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

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

agosto, 2025

UNIVERSIDAD
D
CÓRDOBA

Resumen

rellenar
Palabras clave: Palabra-1, Palabra-2, Palabra-3, Palabra-4.

Abstract

rellenar
Keywords: Palabra-1, Palabra-2, Palabra-3, Palabra-4.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Capítulo 2

Estado de la técnica

2.1. Aprendizaje automático

2.1.1. Balanceo de datos

2.1.1.1. Sobremuestreo

2.1.1.2. Submuestreo

2.1.2. Reducción de la dimensionalidad

2.1.2.1. Análisis de componentes principales

2.1.2.2. Análisis factorial

2.1.2.3. Descomposición en valores singulares

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. Exactitud

La exactitud o *Accuracy* 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) [1].

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

2.1.3.2. 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.

2.1.3.3. 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) [1]. Un valor alto de sensibilidad indica que se han obtenido pocos falsos negativos.

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

2.1.3.4. 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

2.1.3.5. Valor-F

El valor-F o *F1-score* mide el equilibrio entre la precisión y la sensibilidad [1]. 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.

2.1.3.6. Matriz de confusión

2.1.4. Técnicas de validación

2.1.4.1. Validación cruzada

2.1.4.2. Validación estratificada

2.1.4.3. Problemas de entrenamiento

2.1.5. Preprocesamiento de datos

2.1.5.1.

2.1.6. Algoritmos de clasificación

2.1.6.1.

2.1.7.

2.1.7.1.

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 [2]. 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

2.2.2. *Malware*

2.2.2.1. Tipos de *Malware*

El *software* malicioso o *malware* es cualquier tipo de *software* que se introduce de manera encubierta con el objetivo de comprometer la confidencialidad, integridad o disponibilidad de la información o el sistema [3]. El *malware* se ha convertido en una de las amenazas externas más relevantes debido al daño que puede llegar a causar en una organización. Podemos clasificar el *malware* en diferentes categorías [4] 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 [5].

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 [6]. A continuación se estudiarán algunas de las técnicas más usadas.

2.2.3.1. Detección basada en firmas

2.2.3.2. Detección heurística y análisis estático

2.2.3.3. Detección basada en comportamiento (análisis dinámico)

2.2.3.4. Métodos híbridos

2.2.3.5. Detección mediante aprendizaje automático

2.2.4. Retos y tendencias

Capítulo 3

Formunación del problema

Capítulo 4

Objetivos

Debe de ser igual a los mencionados en el anteproyecto.

Describir el objetivo principal y los objetivos especificos llevados a cabo para conseguir el objetivo principal.

Capítulo 5

Metodología de trabajo

5.1. Enfoque metodológico

5.2. Procedimiento seguido

5.3. Técnicas y herramientas empleadas

5.4. Criterios de selección de datos y modelos.

5.4.1. Selección del conjunto de datos

En lo que a *malware* se refiere, *BODMAS* [7] 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* [8], 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* [9]. 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* [10]. En este caso tenemos un conjunto de datos muy amplio con casi medio *terabyte*, pero además de los inconvenientes 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

5.4.2. Selección de los modelos

Capítulo 6

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 multinivel 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.

6.1. Modelos utilizados

En este proyecto se han empleado diversos algoritmos de aprendizaje automático, seleccionados en función de los criterios mencionados en el capitulo 5 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.

6.2. Procesamiento del conjunto de datos

Dadas las limitaciones *hardware* y la cantidad de datos, aproximadamente 135000 patrones y 2400 atributos por cada patrón, 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.

6.2.1. 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 6.1.

Tabla 6.1: Codificación de las clases *malware*.

Categoría	Codificación	Nº de patrones
benign	0	77142
trojan	1	29972
worm	2	16697
backdoor	3	7331
downloader	4	1031
informationstealer	5	448
dropper	6	715
ransomware	7	821
rootkit	8	3
cryptominer	9	20
pua	10	29
exploit	11	12
virus	12	192
p2p-worm	13	16
trojan-gamethief	14	6

Para obtener una nueva matriz de salidas deseadas que incluya los tipos de *malware*, una vez cargados los datos en sus correspondiente variables de *python*, usamos la función *merge* [11] perteneciente a la clase *pandas.DataFrame* para incluir en *metadata* los datos de *mw_category['category']* en las entradas donde coincide la columna *sha*.

