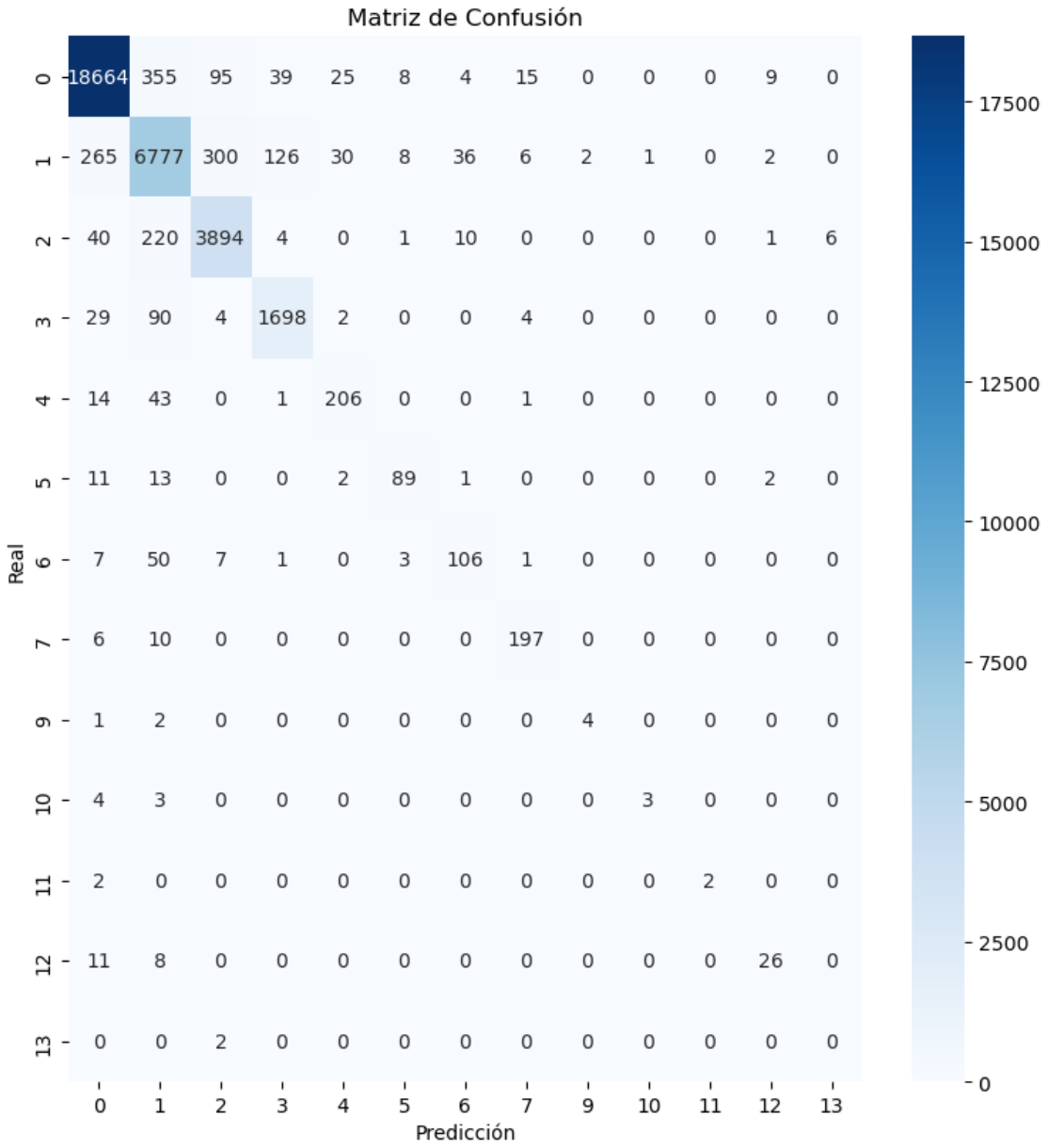


Figura 5.1: Matriz de confusión para la clasificación multiclase

Según el estudio *Malware Behavior Analysis: Learning and Understanding Current Malware Threats* [79], algunos de los tipos de *malware* que tenemos con menos patrones, se pueden agrupar en algunas de las clases más representadas de nuestro conjunto de datos. En este estudio se comenta que *p2p-worm* añade un comportamiento específico al comportamiento de un gusano, generando problemas de red y de pérdida de datos. Algo similar pasa con *Gamethief trojan*. De esta forma podemos agrupar estos patrones a sus respectivas clases similares sin perder efectividad a la hora de clasificar y además eliminar así dos de las clases que nos pueden dar problemas por falta de información.

Por otro lado, se han planteado dos formas de solucionar este problema, aunque ambas presentan inconvenientes:

- Eliminar las clases menos representadas. Tiene el riesgo de no reconocer un nuevo patrón si es de un tipo distinto de *malware*.
- Agruparlas en una nueva clase que represente varios tipos de *malware*. En este caso estamos suponiendo que los patrones agrupados tienen unas características similares.

Finalmente hemos decidido agrupar las clases con menos de 30 patrones en una nueva categoría *otros*. Por número de patrones sería recomendable agrupar también la clase *virus*, pero podría tener demasiado peso en la categoría otros y hemos considerado que es lo suficientemente relevante como para estudiarla por separado.

En la tabla 5.6 podemos ver que, aunque mejoramos la mínima sensibilidad, no se producen unas mejoras significativas en la precisión de clasificación pero dada la alta precisión presentada por los modelos y la mejora en la mínima sensibilidad puede considerarse una buena actualización. Podemos ver la nueva codificación en la tabla 5.5

Tabla 5.5: Nueva codificación de las clases de *malware*.

Categoría	Codificación	Nº de patrones	Porcentaje
benign	0	77142	53.78 %
trojan	1	29978	20.90 %
worm	2	16713	11.65 %
backdoor	3	7331	5.11 %
downloader	4	1031	0.72 %
informationstealer	5	448	0.31 %
dropper	6	715	0.50 %
ransomware	7	821	0.57 %
virus	8	192	0.13 %
otros	9	64	0.04 %

Tabla 5.6: Clasificación multiclase con la nueva codificación.

Clasificador	Tiempo (s)	Entrenamiento			Test		
		Acc	MS	F1	Acc	MS	F1
Decission tree	1.220	0.998	0.992	0.998	0.938	0.670	0.975
Random forest	31.201	0.998	0.993	0.998	0.953	0.670	0.980
K-NN	0.083	0.951	0.431	0.980	0.936	0.333	0.974

Por último, se han considerado otras opciones para mejorar la clasificación de las clases minoritarias, pero podrían exceder la complejidad de este proyecto:

- Utilizar métodos de sobremuestreo, ya mencionados en la sección 2.1.1.2, que consisten en aumentar la cantidad de patrones de estas clases de forma sintética.
- Utilizar métodos jerárquicos que primero clasifiquen usando la categoría *otros*, para después dividirla en sus diferentes clases y entrenar un modelo específico.

5.2. Preparación del entorno

5.2.1. Protocolo de experimentación y validación

En esta sección se establecen las condiciones de evaluación del rendimiento de los modelos utilizados y se explican los procedimientos seguidos, las técnicas de validación usadas y los criterios que permiten medir de forma objetiva la calidad de las predicciones. Todo esto tiene objetivo de minimizar posibles sesgos, evitar el sobreajuste y obtener conclusiones fiables.

5.2.1.1. Diseño experimental

Inicialmente se han planteado tres formas de estructurar el diseño experimental y como se evaluarán posteriormente las pruebas. La primera ha sido comparar distintos modelos para cada tipo de clasificación. La segunda, comparar, clasificación binaria y multiclase para cada clasificador. Por último, se ha planteado la posibilidad de una combinación de ambas comparaciones. Este último caso se ha descarta porque, aunque puede ser interesante la comparación combinada por proporcionar una amplia visión del problema, duplica la carga de trabajo y puede exceder la complejidad del proyecto.

La segunda opción planteada puede servir para comparar el rendimiento de uno o varios modelos según la naturaleza del problema y realizar un análisis de coste computacional. Son aspectos interesantes a estudiar, pero no entran dentro de los objetivos de este estudio.

Finalmente se ha seleccionado la primera opción. Aunque el problema de la detección de *malware* puede enfocarse tanto para la simple detección de un programa malicioso como para identificar a que tipo pertenece, los problemas de clasificación binaria y multiclase tienen enfoques muy diferentes. Por otro lado, el conjunto de datos usado para clasificación multiclase contiene varias clases con muy pocos patrones y la comparación podría no ser justa.

5.2.1.2. Validación de resultados

Para evitar sesgos y resultados poco concluyentes se han empleado varias técnicas.

- **Validación cruzada**: Se ha usado el parámetro `cv` de `GridSearchCV`. En general se han usado 5, aunque en algunos casos ha sido necesario ajustarlo por tiempo.
- **Validación cruzada estratificada adaptativa**: la función `cv()` que encontramos en el Anexo A.4 ajusta el numero de particiones en caso de que una clase tenga menos muestras que particiones indicadas.
- **Particion entrenamiento/prueba**: se ha dividido el conjunto de datos en un 75-25 para entrenamiento y pruebas respectivamente usando la variable `random_state` con la semilla usada en las pruebas.
- **Repetición con semillas aleatorias**: para repetir los experimentos con 10 semillas y tener una visión más amplia.
- **Ajuste de pesos de clase**: mediante `class_weight = "balanced"` en los clasificadores en los que se encuentra disponible.

