UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

TÍTULO: Detección y Clasificación Zero-Day Malware a través de Data Mining y Machine Learning
AUTORES: Augusto Recordon – Silvia Ruiz Diaz
DIRECTOR: Dra. Claudia Pons
CODIRECTOR:
ASESOR PROFESIONAL:
CARRERA: Licenciatura en Sistemas

Resumen

Dado el constante incremento, tanto en número como en complejidad, de los ataques informáticos, los mecanismos convencionales de detección resultan ineficientes en la mayoría de los escenarios. En este contexto, la presente investigación propone determinar si técnicas de data mining y machine learning pueden ser utilizadas efectivamente para el entrenamiento de algoritmos capaces de detectar y clasificar correctamente nuevos tipos de amenazas.

Palabras Clave

Machine learning, malware, seguridad informática, virus, zero-day, data mining, inteligencia artificial, redes neuronales.

Conclusiones

Teniendo en cuenta los resultados obtenidos, es posible afirmar que la utilización de técnicas de data mining y machine learning para clasificar familias de malware ha resultado muy efectiva. En cuanto a la problemática de la detección, se cree que, de contar con un conjunto de datos considerablemente más grande, los algoritmos podrían lograr resultados ampliamente superiores.

Trabajos Realizados

Para las tareas de clasificación, a partir de un conjunto de once mil muestras de malwares, se construyó un dataset conformado por los atributos considerados más relevantes para cada familia. Con estos datos, se implementaron y ejecutaron distintos algoritmos de machine learning de clasificación.
En cuanto a la detección de malware, se aplicaron las mismas técnicas que para las tareas de clasificación, utilizando como datos de partida programas considerados benignos, los cuales tuvieron que ser desensamblados.

Trabajos Futuros

En cuanto a la tarea de clasificación de malware, se puede extender el trabajo realizado comprendiendo un mayor número de familias y una mayor cantidad de muestras para aquellas familias con menos observaciones.
Para la detección de malware, se recomienda volver a ejecutar los algoritmos aquí implementados con una cantidad significativamente mayor de archivos benignos en el conjunto de datos.

Fecha de la presentación: Octubre de 2020

FACULTAD DE INFORMÁTICA
UNIVERSIDAD NACIONAL DE LA PLATA

DETECCIÓN Y CLASIFICACIÓN DE ZERO-DAY MALWARE A TRAVÉS DE DATA MINING Y MACHINE LEARNING

TESINA DE GRADO

Autores:
Augusto RECORDON
Silvia RUIZ DIAZ

Directora:
Dra. Claudia PONS

17 de octubre de 2020

«*Success is not final, failure is not fatal: it is the courage to continue that counts.*»

Sir Winston Churchill

UNIVERSIDAD NACIONAL DE LA PLATA

Facultad de Informática

Resumen

Tesis de Licenciatura en Sistemas

DETECCIÓN Y CLASIFICACIÓN DE ZERO-DAY MALWARE A TRAVÉS DE DATA MINING Y MACHINE LEARNING

por Augusto RECORDON
Silvia RUIZ DIAZ

Muchos estudios sugieren que, durante los últimos años, ha habido un incremento exponencial de los ataques informáticos, causando a las organizaciones pérdidas financieras en el orden de los millones. Mientras muchas compañías dedican tiempo y recursos al desarrollo de antivirus; la complejidad, la velocidad de propagación y la capacidad polimórfica que poseen los virus modernos representan enormes desafíos para estas empresas. Motivados por encontrar nuevas alternativas, la comunidad de científicos de datos ha descubierto que la utilización de técnicas de *machine learning* y *deep learning* para la detección y clasificación de *malware* puede ofrecer una opción más que competitiva. Para esta investigación se comenzará realizando las extracción de información de un conjunto de datos compuesto por once mil archivos `ASM` y `bytes` correspondientes a nueve familias distintas de *malwares*. Luego, mediante la implementación de algoritmos de *machine learning* se intentará clasificar estos *malwares* en sus correspondientes familias. De forma complementaria, se realizará una clasificación binaria para detección *malware/no malware*, con un conjunto reducido de programas benignos, finalizando así con la elaboración de comparaciones y conclusiones.