Antes de codificar necesitamos darle una etiqueta a los datos vacíos, los cuales significan que esa muestra es benigna. Para ello usamos la función *pandas.DataFrame.fillna* [12], que nos permite completar datos vacíos de distintas formas. Para nuestro caso usamos la etiqueta *benign*. También eliminamos las columnas que no vamos a necesitar, dejando solo la categoría a la que pertenece cada muestra.

Ahora podemos codificar los datos usando la función *pandas.DataFrame.map* [13]. Este método aplica una función que acepta y devuelve un valor escalar a cada elemento del DataFrame, lo que permite asignar un valor numérico a cada clase.

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

6.2.2. Reducción del conjunto de datos

Reducir el número de datos con el que vamos a trabajar tiene el objetivo de principal 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* [14] 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.

6.2.2.1. Número de patrones

Como ya hemos estudiado en la sección 2.1.1.2, 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. En nuestro caso, el desbalance no es demasiado grande ya que *BODMAS* contiene 57293 muestras *malware* y 77142 muestras benignas.

El método *RandomUnderSampler* [15] 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.

6.2.2.2. Número de características

Este método, también conocido como reducción de la dimensionalidad, 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* [16] perteneciente a *sklearn.decomposition*. 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.

6.2.2.3. 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*.

Los resultados obtenidos se reflejan en las tablas 6.2, 6.3 y 6.4 respectivamente.

Tabla 6.2: Clasificación binaria con *PCA*.

Clasificador	Tiempo (s)	Entrenamiento			Test		
		Acc	MS	F1	Acc	MS	F1
Decission 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 6.3: Clasificación binaria con *PCA* y *Undersampling*.

Clasificador	Tiempo (s)	Entrenamiento			Test		
		Acc	MS	F1	Acc	MS	F1
Decission 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

Tabla 6.4: Clasificación multiclase con *PCA*

Clasificador	Tiempo (s)	Entrenamiento			Test		
		Acc	MS	F1	Acc	MS	F1
Decission 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

En cuanto a la clasificación binaria, 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.

Para la clasificación multiclase 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.

También en referencia a la clasificación multiclase, podemos ver en la tabla 6.4 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 6.1, 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.