A pesar de todas estas técnicas, es bastante probable que las clases extremadamente minoritarias del conjunto de datos para la clasificación multiclase pueden tener una influencia muy limitada.

5.2.1.3. Reproducibilidad

Durante el desarrollo del código y de las pruebas, se han adoptado diferentes medidas para garantizar que las comparaciones entre modelos sean justas.

1. **Fijación de semillas:**
   Se ha hecho uso de una semilla controlada dentro de un bucle para repetir el experimento. Con ella se controla:

   - La partición aleatoria de test y entrenamiento.
   - la inicialización interna de los clasificadores que aceptan `randon_state`.

2. **Número de repeticiones:**
   Si bien el número de repeticiones es ajustable dentro del código utilizado, para asegurar una justa comparación y por las limitaciones del equipo, se han usado 10 semillas en todos los experimentos. Esto permite obtener la media y la desviación típica de las métricas y reducir la variabilidad.

3. **Control de parámetros:**
   Los hiperparámetros se optimizan con `GridSearchCV` usando la misma rejilla para todas las semillas para poder tener una comparación coherente.

Estas medidas permiten obtener los mismo resultados si se usan las mismas semillas, configuraciones y conjunto de datos.

5.2.1.4. Control de parámetros

Para la optimización de hiperparámetros hemos usado búsqueda en rejilla de `GridSearchCV`. Esta técnica hace pruebas con todas las combinaciones posibles de los parámetros proporcionados y usa validación cruzada para garantizar la robustez de los resultados. El problema con esta técnica es el elevado número de pruebas, ya que se prueban todas las combinaciones de parámetros posibles en cada uno de los conjuntos de la validación cruzada, lo que eleva el tiempo necesario de manera considerable.

Una opción considerada y probada para evitar esta limitación es la búsqueda aleatoria de `RandomizedSearchCV`, que permite establecer un número máximo de combinaciones a probar y puede reducir considerablemente el número de combinaciones evaluadas. El inconveniente que ha surgido con esta técnica es que al disponer de un *hardware* muy limitado, la cantidad de combinaciones usadas es pequeña y limitar aún más con la búsqueda aleatoria puede suponer que los resultados sean menos representativos.

La rejilla se ha establecido para cada modelo en función de las limitaciones del equipo, el tiempo necesario para el entrenamiento de cada modelo y cuanto influye ese parámetro en el tiempo de entrenamiento y el peso que tiene en los resultados.

5.3. Implementación y pruebas

En esta sección se describe, principalmente, la estructura del código empleado para realizar los experimentos y el procedimiento seguido dentro del mismo para entrenar y evaluar los modelos seleccionados. Además se va a tratar la preparación del conjunto de datos, es decir, cómo se cargan y cómo se divide la información para realizar el entrenamiento y las pruebas. Por último, se tratará la forma que hemos seguido para presentar los resultados y las métricas que se han mencionado en la sección 4.2.5.

5.3.1. Procedimiento de entrenamiento y evaluación

El planteamiento seguido para entrenar los diferentes modelos ha sido usar `GridSearchCV` para ajustar los modelos de clasificación con los mejores parámetros posibles. Para obtener una visión más amplia y más justa del problema, se ha repetido el entrenamiento, con las mismas 10 semillas para todos los modelos. Con esto conseguimos que el experimento sea controlado y reproducible, ya que para un mismo modelo, una misma semilla y la misma rejilla de parámetros, obtendremos siempre los mismos resultados. Una vez calculadas las métricas seleccionadas en la sección 4.2.5, se calcula la media y la desviación típica de todas ellas para usarlas como valor final de comparación entre modelos.

5.3.2. Preparación y uso de los conjuntos de datos

Además del tratamiento previo del conjunto de datos realizado en la sección 5.1, es necesario procesar la información antes de entrenar. Con la función `load`, cargamos el conjunto de datos en dos matrices de `Numpy`, la matriz de información y la matriz de clases. La matriz de patrones de entrada se normaliza haciendo uso de la clase `MinMaxScaler` del módulo `preprocessing` de `Scikit-Learn`.

Por último, haciendo uso de la función `train_test_split`, dividimos el conjunto de datos en test y entrenamiento. Esto se hace dentro del bucle y para cada semilla con el objetivo de tener una evaluación más robusta, ya que permite tener una división distinta y controlada para cada semilla.

5.3.3. Métricas y análisis de resultados

Para calcular las métricas se ha usado la función `minimum_sensitivity` para la métrica mínima sensibilidad. Esta se encuentra disponible en el módulo `metrics` de la librería `dlordinal`. Para calcular la exactitud o *accuracy* del entrenamiento, se ha usado la función `accuracy_score` disponible en el módulo `metrics` de la librería `Scikit-Learn`, en el que también encontramos `f1_score` para calcular el Valor-F1.

Finalmente, una vez calculados los resultados para todas las semillas, se guardan en un objeto `DataFrame` de `Pandas` con el formato que se muestra en el ejemplo del Anexo A.5. Haciendo uso de los métodos `mean` y `std` de esta clase, se obtiene la media y la desviación típica de todas las semillas.

Capítulo 6

Resultados y discusión

En este capítulo se presentan los resultados obtenidos tras aplicar los distintos algoritmos de clasificación sobre los conjuntos de datos preparados en las fases anteriores. El objetivo principal es evaluar el rendimiento de cada modelo bajo diferentes escenarios y métricas, con el fin de identificar sus fortalezas y limitaciones en la detección de *malware*.

Para ello, se analizan tanto los experimentos realizados en el problema binario, donde se distingue entre *software* malicioso y benigno, como en el problema multiclase, en el que se busca identificar el tipo específico de amenaza. Los clasificadores se evalúan considerando como métricas: la sensibilidad mínima, el Valor-F1 y la precisión de los modelos.

Asimismo, se discute la capacidad de generalización de cada modelo, observando las diferencias de rendimiento entre los conjuntos de entrenamiento y prueba, así como la influencia de la variabilidad introducida por el desbalance de clases. Con este análisis se pretende ofrecer una visión comparativa que facilite la elección del modelo más adecuado en un contexto práctico de detección de amenazas.

6.1. Clasificación binaria

En primer lugar se expondrán los resultados obtenidos en clasificación binaria.

6.1.1. Árboles de decisión

La tabla 6.1 muestra los resultados de la clasificación binaria utilizando el modelo *DecisionTreeClassifier*, evaluados en 10 ejecuciones distintas con diferentes estados.

Para el entrenamiento, la precisión (*Acc*), la sensibilidad mínima (*MS*) y el *Valor-F1* alcanzan valores muy próximos a 1 en todas las ejecuciones. En cuanto al test, la precisión oscila entre 0.942 y 0.953, mientras que la sensibilidad mínima se sitúa entre 0.935 y 0.946. El Valor-F1 se mantiene constante en 1 en todas las ejecuciones.

El gráfico 6.1 compara la distribución de los valores de *accuracy* en entrenamiento y en test para el clasificador basado en árboles de decisión. Se observa que en entrenamiento son consistentemente cercanos a 1, sin apenas variabilidad, lo que indica que el modelo es capaz de ajustarse casi perfectamente a los datos de entrenamiento.

En contraste, los valores en test muestran una ligera caída, con una variabilidad mayor que en entrenamiento. Esta diferencia refleja que el modelo generaliza de forma aceptable, aunque la brecha respecto al rendimiento en entrenamiento sugiere la existencia de cierto sobreajuste.

El uso combinado de violinplot y boxplot permite apreciar tanto la concentración de los valores en torno a la media como la dispersión entre diferentes ejecuciones. En este caso, el test mantiene una distribución compacta, sin valores atípicos extremos, lo que aporta robustez a la evaluación del modelo.