Índice general

Índice de figuras

CAPÍTULO 1

Introducción

Vivimos en una era en donde la tecnología cumple un rol muy importante en nuestras vidas. Las personas utilizan Internet para todo tipo de actividades, desde tareas sensibles como pagar sus cuentas, realizar compras y consultar estados bancarios, hasta para mantener contacto con sus amistades y familiares, ver películas y leer las noticias. Esto frecuentemente nos lleva a instalar distintas aplicaciones en nuestros dispositivos y a aceptar los términos y condiciones de sitios y compañías, sin saber realmente a qué datos tienen acceso, ni para qué éstos serán usados. Del mismo modo, inadvertidamente o no, diariamente transmitimos información sensible, como contraseñas y datos de tarjetas de crédito, por canales que confiamos que sean seguros y que nadie está escuchando. De forma similar, compartimos información personal, como ubicación y fotos, en redes sociales; raramente deteniéndonos a pensar que, del otro lado, puede haber alguien malintencionado creando un perfil nuestro y de nuestras actividades.

El número de aplicaciones y utilidades de Internet es, sin lugar a dudas muy extenso, pero la base es una misma: el *software*. *Software* en nuestros dispositivos, en servidores y en equipos intermedios que nos conectan. *Software* en el cual, como se ha mencionado, confiamos diariamente. Sin embargo éste, como toda construcción humana, es susceptible a errores, vulnerabilidades y escenarios no previstos. Más aún, los errores y vulnerabilidades, una vez detectados, llevan tiempo corregir y, en muchos casos, depende de las compañías y usuarios de estos sistemas aplicar la actualizaciones que solucionan los problemas.

En este contexto, los *hackers*, personas malintencionadas que se aprovechan de las vulnerabilidades del *software* y/o de la confianza de las personas, despliegan sus conocimientos y herramientas para llevar a cabo sus objetivos. Quizás la más común de estas herramientas es el **malware**. El término *malware*, de **mal**icious soft**ware**, es un programa cuyo fin es comprometer cualquier computadora o dispositivo inteligente, diseñado por un *hacker* con fines maliciosos, ya sea para robar información confidencial, penetrar redes, dañar infraestructuras críticas, etc. Estos programas pueden

incluir virus, *worms*, troyanos, *spyware*, *bots*, *rootkits*, *ransomware*, entre otros. Según el prestigioso instituto dedicado a la seguridad IT, AV-Test[1], cada día se producen más de 350.000 nuevos programas maliciosos (*malware*) y *potentially unwanted applications (PUA)*, con un incremento en los últimos 10 años cercano al 2000 %. Esto representa pérdidas millonarias para las compañías y los gobiernos. Para el 2021, el sitio *Cybersecurity Ventures*[2] estima que la pérdida anual por daños relacionados al cibercrimen alcanzaría los 6 mil millones de dólares a nivel global. Estos datos son realmente alarmantes y representarán un desafío que la humanidad deberá enfrentar por las próximas décadas.

Empresas dedicadas al desarrollo de anti-virus, tales como *Norton*, *AVG*, *McAfee*, *Kaspersky*, entre otros; hacen su mejor esfuerzo para hacer frente a esta problemática. Tradicionalmente, estos programas utilizaban el método *signature-based* para la detección de *malware*. La *signature* es una secuencia corta de bytes que sirve para identificar *malwares* conocidos, sin embargo este método no es capaz de proveer una forma de identificar ataques *zero-day*, o *malwares* polimórficos que utilizando técnicas de ofuscación son capaces de crear cientos de variedades de si mismo, dado que no cuentan con registros de los mismos.