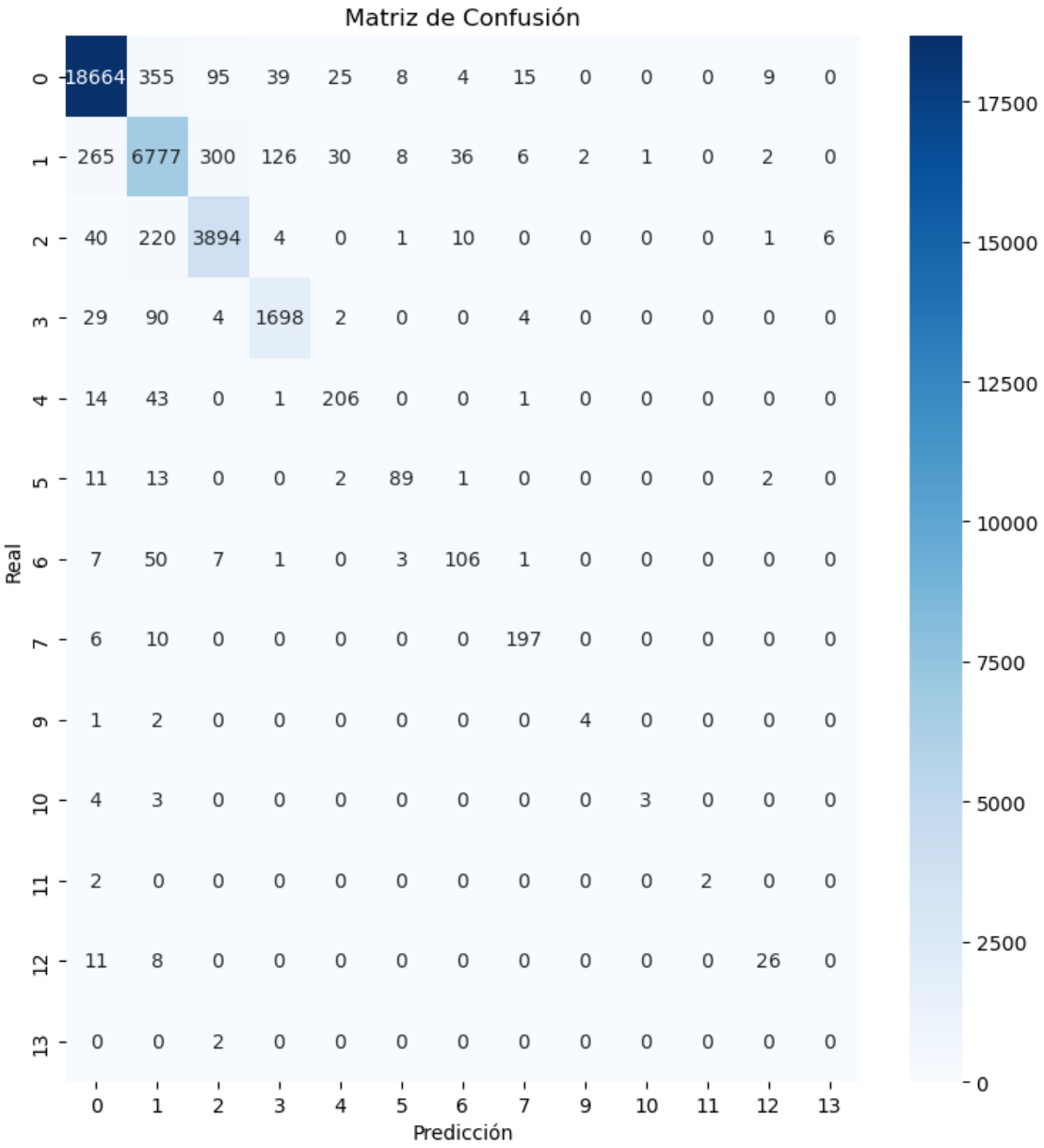

Figura 6.1: Matriz de confusión para la clasificación multicase

Según el estudio *Malware Behavior Analysis: Learning and Understanding Current Malware Threats* [17], 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 6.6 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 6.5

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 2.1.1.1, 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.

Tabla 6.5: Nueva codificación de las clases *malware*.

Categoría	Codificación	Nº de patrones
benign	0	77142
trojan	1	29978
worm	2	16713
backdoor	3	7331
downloader	4	1031
informationstealer	5	448
dropper	6	715
ransomware	7	821
virus	8	192
otros	9	64

Tabla 6.6: 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

6.3. 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.

6.3.1. 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 [18].

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.

Las bibliotecas utilizadas para construir un modelo y analizar los datos son *Scikit-learn*, *DLOrdinal*, *Matplotlib*, *NumPy*, *Pandas*, *LightGBM* y *Seaborn*.

6.3.1.1. *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 [19].

6.3.1.2. *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 [20]. 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 algunas de las métricas que finalmente hemos usado: mínima sensibilidad y valor-F.

6.3.1.3. *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 [21]. Se ha usado, en combinación con *Seaborn*, descrita en la sección 6.3.1.7.

6.3.1.4. *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 [22]. 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.

6.3.1.5. *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 [23]. Varias de ellas se han utilizado durante el desarrollo y la preparación del conjunto de datos.

6.3.1.6. *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 [24]. 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)* [25].

6.3.1.7. *Seaborn*

Basada en *Matplotlib*, *Seaborn* proporciona una interfaz de alto nivel para generar gráficos estadisticos [26]. 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.

6.3.2. 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 com-

puesto 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*.

6.3.3. Protocolo de experimentación y validación

En esta sección se establecen las condiciones de evaluación del rendimiento de los modelos mencionados en la sección 6.1 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.

6.3.3.1. 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.

6.3.3.2. 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 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.

6.3.3.3. 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.

6.3.3.4. 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 aun 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.

6.3.3.5. Criterios de evaluación

Para evaluar la efectividad de los modelos implementados, se han utilizado las siguientes métricas, cuya descripción teórica se encuentra en la sección 2.1.3:

- ***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.

6.4. 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 6.3.3.5

6.4.1. Estructura del código

6.4.2. 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 6.3.3.5, se calcula la media y la desviación típica de todas ellas para usarlas como valor final de comparación entre modelos.

6.4.3. Preparación y uso de los conjuntos de datos

Además del tratamiento previo del conjunto de datos realizado en la sección 6.2, 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.