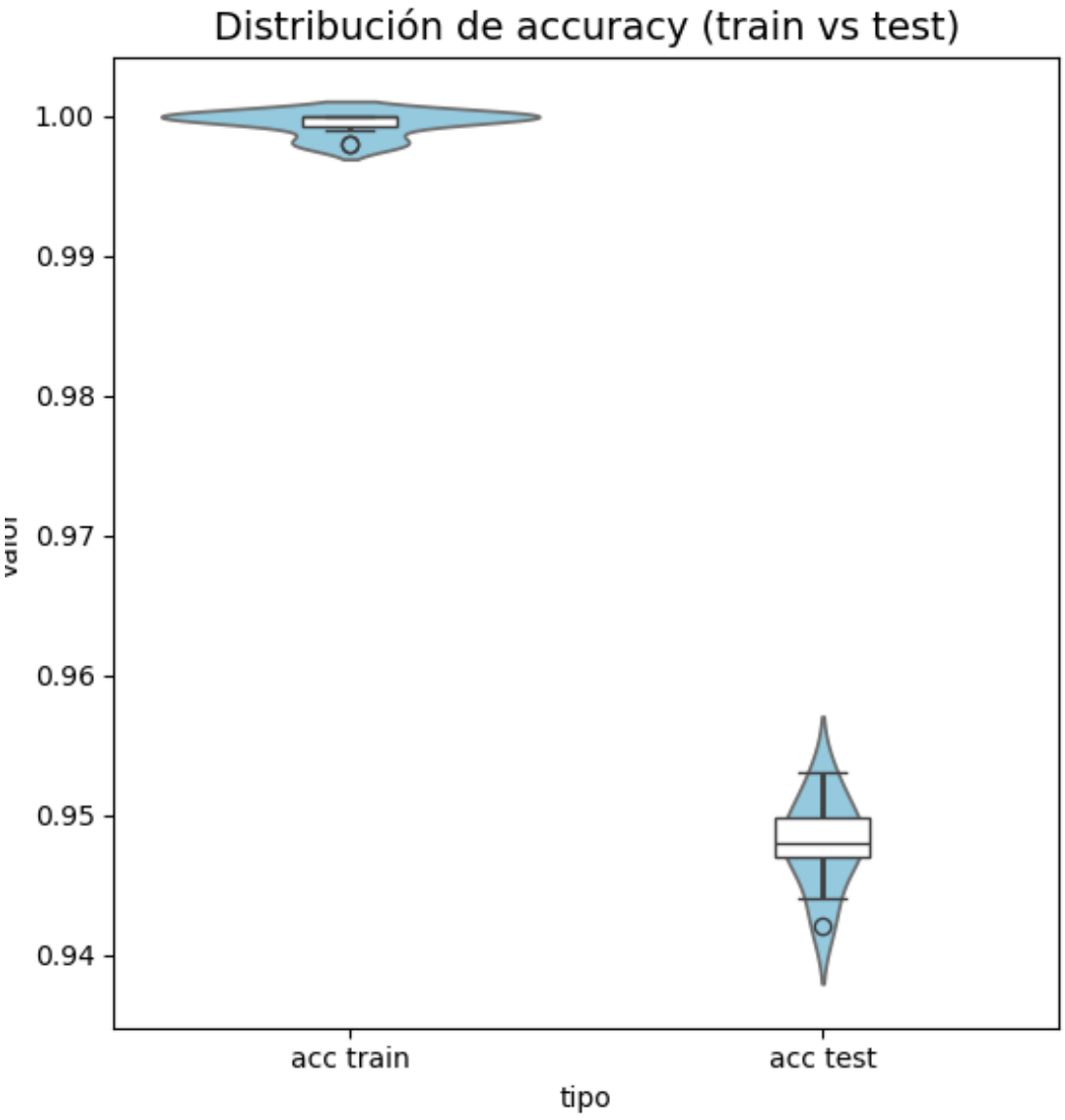


Figura 6.1: Boxplot con violinplot para árboles de decisión

Tabla 6.1: Clasificación binaria con *DecisionTreeClassifier*

	Entrenamiento			Generalización		
Semilla	Acc	MS	F1	Acc	MS	F1
0	**1.000**	**1.000**	**1.000**	*0.951*	0.942	**1.000**
1	*1.000*	*1.000*	*1.000*	0.944	0.935	*1.000*
2	1.000	1.000	1.000	0.942	0.935	1.000
3	1.000	1.000	1.000	0.948	0.938	1.000
4	1.000	1.000	1.000	**0.953**	**0.946**	1.000
5	0.998	0.997	1.000	0.947	0.936	1.000
6	0.998	0.997	1.000	0.947	0.941	1.000
7	1.000	1.000	1.000	0.949	*0.946*	1.000
8	0.999	0.998	1.000	0.948	0.937	1.000
9	1.000	1.000	1.000	0.950	0.940	1.000
Mean	0.999	0.999	1.000	0.948	0.940	1.000
STD	0.001	0.001	0.000	0.003	0.004	0.000

6.1.2. *Random forest*

En la tabla 6.2 se muestran los resultados de la clasificación binaria utilizando el modelo RandomForestClassifier para diferentes estados aleatorios.

En el conjunto de entrenamiento, las métricas presentan valores altos y consistentes: la exactitud oscila entre 0.978 y 0.989, la métrica de sensibilidad mínima entre 0.920 y 0.956, y la puntuación F1 entre 0.988 y 0.992.

En el conjunto de prueba, la exactitud mantiene valores muy estables alrededor de 0.936–0.941. La sensibilidad mínima muestra una mayor variabilidad, con valores que van desde 0.400 hasta 0.609, mientras que la métrica F1 se mantiene muy uniforme, entre 0.971 y 0.974.

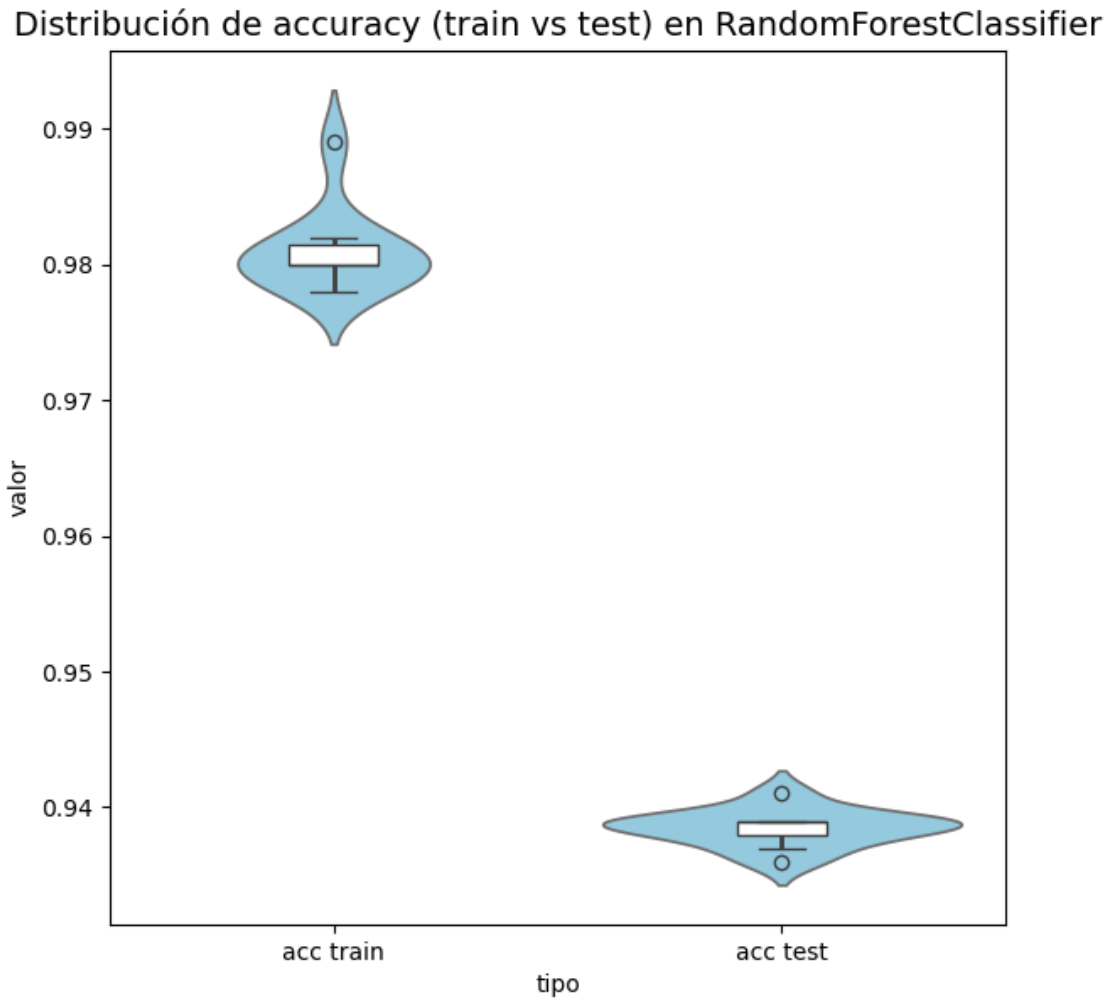


Figura 6.2: Boxplot con violinplot para *Random forest*

El modelo muestra un rendimiento muy alto y consistente en precisión tanto en el entrenamiento como en el test. En el gráfico 6.2, se observa que la distribución de accuracy es estrecha, con valores muy concentrados en torno a 0.98 en entrenamiento y 0.94 en test, lo que indica estabilidad en ambas fases.

El F1-Score también refleja una gran solidez, con valores prácticamente constantes en todas las ejecuciones, lo que sugiere un buen equilibrio entre precisión y exhaustividad en la clasificación de las dos clases.

Sin embargo, la métrica de sensibilidad mínima (MS) introduce un matiz importante: aunque en entrenamiento se mantiene elevada con poca dispersión, en test presenta una gran variabilidad. El rango de valores oscila entre 0.400 y 0.609 según la semilla utilizada, lo que se traduce en distribuciones más amplias en los gráficos. Esto indica que, en algunos casos, el modelo no logra identificar de forma adecuada los ejemplos más difíciles de una de las clases, comprometiendo la robustez de la clasificación en escenarios concretos.