En los últimos años, grupos de investigadores junto con la comunidad *anti-malware* han reportado estar utilizando técnicas de *machine learning* y *deep learning*, tanto para el análisis, como para la detección de *malware*. Existen dos técnicas que se pueden aplicar para decidir si el código de un programa es benigno o maligno, ya sea, realizando un análisis estático o uno dinámico. Las técnicas de análisis estático no ejecutan el código, sino que sólo examinan su estructura y algunas propiedades de los datos binarios. Las téncicas de análisis dinámico, en cambio, ejecutan el código para observar el comportamiento durante su ejecución sobre la red o en ambientes controlados, como por ejemplo un *sandbox*. Muchos sistemas de detección de *malwares* utilizan análisis estático o dinámico, e incluso ambos.

El propósito de esta investigación tiene como objetivo central evaluar la efectividad de utilizar distintas técnicas de análisis estático relacionadas con *machine learning* y *data mining* para la detección y clasificación de *malware*. Aplicar estas técnicas para la detección temprana de *malware* puede resultar de particular utilidad para identificar ataques *zero-day*. Un *zero-day* es un *malware* desconocido, es decir, un nuevo tipo de código malicioso, para el cual aún no se han creado parches o revisiones que resuelvan la falla de seguridad.

El proceso de clasificación y detección que será llevado a cabo en este trabajo constará de diversas etapas, comenzando con la obtención de los datos. Éstos serán los

[1] https://www.av-test.org/en/statistics/malware/
[2] https://cybersecurityventures.com/cybercrime-damages-6-trillion-by-2021/

que componen el conjunto de partida y estarán conformados por casi 11.000 muestras de archivos malignos desensamblados. Una vez hecho esto, tendrá lugar la construcción de un *dataset* a partir de la extracción de las características más relevantes, o consideradas de mayor interés, de todo el conjunto de datos. A esta etapa la denominamos *data mining* y la consideramos, quizás, como la más importantes y que más tiempo demandará de toda la investigación. El *dataset* generado deberá ser sometido a distintas técnicas de preprocesamiento para su depurado y preparación para que pueda ser utilizado por los algoritmos de *machine learning*. Finalmente se realizará la construcción y ajuste de los modelos de clasificación y de detección que sean considerados apropiados para dicha tarea. Los resultados obtenidos serán analizados y comparados empíricamente basándonos en distintas métricas.

Parte I

Marco Teórico

CAPÍTULO 2

Visión Histórica

El avance de la tecnología en los últimos tiempos han traído consigo grandes beneficios a la población y a los usuarios en general. Avances en el campo de la medicina, la seguridad, la intercomunicación entre personas que se encuentran físicamente en distintos lugares del planeta, e incluso han permitido la aparición de nuevos mercados con un gran potencial de crecimiento. Sin embargo, estos beneficios vienen acompañados de riesgos casi de manera inherente, por lo que además de oportunidades también conlleva amenzazas. Principalmente en la industria del *software*, en donde las amenazas por *malware* han tenido un incremento exponencial, haciéndolos a estos no sólo más sofisticados y complejos, sino también más difíciles de detectar.

En el siguiente capítulo se dará brevemente un marco histórico relacionado al riesgo de la tecnología y los ataques informáticos, mientras que más adelante se abordarán los avances que ha tenido el campo de la inteligencia artificial en los últimos años.

2.1. Los riesgos inherentes de la tecnología

Las nuevas tecnologías tienen como propósito otorgar diferentes beneficios a las personas, algo que han logrado exitosamente. Sin embargo, también existen consecuencias adversas derivadas de este progreso. Según el sitio *WeLiveSecurity*[1] de *Eset*[2], éstos son algunos riesgos asociados a la tecnología:

- **Inteligencia Artificial (AI)**: algo que ha sido denominado como el "mayor avance tecnológico"de los últimos años, ha comenzado a mostrar algunos indicios de peligro. Por ejemplo, los *fake news* (noticias falsas), y los *deep fakes* (imágenes donde la persona ha sido alterada digitalmente); así como otras características que aún no han vislumbrado por completo, como las interfaces computadora-cerebro y la híper-automatización, esto quiere decir, la combinación de la robótica con la inteligencia articial.