6.4.4. Métricas y análisis de resultados

Para calcular las métricas se han usado las funciones *accuracy_off1* y *minimum_sensitivity* para las métricas valor-F y mínima sensibilidad respectivamente. Estas se encuentran disponibles 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*. 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 7

Resultados y discusión

7.1. Clasificación binaria

En esta fase se lleva a cabo la implementación práctica del estudio, haciendo uso de los modelos de aprendizaje automático implementados principalmente en la librería *Scikit-Learn* de *python* y descritos en el capítulo 5. Para ello 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 de este proyecto se estudiará desde dos enfoques distintos. Por un lado se evaluarán los modelos seleccionados en la detección de *malware*, es decir, se realizarán pruebas de clasificación binaria donde se estudiará sin un un patrón corresponde a un programa malicioso o no. Por otro, se estudiará si, para estos mismos patrones, es posible realizar una clasificación más exhaustiva y reconocer con que tipo de *malware* se corresponde cada patrón.

Tabla 7.1: Clasificación binaria con *DecisionTreeClassifier*

	Entrenamiento			Test		
Estado aleatorio	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

Tabla 7.2: Clasificación binaria con *RandomForestClassifier*

	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

7.1.1. Árboles de decisión

7.1.2. *Random forest*

7.1.3. *K-NN*

7.1.4. Máquinas de vectores de soporte

7.1.5. *Ridge*

7.1.6. Redes neuronales: Perceptrón multicapa

7.1.7. *Light Gradient Boosting Machine*

7.2. Clasificación multiclase

7.2.1. Árboles de decisión

7.2.2. *Random forest*

7.2.3. *K-NN*

7.2.4. 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,

Tabla 7.3: Clasificación binara con *KNeighborscClassifier*

	Entrenamiento			Test		
Estado aleatorio	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

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 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.

7.2.5. *Ridge*

7.2.6. Redes neuronales: Perceptrón multicapa

7.2.7. *Light Gradient Boosting Machine*

Tabla 7.4: Clasificación binaria con *SVC*

	Entrenamiento			Test		
Estado aleatorio	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

Tabla 7.5: Clasificación binaria con *RidgeClassifier*

	Entrenamiento			Test		
Estado aleatorio	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

Tabla 7.6: Clasificación binaria con *MLPClassifier*

	Entrenamiento			Test		
Estado aleatorio	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

Tabla 7.7: Clasificación binaria con *LGBMClassifier*

	Entrenamiento			Test		
Estado aleatorio	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

Tabla 7.8: 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

Tabla 7.9: 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 7.10: Clasificación multiclase con *KNeighborsClasiffier*

	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 7.11: 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 7.12: 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 7.13: 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 8

Conclusiones y recomendaciones

Bibliografía

[1] 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.

[2] Seguridad informática. URL https://es.wikipedia.org/wiki/Seguridad_inform%C3%A1tica#Objetivos.

[3] Joseph Nusbaum Peter Mell, Karen Kent. Guide to malware incident prevention and handling. URL https://profsite.um.ac.ir/kashmiri/nist/SP800-83.pdf.

[4] Jorge Pablo Trías Posa. Aprendizaje automático aplicado a la detección de malware y de ciberataques. URL http://hdl.handle.net/10609/150576.

[5] Antivirus. URL https://es.wikipedia.org/wiki/Antivirus.

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

[7] Bodmas. URL https://whyisyoung.github.io/BODMAS/.

[8] Virusshare. URL https://virusshare.com/.

[9] thezoo. URL https://github.com/ytisf/theZoo.

[10] Microsoft malware classification challenge. URL https://www.kaggle.com/c/malware-classification/data.

[11] pandas.dataframe.merge. URL https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.merge.html.

[12] pandas.dataframe.fillna. URL https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.fillna.html.

[13] pandas.dataframe.map. URL https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.map.html.

[14] 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.

[15] Randomundersampler. URL https://imbalanced-learn.org/stable/references/generated/imblearn.under_sampling.RandomUnderSampler.html.

[16] sklearn.decomposition.pca. URL https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] Matplotlib. URL https://en.wikipedia.org/wiki/Matplotlib.

[22] Numpy. URL https://es.wikipedia.org/wiki/NumPy.

[23] Pandas. URL https://es.wikipedia.org/wiki/Pandas_(software).

[24] Lightgbm. URL https://en.wikipedia.org/wiki/LightGBM.

[25] 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.

[26] Seaborn. URL https://seaborn.pydata.org/.

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