Tabla 6.2: Clasificación binaria con *RandomForestClassifier*

	Entrenamiento			Generalización		
Semilla	Acc	MS	F1	Acc	MS	F1
0	0.980	0.928	0.989	0.939	0.524	0.972
1	0.980	0.929	0.990	*0.939*	0.500	0.973
2	0.980	0.927	0.989	**0.941**	0.429	**0.974**
3	0.979	0.923	0.989	0.939	0.444	0.973
4	0.982	0.931	0.990	0.939	0.500	*0.974*
5	0.980	0.929	0.990	0.937	**0.609**	0.971
6	0.978	0.920	0.988	0.938	0.550	0.971
7	0.980	0.929	0.990	0.939	*0.562*	0.972
8	*0.982*	*0.934*	*0.991*	0.938	0.489	0.971
9	**0.989**	**0.956**	**0.992**	0.936	0.400	0.972
Mean	0.981	0.931	0.990	0.939	0.501	0.972
STD	0.003	0.010	0.001	0.001	0.064	0.001

6.1.3. *K-NN*

En la tabla 6.3 se muestran los resultados obtenidos con el clasificador KNeighborsClassifier bajo diferentes semillas.

En la fase de entrenamiento, todas las semillas reportan valores de Acc, MS y F1 iguales a 1.000, lo que refleja una completa uniformidad en las métricas. La media confirma este comportamiento perfecto, con desviaciones estándar nulas en las tres métricas.

En la fase de generalización, la precisión presenta valores que oscilan entre 0.939 y 0.949, con una media de 0.947 y una desviación estándar reducida de 0.003. MS toma valores entre 0.926 y 0.940, con una media de 0.934 y una desviación estándar de 0.004, mostrando ligeras variaciones según la semilla utilizada. Finalmente, F1 alcanza en todos los casos el valor de 1.000, con media y desviación estándar constantes.

Tabla 6.3: Clasificación binara con *KNeighborsClassifier*

	Entrenamiento			Generalización		
Semilla	Acc	MS	F1	Acc	MS	F1
0	1.000	1.000	1.000	0.947	0.935	1.000
1	1.000	1.000	1.000	0.949	0.938	1.000
2	1.000	1.000	1.000	0.939	0.926	1.000
3	1.000	1.000	1.000	0.949	0.937	1.000
4	1.000	1.000	1.000	0.949	0.936	1.000
5	1.000	1.000	1.000	0.946	0.932	1.000
6	1.000	1.000	1.000	0.946	0.931	1.000
7	1.000	1.000	1.000	0.944	0.934	1.000
8	1.000	1.000	1.000	0.947	0.935	1.000
9	1.000	1.000	1.000	0.949	0.940	1.000
Mean	1.000	1.000	1.000	0.947	0.934	1.000
STD	0.000	0.000	0.000	0.003	0.004	0.000

El gráfico 6.3 muestra como para el entrenamiento, todos los valores de *accuracy* se encuentran exactamente en 1.000, lo que se refleja en el `violinplot` como una única línea y en el `boxplot` como una caja colapsada. Esto indica que el modelo clasifica perfectamente todos los patrones del conjunto de entrenamiento en cada ejecución.

Para la generalización, los valores oscilan ligeramente entre 0.939 y 0.949. El `violinplot` muestra una distribución muy estrecha alrededor de la media, y el `boxplot` confirma que la mediana es cercana a 0.947, con una ligera variabilidad reflejada por los bigotes.

Se aprecia un entrenamiento perfecto y una generalización muy alta y consistente, con poca dispersión en los resultados de test, aunque podemos ver una ligera caída en las métricas.

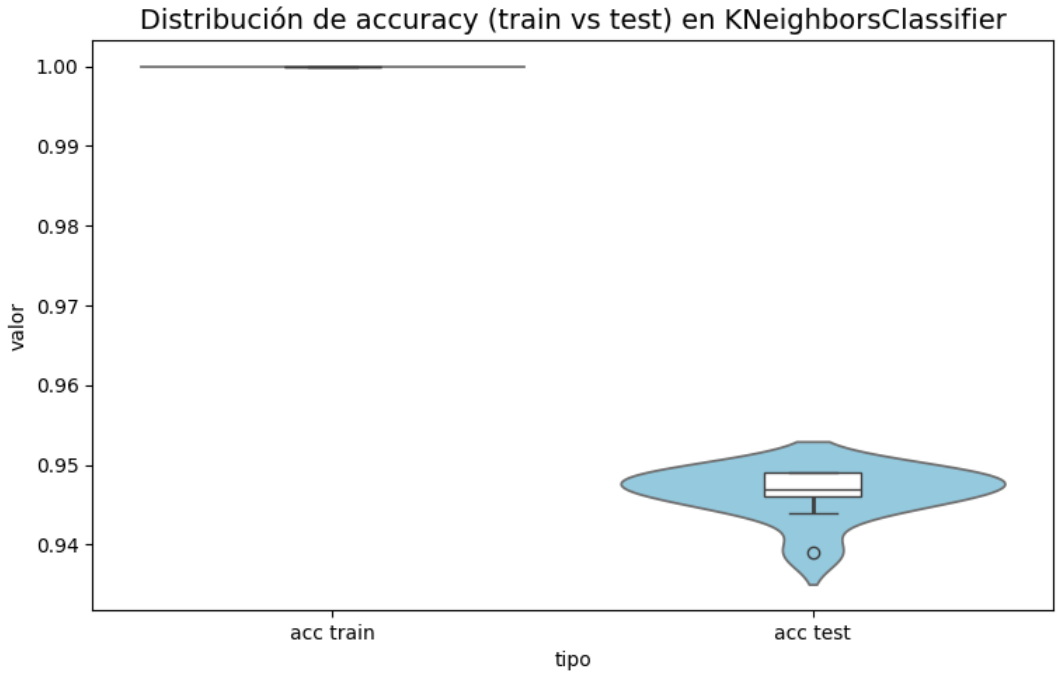
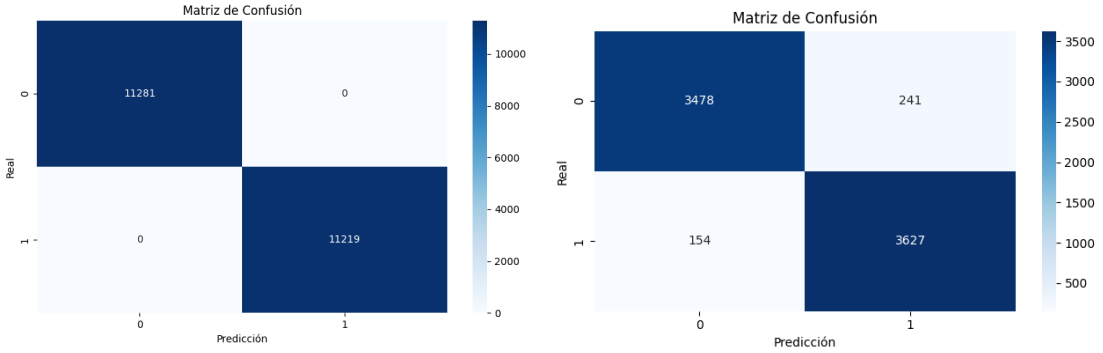


Figura 6.3: Boxplot con violinplot para *K-NN*

Para ver más clara la caída en las métricas anteriores, podemos hacer uso de las matrices de confusión reflejadas en la figura 6.4. En ella podemos ver que la

clasificación en entrenamiento es perfecta, acertando en todos los patrones, mientras que el para la generalización los resultados son peores por un aumento significativo de los falsos positivos y los falsos negativos. Esto podría significar un ligero sobreajuste del modelo.



(a) Título subfigura 1 (b) Título subfigura 2

Figura 6.4: Matriz de confusión en *K-NN*

6.1.4. Máquinas de vectores de soporte

En la tabla 6.5 podemos ver como para el conjunto de entrenamiento, la precisión se mantiene en valores cercanos a 0.76 en todas las ejecuciones, mientras que la mínima sensibilidad (MS) muestra variaciones entre 0.656 y 0.704.

En el conjunto de generalización, todas las métricas presentan unos resultados muy similares a los de entrenamiento. Esto se puede apreciar claramente observando la desviación estándar (*STD*), que indica una baja variabilidad en todas las métricas.

Tabla 6.4: Clasificación binaria con *SVC*

	Entrenamiento			Generalización		
Semilla	Acc	MS	F1	Acc	MS	F1
0	0.757	0.656	1.000	0.764	0.672	1.000
1	0.761	0.671	1.000	0.769	0.681	1.000
2	0.766	0.702	1.000	0.757	0.699	1.000
3	0.762	0.700	1.000	0.766	0.704	1.000
4	0.760	0.684	1.000	0.768	0.696	1.000
5	0.761	0.663	1.000	0.753	0.655	1.000
6	0.762	0.702	1.000	0.762	0.683	1.000
7	0.763	0.699	1.000	0.759	0.697	1.000
8	0.766	0.704	1.000	0.758	0.693	1.000
9	0.760	0.662	1.000	0.758	0.666	1.000
Mean	0.762	0.684	1.000	0.762	0.685	1.000
STD	0.003	0.020	0.000	0.005	0.016	0.000

Se muestran valores muy consistentes tanto en el conjunto de entrenamiento como en el de test. En el gráfico 6.5, las distribuciones para ambos conjuntos aparecen concentradas en torno al 0.76, sin grandes variaciones entre diferentes semillas. Esto queda reforzado por la baja dispersión que se observa en el `boxplot` y por la forma compacta del `violinplot`, que refleja que no existen valores extremos significativos.