[1] https://www.welivesecurity.com/la-es/2020/02/13/ciberataques-principales-amenazas-2020/
[2] https://www.eset.com/

- **Tecnología móvil de quinta generación (5g)**: Estas nuevas tecnologías dependen de infraestructuras de alta velocidad, sin embargo, dadas las condiciones actuales se pronostican déficits significativos en la capacidad de cobertura y de inversiones en estas redes de telecomunicaciones. El desafío estará en construir infraestructura moderna, además de introducir sistemas que sean seguros y confiables dentro de las capacidades existentes.
- **Computación cuántica**: La computación cuántica podría reducir drásticamente el tiempo necesario para resolver problemas matemáticos en los que actualmente se apoyan las técnicas de cifrado. Esto es importante teniendo en cuenta que la capacidad de procesamiento podría volver imprácticos los algoritmos criptográficos de la actualidad, por lo que se correría el riego de inutilizar la mayoría de los sistemas actuales de infraestructura crítica y de seguridad de los datos.
- **Computación en la nube**: La computación en la nube tiene el potencial de desarrollar distintos sectores, expandir el acceso tecnológico a áreas remotas, así como vincularse con otras tecnologías al mismo tiempo, con una mayor cantidad de datos alojados en la nube, las empresas están acumulando cada vez más información personal, lo que crea potenciales riesgos a la privacidad y seguridad de los datos.

Además de los riesgos asociados a la tecnología, también se destacan aspectos relacionados a la ciberseguridad, en donde los ataques cibernéticos adoptan múltiples formas y se están extendiendo incluso al ambiente físico. En este sentido, los ciberataques a infraestructuras críticas comienzan a aparecer con normalidad en industrias como la energética, salud o transporte, afectando incluso a ciudades enteras.

Mientras la tecnología es un campo que se encuentran en permanente crecimiento, también lo es el cibercrimen, con ataques perpetrados por grupos cada vez más organizados, que cuentan con una gran disponibilidad de herramientas creadas para tales fines, y con una probabilidad muy baja de ser detectados y por lo tanto enjuiciados.

2.2. Los Ataques Informáticos y el Ciberdelito

La última década ha mostrado tener un importante incremento de los ataques informáticos, la razón para tal crecimiento se debe al cambio de motivación detrás de estos ataques. Hasta hace unos años, el principal objetivo era ver qué tan lejos se podía llegar. Sin embargo, ese ha dejado de ser el caso. En la actualidad los ataques esconden una gran gama de intereses, siendo el rédito económico el más preponderante. Incluso algunos estados han empezado a realizar ataques a otros con el fin de robar información y/o desestabilizarlos.

El siguiente gráfico permite observar claramente el aumento en el número de ataques que se registran cada año, correspondientes a la última década.