Aunque no se han obtenido los mejores resultados con las máquinas de vectores de soporte, es interesante la estabilidad que consiguen a la hora de generalizar. Esta cercanía entre ambas distribuciones indica que el modelo mantiene un rendimiento estable al generalizar sobre datos no vistos.

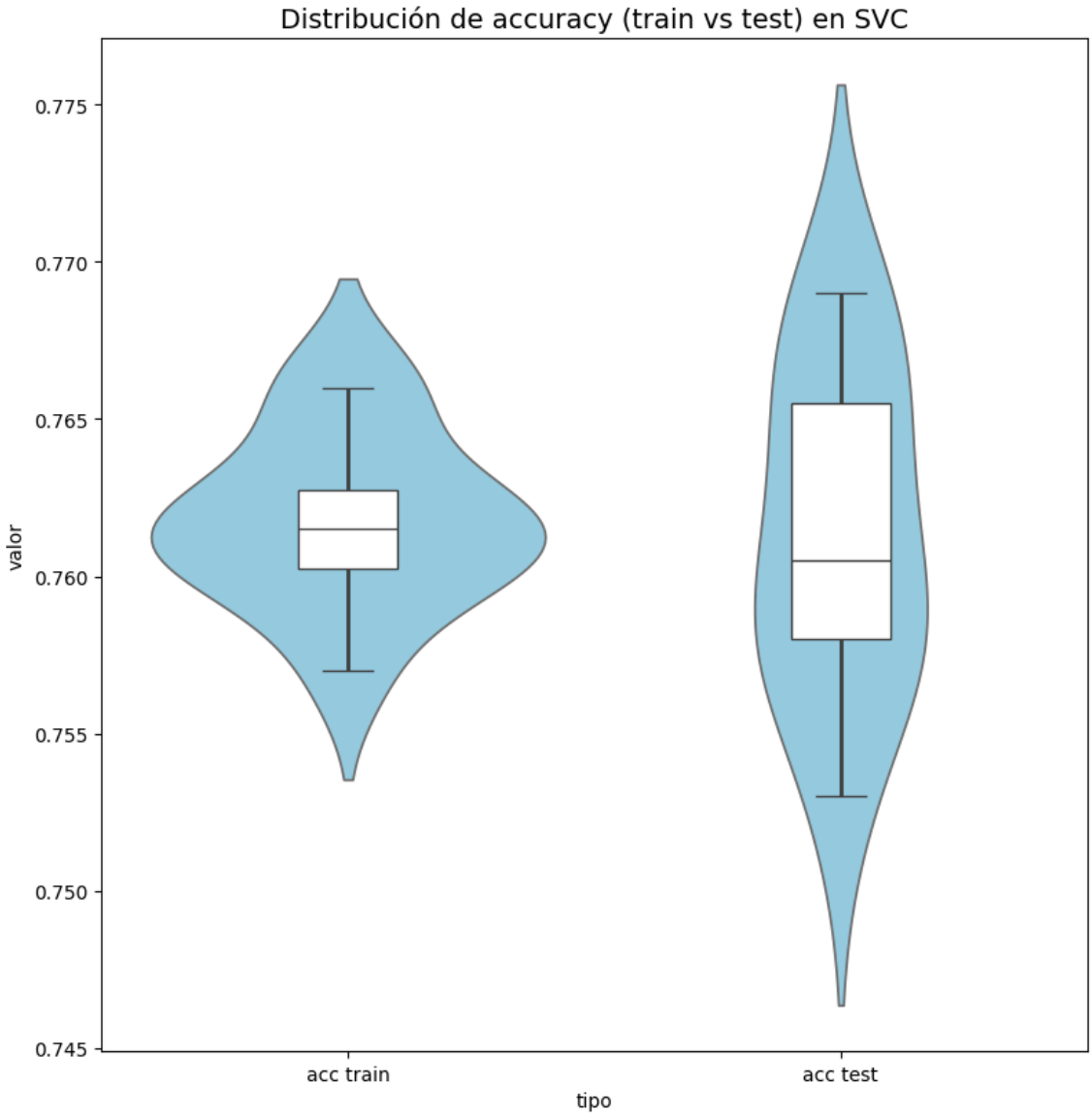


Figura 6.5: Boxplot con violinplot para *SVC*

6.1.5. *Ridge*

Tabla 6.5: Clasificación binaria con *RidgeClassifier*

	Entrenamiento			Generalización		
Semilla	Acc	MS	F1	Acc	MS	F1
0	0.649	0.549	1.000	0.648	0.530	1.000
1	0.645	0.558	1.000	0.655	0.569	1.000
2	0.652	0.573	1.000	0.645	0.564	1.000
3	0.649	0.567	1.000	0.653	0.570	1.000
4	0.651	0.573	1.000	0.651	0.573	1.000
5	0.647	0.562	1.000	0.648	0.558	1.000
6	0.648	0.556	1.000	0.650	0.573	1.000
7	0.651	0.571	1.000	0.650	0.573	1.000
8	0.651	0.564	1.000	0.639	0.551	1.000
9	0.650	0.563	1.000	0.645	0.551	1.000
Mean	0.649	0.564	1.000	0.648	0.561	1.000
STD	0.002	0.008	0.000	0.005	0.014	0.000

6.1.6. Perceptrón multicapa

Tabla 6.6: Clasificación binaria con *MLPClassifier*

	Entrenamiento			Generalización		
Semilla	Acc	MS	F1	Acc	MS	F1
0	0.783	0.771	1.000	0.789	0.778	1.000
1	0.788	0.736	1.000	0.792	0.740	1.000
2	0.788	0.750	1.000	0.782	0.739	1.000
3	0.733	0.605	1.000	0.737	0.609	1.000
4	0.767	0.759	1.000	0.769	0.760	1.000
5	0.790	0.736	1.000	0.783	0.730	1.000
6	0.777	0.772	1.000	0.783	0.781	1.000
7	0.774	0.767	1.000	0.770	0.763	1.000
8	0.778	0.704	1.000	0.772	0.705	1.000
9	0.788	0.762	1.000	0.784	0.751	1.000
Mean	0.776	0.736	1.000	0.776	0.736	1.000
STD	0.017	0.051	0.000	0.016	0.050	0.000

6.1.7. *Light gradient boosting machine*

Tabla 6.7: Clasificación binaria con *LGBMClassifier*

	Entrenamiento			Generalización		
Semilla	Acc	MS	F1	Acc	MS	F1
0	0.984	0.981	1.000	0.953	0.952	1.000
1	0.984	0.980	1.000	0.951	0.947	1.000
2	0.985	0.983	1.000	0.949	0.946	1.000
3	0.985	0.982	1.000	0.952	0.951	1.000
4	0.984	0.981	1.000	0.950	0.945	1.000
5	0.985	0.981	1.000	0.949	0.948	1.000
6	0.985	0.982	1.000	0.952	0.949	1.000
7	0.986	0.984	1.000	0.948	0.947	1.000
8	0.984	0.979	1.000	0.953	0.952	1.000
9	0.989	0.989	1.000	0.953	0.950	1.000
Mean	0.985	0.982	1.000	0.951	0.949	1.000
STD	0.002	0.003	0.000	0.002	0.002	0.000

6.1.8. Discusión de los resultados

6.2. Clasificación multiclase

6.2.1. Árboles de decisión

6.2.2. *Random forest*

6.2.3. *K-NN*

6.2.4. Máquinas de vectores de soporte

En este caso, el proceso de entrenamiento presentó una mayor complejidad y dificultad para obtener resultados comparables con los de otros modelos evaluados, principalmente debido a las limitaciones del equipo utilizado. El elevado tiempo requerido para el entrenamiento sin ajuste de parámetros, junto con los resultados poco satisfactorios obtenidos para las dos semillas empleadas —con una precisión aproximada del 20 %—, motivaron la decisión de no continuar con las máquinas de

Tabla 6.8: Clasificación multiclase con *DecisionTreeClassifier*

	Entrenamiento			Test		
Estado aleatorio	Acc	MS	F1	Acc	MS	F1
0	0.980	0.928	0.989	0.939	0.524	0.972
1	0.980	0.929	0.990	0.939	0.500	0.973
2	0.980	0.927	0.989	0.941	0.429	0.974
3	0.979	0.923	0.989	0.939	0.444	0.973
4	0.982	0.931	0.990	0.939	0.500	0.974
5	0.980	0.929	0.990	0.937	0.609	0.971
6	0.978	0.920	0.988	0.938	0.550	0.971
7	0.980	0.929	0.990	0.939	0.562	0.972
8	0.982	0.934	0.991	0.938	0.489	0.971
9	0.989	0.956	0.992	0.936	0.400	0.972
Mean	0.981	0.931	0.990	0.939	0.501	0.972
STD	0.003	0.010	0.001	0.001	0.064	0.001

vectores de soporte para la clasificación multiclase. No obstante, estos resultados no indican que el modelo sea inadecuado para el problema planteado, sino que tiene una mayor exigencia en cuanto a los recursos necesarios para su entrenamiento.

6.2.5. *Ridge*

6.2.6. Perceptrón multicapa

6.2.7. *Light gradient boosting machine*

Tabla 6.9: Clasificación multiclase con *RandomForestClassifier*

	Entrenamiento			Test		
Estado aleatorio	Acc	MS	F1	Acc	MS	F1
0	0.981	0.926	0.990	0.951	0.524	0.977
1	0.981	0.926	0.990	0.953	0.735	0.978
2	0.981	0.926	0.990	0.954	0.429	0.978
3	0.980	0.923	0.990	0.954	0.500	0.978
4	0.981	0.926	0.990	0.954	0.500	0.979
5	0.981	0.927	0.990	0.952	0.638	0.977
6	0.980	0.923	0.990	0.952	0.550	0.977
7	0.981	0.927	0.991	0.954	0.500	0.978
8	0.981	0.926	0.990	0.953	0.471	0.978
9	0.981	0.926	0.990	0.953	0.400	0.978
Mean	0.981	0.926	0.990	0.953	0.525	0.978
STD	0.000	0.002	0.000	0.001	0.098	0.001

Tabla 6.10: Clasificación multiclase con *KNeighborsClassifier*

	Entrenamiento			Test		
Estado aleatorio	Acc	MS	F1	Acc	MS	F1
0	0.994	0.811	0.997	0.940	0.524	0.976
1	0.994	0.794	0.996	0.943	0.500	0.977
2	0.994	0.811	0.997	0.940	0.357	0.977
3	0.994	0.815	0.996	0.942	0.389	0.978
4	0.994	0.810	0.997	0.941	0.375	0.976
5	0.994	0.807	0.996	0.940	0.435	0.976
6	0.994	0.802	0.996	0.941	0.500	0.976
7	0.994	0.849	0.996	0.940	0.500	0.976
8	0.994	0.817	0.996	0.939	0.529	0.976
9	0.994	0.834	0.996	0.940	0.400	0.976
Mean	0.994	0.815	0.996	0.941	0.451	0.976
STD	0.000	0.016	0.000	0.001	0.067	0.001

Tabla 6.11: Clasificación multiclase con *RidgeClassifier*

	Entrenamiento			Test		
Estado aleatorio	Acc	MS	F1	Acc	MS	F1
0	0.189	0.000	0.301	0.186	0.000	0.299
1	0.195	0.000	0.308	0.191	0.000	0.307
2	0.173	0.000	0.284	0.176	0.000	0.287
3	0.172	0.000	0.283	0.172	0.000	0.280
4	0.185	0.000	0.297	0.189	0.000	0.305
5	0.187	0.000	0.300	0.189	0.000	0.303
6	0.170	0.000	0.280	0.166	0.000	0.274
7	0.186	0.000	0.299	0.191	0.000	0.303
8	0.187	0.000	0.300	0.187	0.000	0.301
9	0.171	0.000	0.282	0.173	0.000	0.284
Mean	0.182	0.000	0.293	0.182	0.000	0.294
STD	0.009	0.000	0.010	0.009	0.000	0.012

Tabla 6.12: Clasificación multiclase con *MLPClassifier*

	Entrenamiento			Test		
Estado aleatorio	Acc	MS	F1	Acc	MS	F1
0	0.725	0.000	0.885	0.722	0.000	0.883
1	0.724	0.000	0.901	0.724	0.000	0.900
2	0.724	0.000	0.885	0.723	0.000	0.885
3	0.679	0.000	0.885	0.681	0.000	0.888
4	0.730	0.000	0.904	0.735	0.000	0.902
5	0.721	0.000	0.888	0.717	0.000	0.884
6	0.724	0.000	0.889	0.723	0.000	0.888
7	0.711	0.000	0.885	0.711	0.000	0.884
8	0.719	0.000	0.910	0.720	0.000	0.910
9	0.716	0.000	0.886	0.718	0.000	0.885
Mean	0.717	0.000	0.892	0.718	0.000	0.891
STD	0.014	0.000	0.009	0.014	0.000	0.009

Tabla 6.13: Clasificación multiclase con *LGBMClassifier*

	Entrenamiento			Test		
Estado aleatorio	Acc	MS	F1	Acc	MS	F1
0	0.938	0.821	0.965	0.916	0.600	0.953
1	0.936	0.820	0.964	0.916	0.735	0.953
2	0.890	0.749	0.941	0.884	0.357	0.936
3	0.323	0.000	0.460	0.327	0.000	0.460
4	0.888	0.747	0.940	0.880	0.500	0.936
5	0.938	0.828	0.967	0.917	0.565	0.955
6	0.893	0.758	0.943	0.881	0.550	0.935
7	0.936	0.821	0.964	0.917	0.562	0.952
8	0.891	0.750	0.941	0.880	0.588	0.933
9	0.893	0.760	0.942	0.883	0.467	0.936
Mean	0.853	0.706	0.903	0.840	0.492	0.895
STD	0.187	0.250	0.156	0.181	0.198	0.153

Capítulo 7

Conclusiones y recomendaciones

En este último capítulo del proyecto se expondrán las conclusiones, recomendaciones y mejoras de este proyecto.

7.1. Conclusiones de investigación

En cuanto a los resultados obtenidos en la investigación y dado el enfoque seleccionado en el capítulo 4, es necesario hacer una diferenciación entre clasificación binaria y clasificación multiclase.

7.1.1. Clasificación binaria

Si bien es cierto que los resultados obtenidos por los clasificadores más sencillos son muy altos en cuanto a precisión, todos ellos presentan una caída apreciable en la métrica de sensibilidad mínima. Esto significa que es bastante probable un sobreajuste de los modelos y una mala generalización, lo que puede llevar a demasiados fallos en una situación real. Por otro lado, `LGBMClassifier` tiene una precisión ligeramente inferior pero mayor estabilidad entre las métricas de entrenamiento y clasificación. A pesar de ser más complejo a la hora de entrenar e interpretar esta consistencia hace que sea una muy buena opción a tener en cuenta.

7.1.2. Clasificación multiclase

Este tipo de clasificación ha presentado varios problemas además del aumento necesario de tiempo por la configuración del conjunto de datos con todos los patrones. Por un lado, la fuerte caída de la mínima sensibilidad indica que los clasificadores tienden a priorizar las clases mayoritarias, dejando sin apenas capacidad de detección a las minoritarias. Esto provoca que, aunque la precisión o el valor-F1 puedan mantenerse razonablemente altos, el rendimiento real frente a todas las clases no es confiable.

Por otro lado, algunos modelos como `LGBMClassifier` presentan una alta varianza entre semillas. Esto implica que el conjunto de datos está fuertemente afectado por el desbalanceo, y que los modelos probados no garantizan una generalización. Además, algunos modelos presentan una sensibilidad mínima de 0, es decir, hay clases que directamente no se predicen en absoluto.

7.2. Recomendaciones

Las principales recomendaciones referentes a este estudio tienen su origen en las limitaciones del equipo con el que se han realizado las pruebas. A pesar de que se han obtenido muy buenos resultados tanto en entrenamiento como en test con algoritmos ligeros, es recomendable hacer pruebas con un *hardware* capaz de entrenar modelos algo más lentos para hacer unas pruebas realmente concluyentes. Por ejemplo, para el modelo de máquinas de vectores de soporte, ha sido muy difícil realizar el entrenamiento ajustando una rejilla relativamente completa de parámetros, por lo que los resultados podrían llegar a mejorar. En cuanto al perceptrón multicapa sí se han podido realizar más pruebas, pero a cada ajuste de parámetros que se ha probado mejoraban considerablemente los resultados. Un equipo más potente permitiría ajustar correctamente el modelo y obtener los mejores resultados posibles.

En cuanto al conjunto de datos, si bien es cierto que es amplio y en clasificación binaria funciona muy bien, para clasificación multiclase es insuficiente. Su principal problema se encuentra en el desbalanceo de clases. Como podemos ver en la tabla

5.5, en la cuarta clase más poblada el número de patrones es aproximadamente diez veces menos que en la más poblada. Esto implica que a la hora de entrenar, la clase más poblada influye mucho en el entrenamiento incluso aplicando varias técnicas para evitarlo. Sería recomendable hacer pruebas con distintos conjuntos de datos que dispongan de una mayor cantidad de muestras en clases minoritarias para poder hacer un entrenamiento equilibrado.

Bibliografía

[1] Bob Thomas Morris. Creeper. URL https://es.wikipedia.org/wiki/Creeper_(virus). Última consulta: 26 de Mayo de 2025.

[2] Arpanet. URL https://es.wikipedia.org/wiki/ARPANET. Última consulta: 3 de Septiembre de 2025.

[3] Gusano morris. URL https://es.wikipedia.org/wiki/Gusano_Morris. Última consulta: 15 de Junio de 2025.

[4] Cert. URL https://es.wikipedia.org/wiki/Equipo_de_Respuesta_ante_Emergencias_Inform%C3%A1ticas. Última consulta: 15 de Junio de 2025.

[5] Wannacry. URL https://es.wikipedia.org/wiki/WannaCry. Última consulta: 15 de Junio de 2025.

[6] Ransomware, . URL https://es.wikipedia.org/wiki/Ransomware. Última consulta: 15 de Junio de 2025.

[7] Wannacry: el ransomware que tiene «secuestrados» los sistemas de telefónica y de otras empresas. URL https://www.abc.es/tecnologia/redes/abci-wannacry-ransomware-tiene-secuestrados-sistemas-telefonica-y-otras-empresas-noticia.html?ref=https%3A%2F%2Fwww.google.com%2F. Última consulta: 23 de Junio de 2025.

[8] Detección basada en firmas, . URL https://fastercapital.com/es/tema/%C2%BFqu%C3%A9-es-la-detecci%C3%B3n-basada-en-la-firma.html. Última consulta: 23 de Junio de 2025.

[9] Machine learning. URL https://es.wikipedia.org/wiki/Aprendizaje_autom%C3%A1tico#Tipos_de_algoritmos. Última consulta: 23 de Junio de 2025.

[10] Estadisticas de ciberseguridad. URL https://purplesec.us/resources/cybersecurity-statistics/. Última consulta: 6 de Septiembre de 2025.

[11] A low complexity ml-based methods for malware classification. URL https://www.researchgate.net/publication/383827671_A_Low_Complexity_ML-Based_Methods_for_Malware_Classification. Última consulta: 2 de Agosto de 2025.

[12] La máquina tabuladora. URL https://es.wikipedia.org/wiki/Herman_Hollerith#La_m%C3%A1quina_tabuladora. Última consulta: 2 de Agosto de 2025.

[13] Rafael Prieto Meléndez, A Herrera, J Pérez, and Alejandro Padrón-Godínez. El modelo neuronal de mcculloch y pitts. interpretación comparativa del modelo. 10 2000. URL https://www.researchgate.net/publication/343141076_EL_MODELO_NEURONAL_DE_McCULLOCH_Y_PITTS_Interpretacion_Comparativa_del_Modelo.

[14] Balanceo de datos. URL https://es.linkedin.com/pulse/balanceo-de-datos-ainad-empresarial#:~:text=%C2%BFQu%C3%A9%20es%20el%20balanceo%20de,en%20nuestro%20conjunto%20de%20datos. Última consulta: 2 de Agosto de 2025.

[15] Joaquín García Abad. Comparativa de técnicas de balanceo de datos. aplicación a un caso real para la predicción de fuga de clientes. URL https://digibuo.uniovi.es/dspace/bitstream/handle/10651/60629/TFM_Joaqu%C3%ADnGarc%C3%ADaAbad.pdf?sequence=4. Última consulta: 2 de Agosto de 2025.

[16] Jason Brownlee. Random oversampling and undersampling for imbalanced classification. URL https://machinelearningmastery.com/

random-oversampling-and-undersampling-for-imbalanced-classification/. Última consulta: 4 de Agosto de 2025.

[17] Cnn (condensed nearest neighbors). URL https://abhic159.medium.com/cnn-condensed-nearest-neighbors-3261bd0c39fb. Última consulta: 4 de Agosto de 2025.

[18] Tusneem Elhassan, Aljourf M, Al-Mohanna F, and Mohamed Shoukri. Classification of imbalance data using tomek link (t-link) combined with random under-sampling (rus) as a data reduction method. *Global Journal of Technology and Optimization*, 01, 01 2016. doi: 10.4172/2229-8711.S1111.

[19] Roberto Alejo, José Sotoca, Rosa Valdovinos, and P. Toribio. Edited nearest neighbor rule for improving neural networks classifications. pages 303–310, 06 2010. doi: 10.1007/978-3-642-13278-0_39.

[20] Dina Elreedy and Amir F. Atiya. A comprehensive analysis of synthetic minority oversampling technique (smote) for handling class imbalance. *Information Sciences*, 505:32–64, 2019. ISSN 0020-0255. doi: https://doi.org/10.1016/j.ins.2019.07.070. URL https://www.sciencedirect.com/science/article/pii/S0020025519306838.

[21] Haibo He, Yang Bai, Edwardo Garcia, and Shutao Li. Adasyn: Adaptive synthetic sampling approach for imbalanced learning. *Proceedings of the International Joint Conference on Neural Networks*, pages 1322 – 1328, 07 2008. doi: 10.1109/IJCNN.2008.4633969.

[22] Maldición de la dimensión, . URL https://es.wikipedia.org/wiki/Maldici%C3%B3n_de_la_dimensi%C3%B3n. Última consulta: 11 de Agosto de 2025.

[23] Reducción de dimensionalidad, . URL https://es.wikipedia.org/wiki/Reducci%C3%B3n_de_dimensionalidad#Ventajas_de_la_reducci%C3%B3n_de_dimensionalidad. Última consulta: 4 de Agosto de 2025.

[24] Análisis de componentes principales. URL https://es.wikipedia.org/wiki/An%C3%A1lisis_de_componentes_principales. Última consulta: 4 de Agosto de 2025.

[25] Análisis factorial. URL https://es.wikipedia.org/wiki/An%C3%A1lisis_factorial. Última consulta: 4 de Agosto de 2025.

[26] Descomposición en valores singulares. URL https://es.wikipedia.org/wiki/Descomposici%C3%B3n_en_valores_singulares. Última consulta: 10 de Agosto de 2025.

[27] Matriz de confusión. URL https://es.wikipedia.org/wiki/Matriz_de_confusi%C3%B3n. Última consulta: 10 de Agosto de 2025.

[28] Joaquim Moré. Evaluación de la calidad de los sistemas de reconocimiento de sentimientos. URL https://openaccess.uoc.edu/server/api/core/bitstreams/6ff15a78-47c1-45ba-9475-442a6e8d19cc/content. Última consulta: 6 de Mayo de 2025.

[29] Validación cruzada, . URL https://es.wikipedia.org/wiki/Validaci%C3%B3n_cruzada. Última consulta: 10 de Agosto de 2025.

[30] Validación cruzada de k iteraciones, . URL https://es.wikipedia.org/wiki/Validaci%C3%B3n_cruzada#Validaci%C3%B3n_cruzada_de_K_iteraciones. Última consulta: 10 de Agosto de 2025.

[31] Validación cruzada aleatoria, . URL https://es.wikipedia.org/wiki/Validaci%C3%B3n_cruzada#Validaci%C3%B3n_cruzada_aleatoria. Última consulta: 10 de Agosto de 2025.

[32] Validación cruzada dejando uno fuera, . URL https://es.wikipedia.org/wiki/Validaci%C3%B3n_cruzada#Validaci%C3%B3n_cruzada_dejando_uno_fuera. Última consulta: 10 de Agosto de 2025.

[33] ¿qué es un árbol de decisión?, . URL https://www.ibm.com/es-es/think/topics/decision-trees. Última consulta: 27 de Agosto de 2025.

[34] Decision trees, . URL https://scikit-learn.org/stable/modules/tree.html. Última consulta: 27 de Agosto de 2025.

[35] Ensembles: Gradient boosting, random forests, bagging, voting, stacking, . URL https://scikit-learn.org/stable/modules/ensemble.html. Última consulta: 27 de Agosto de 2025.

[36] Random forest, . URL https://es.wikipedia.org/wiki/Random_forest#Caracter%C3%ADsticas_(o_rasgos)_y_Ventajas. Última consulta: 27 de Agosto de 2025.

[37] Random forest, . URL https://es.wikipedia.org/wiki/Random_forestt#Desventajas. Última consulta: 1 de Septiembre de 2025.

[38] k vecinos más próximos, . URL https://es.wikipedia.org/wiki/K_vecinos_m%C3%A1s_pr%C3%B3ximos. Última consulta: 1 de Septiembre de 2025.

[39] Nearest neighbors, . URL https://scikit-learn.org/stable/modules/neighbors.html. Última consulta: 1 de Septiembre de 2025.

[40] Ridge classification. URL https://scikit-learn.org/stable/modules/linear_model.html#classification. Última consulta: 3 de Septiembre de 2025.

[41] Perceptrón. URL https://es.wikipedia.org/wiki/Perceptr%C3%B3n. Última consulta: 6 de Septiembre de 2025.

[42] Hani AlOmari, Qussai Yaseen, and Mohammed Al-Betar. A comparative analysis of machine learning algorithms for android malware detection. *Procedia Computer Science*, 220:763–768, 01 2023. doi: 10.1016/j.procs.2023.03.101.

[43] Seguridad informática. URL https://es.wikipedia.org/wiki/Seguridad_inform%C3%A1tica#Objetivos. Última consulta: 6 de Junio de 2025.

[44] Los componentes de la tríada de la cia. URL https://www.checkpoint.com/es/cyber-hub/cyber-security/what-is-it-security/what-is-the-cia-triad/. Última consulta: 11 de Agosto de 2025.

[45] Phishing. URL https://es.wikipedia.org/wiki/Phishing. Última consulta: 11 de Agosto de 2025.

[46] Ataque de denegación de servicio. URL https://es.wikipedia.org/wiki/Ataque_de_denegaci%C3%B3n_de_servicio. Última consulta: 11 de Agosto de 2025.

[47] Joseph Nusbaum Peter Mell, Karen Kent. Guide to malware incident prevention and handling. URL https://profsite.um.ac.ir/kashmiri/nist/SP800-83.pdf. Última consulta: 13 de Mayo de 2025.

[48] Jorge Pablo Trías Posa. Aprendizaje automático aplicado a la detección de malware y de ciberataques. URL http://hdl.handle.net/10609/150576. Última consulta: 13 de Mayo de 2025.

[49] Antivirus. URL https://es.wikipedia.org/wiki/Antivirus. Última consulta: 13 de Mayo de 2025.

[50] Inteligencia artificial para la detección de binarios maliciosos. URL https://openaccess.uoc.edu/bitstream/10609/138409/6/jdiaznavTFM0122memoria.pdf. Última consulta: 26 de Mayo de 2025.

[51] ¿qué es la detección de intrusiones basada en firmas?, . URL https://www.purestorage.com/es/knowledge/signature-based-intrusion-detection.html. Última consulta: 13 de Agosto de 2025.

[52] Eset. URL https://www.eset.com. Última consulta: 11 de Agosto de 2025.

[53] Bitdefender. URL https://www.bitdefender.com. Última consulta: 13 de Agosto de 2025.

[54] Virustotal, . URL https://docs.virustotal.com/docs/how-it-works. Última consulta: 13 de Agosto de 2025.

[55] ¿qué es una vulnerabilidad zero day? URL https://www.incibe.es/ciudadania/blog/que-es-una-vulnerabilidad-zero-day. Última consulta: 15 de Agosto de 2025.

[56] Heurística en antivirus. URL https://es.wikipedia.org/wiki/Heur%C3%ADstica_en_antivirus. Última consulta: 15 de Agosto de 2025.

[57] ¿en qué consiste el análisis heurístico? URL https://www.kaspersky.es/resource-center/definitions/heuristic-analysis. Última consulta: 15 de Agosto de 2025.

[58] ¿qué es el análisis heurístico? URL https://www.fortinet.com/lat/resources/cyberglossary/heuristic-analysis. Última consulta: 15 de Agosto de 2025.

[59] La ofuscación de código: un arte que reina en la ciberseguridad. URL https://www.welivesecurity.com/es/recursos-herramientas/ofuscacion-de-codigo-arte-ciberseguridad/. Última consulta: 15 de Agosto de 2025.

[60] Detección de malware en 2025: técnicas, herramientas y desafíos reales. URL https://s2grupo.es/deteccion-de-malware-en-2025/. Última consulta: 15 de Agosto de 2025.

[61] Jorge Pablo Trías Posa. Aprendizaje automático aplicado a la detección de malware y de ciberataques, 2024. URL https://openaccess.uoc.edu/server/api/core/bitstreams/c07e4b84-89d7-4489-8d58-dc676f411b45/content.

[62] Machine learning en detección de malware. URL https://www.campusciberseguridad.com/blog/machine-learning-en-deteccion-de-malware/. Última consulta: 26 de Mayo de 2025.

[63] Reema Patel Akshit J. Dhruv and Nishant Doshi. Python: The most advanced programming language for computer science applications. URL https://www.scitepress.org/Papers/2020/103079/103079.pdf. Última consulta: 6 de Mayo de 2025.

[64] J. Hao and T. K. Ho. Machine learning made easy: A review of scikit-learn package in python programming language. *Journal of Educational and Behavioral Statistics*, 44(3):348–361, 2019. doi: 10.3102/1076998619832248.

[65] Francisco Bérchez-Moreno, Rafael Ayllón-Gavilán, Víctor M. Vargas, David Guijo-Rubio, César Hervás-Martínez, Juan C. Fernández, and Pedro A. Gutiérrez. dlordinal: A python package for deep ordinal classification. *Neurocomputing*, 622:129305, 2025. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2024.129305. URL `https://www.sciencedirect.com/science/article/pii/S0925231224020769`.

[66] Matplotlib. URL `https://en.wikipedia.org/wiki/Matplotlib`. Última consulta: 9 de Mayo de 2025.

[67] Numpy. URL `https://es.wikipedia.org/wiki/NumPy`. Última consulta: 9 de Mayo de 2025.

[68] Pandas. URL `https://es.wikipedia.org/wiki/Pandas_(software)`. Última consulta: 9 de Mayo de 2025.

[69] Lightgbm. URL `https://en.wikipedia.org/wiki/LightGBM`. Última consulta: 6 de Junio de 2025.

[70] MJ Bahmani. Understanding lightgbm parameters (and how to tune them). URL `https://dev.to/kamil_k7k/understanding-lightgbm-parameters-and-how-to-tune-them-14n0`. Última consulta: 6 de Junio de 2025.

[71] Seaborn. URL `https://seaborn.pydata.org/`. Última consulta: 6 de Junio de 2025.

[72] Bodmas. URL `https://whyisyoung.github.io/BODMAS/`. Última consulta: 2 de Abril de 2025.

[73] Virusshare, . URL `https://virusshare.com/`. Última consulta: 12 de Marzo de 2025.

[74] thezoo. URL https://github.com/ytisf/theZoo. Última consulta: 18 de Marzo de 2025.

[75] Microsoft malware classification challenge, . URL https://www.kaggle.com/c/malware-classification/data. Última consulta: 25 de Marzo de 2025.

[76] A low complexity ml-based methods for malware classification, . URL https://www.researchgate.net/publication/383827671_A_Low_Complexity_ML-Based_Methods_for_Malware_Classification. Última consulta: 17 de Mayo de 2025.

[77] Randomundersampler, . URL https://imbalanced-learn.org/stable/references/generated/imblearn.under_sampling.RandomUnderSampler.html. Última consulta: 25 de Mayo de 2025.

[78] sklearn.decomposition.pca. URL https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html. Última consulta: 2 de junio de 2025.

[79] Mohamad Zolkipli and Aman Jantan. Malware behavior analysis: Learning and understanding current malware threats. URL https://www.researchgate.net/publication/232657598_Malware_Behavior_Analysis_Learning_and_Understanding_Current_Malware_Threats. Última consulta: 6 de Mayo de 2025.

Anexo A

Código del programa

A.1. Codificación de las categorías *malware*

```
X, y        = load('bodmas/bodmas.npz')
metadata    = pd.read_csv('bodmas/bodmas_metadata.csv')
mw_category = pd.read_csv('bodmas/bodmas_malware_category.csv')

# Incluimos los valores de 'category' en metadata cuando coinciden
   los valoes de 'sha'
mw_category = metadata.merge(mw_category, on = 'sha', how = 'left')

# Rellenamos los huecos como software benigno
mw_category['category'] = mw_category['category'].fillna('benign')

# Eliminamos todas las columnas excepto 'category'
mw_category = mw_category['category']

# Codificamos las categorias de malware
category = {
  'benign': 0, 'trojan': 1, 'worm': 2, 'backdoor': 3,
  'downloader': 4, 'informationstealer': 5, 'dropper': 6,
  'ransomware': 7, 'rootkit': 8, 'cryptominer': 9, 'pua': 10,
  'exploit': 11, 'virus': 12, 'p2p-worm': 13, 'trojan-gamethief':
   14
}
```

```
mw_category = mw_category.map(category)

y = mw_category.to_numpy()

save('bodmas/bodmas_multiclass.npz', X, y)
```

A.2. Reducción de la dimensionalidad

```
def resampling(X, y, n_components = 5, size = 15000, u = False):
  if u:
    rus  = RandomUnderSampler(sampling_strategy = {0: size, 1: size
   })
    # rus  = RandomUnderSampler(sampling_strategy = 'majority')
    X, y = rus.fit_resample(X, y)

  X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size = 0.25, random_state = 1
  )

  pca     = PCA(n_components)
  X_train = pca.fit_transform(X_train)
  X_test  = pca.transform(X_test)

  return X_train, X_test, y_train, y_test
```

A.3. Pruebas para la elección del conjunto de datos

```
file = {'pca_binary', 'resampling_binary', 'pca_multiclass'}
clf  = None

print('clasificador,dataset,n patrones,n caracteristicas,accuracy,
    tiempo')

for i in range(3):
  if i == 0: clf = DecisionTreeClassifier()
  elif i == 1: clf = RandomForestClassifier()
  else: clf = KNeighborsClassifier()

  for train_file in file:

    X_train, y_train = load('bodmas/' + train_file + '_train.npz')
    X_test, y_test = load('bodmas/' + train_file + '_test.npz')

    # Entrenar el modelo
    inicio = time.time()
    clf.fit(X_train, y_train)
    tiempo = time.time() - inicio

    # Predecir sobre el conjunto de prueba
    y_pred = clf.predict(X_test)

    # Evaluar
    accuracy = accuracy_score(y_test, y_pred)

    print(f'{i},{train_file},{X_train.shape},{accuracy:.3f},{tiempo
    :.3f}')
```

A.4. Control de la validación cruzada

```
def cv(y, crossval):
  y_ = min(pd.DataFrame(y).value_counts())

  if y_ < crossval:
    return y_

  return crossval
```

A.5. Ejemplo de salida de la información

	acc train	ms train	f1 train	acc test	ms test	f1 test
0	0.648800	0.548712	1.0	0.648133	0.530019	1.0
1	0.645200	0.558267	1.0	0.655200	0.568564	1.0
2	0.652400	0.572655	1.0	0.644667	0.563784	1.0
3	0.648578	0.566829	1.0	0.653467	0.569664	1.0
4	0.650933	0.573087	1.0	0.650933	0.573003	1.0
5	0.647289	0.562228	1.0	0.647867	0.558393	1.0
6	0.647867	0.556000	1.0	0.650267	0.572533	1.0
7	0.650667	0.570983	1.0	0.649600	0.572906	1.0
8	0.650711	0.564155	1.0	0.638800	0.551123	1.0
9	0.649911	0.563205	1.0	0.645200	0.550628	1.0
Mean	0.649236	0.563612	1.0	0.648413	0.561062	1.0
STD	0.002112	0.007797	0.0	0.004713	0.013867	0.0