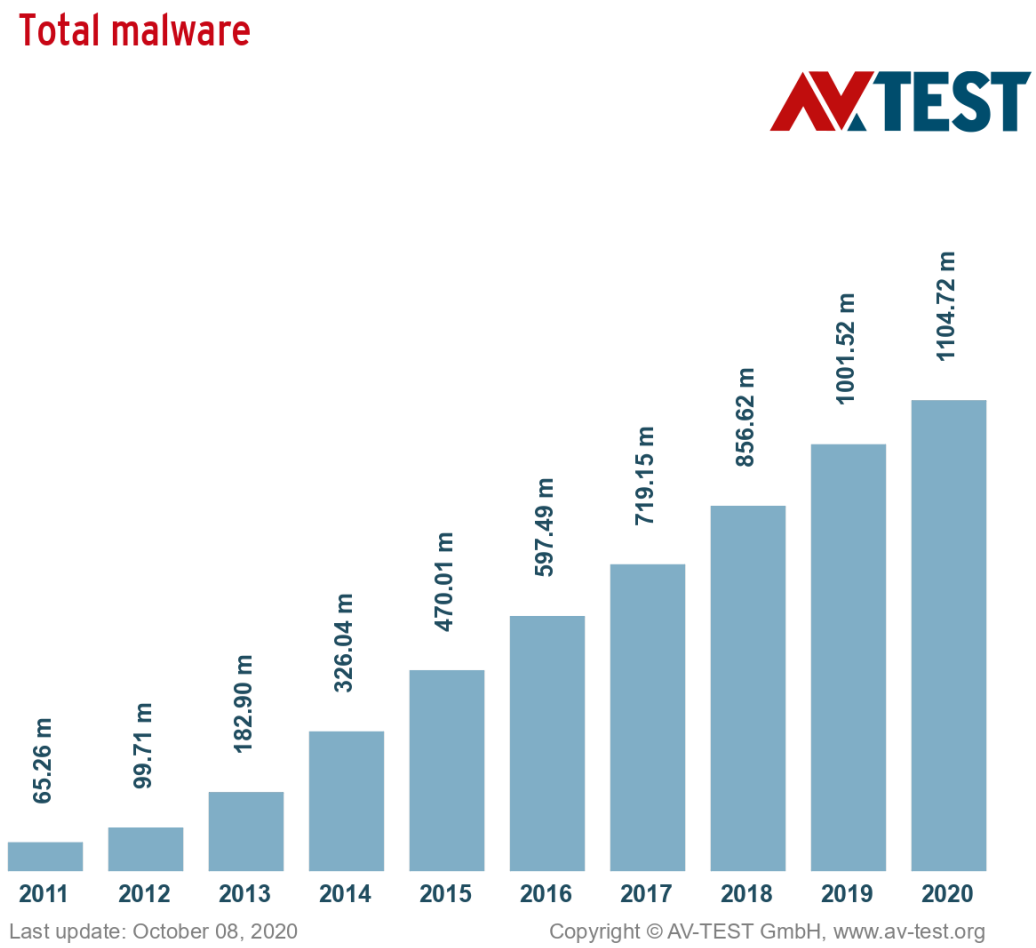

FIGURA 2.1: Estadística de total de ataques

AV-Test, uno de los institutos más reconocidos que se dedican al testeo de productos de seguridad IT, registra 350.000 infecciones por día, siendo el *malware* el tipo de ataque conocido más común. El gráfico que se encuentra a continuación muestra una estadística del último año correspondiente a los dos principales tipos de ataques.

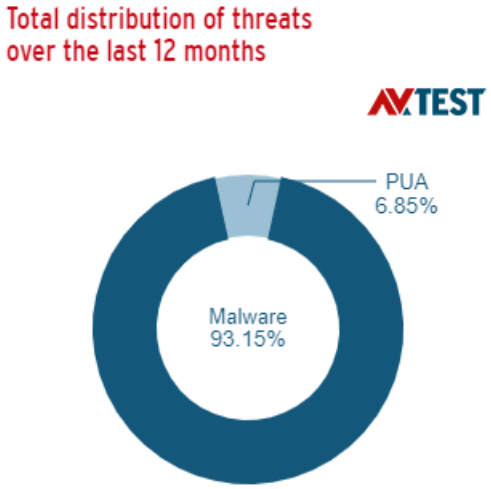

FIGURA 2.2: *Malware* vs *Potencially Unwanted Application* (PUA)

Estos ciberataques no sólo impactan a la industria del *software*, sino también a la economía, el estado de las empresas y a los gobiernos que deben responder ante dichos ataques.

2.3. Los avances en Inteligencia Artificial

El término Inteligencia Artificial fue acuñado en el 1956, y más de 60 años después, los avances en este campo han sido sorprendentes. Hasta hace unos pocos años, la Inteligencia Artificial parecía un asunto futurista que difícilmente podría alternarnos el día a día en el corto plazo. Sin embargo, hoy ya es una realidad que aspira a revolucionar varios aspectos de nuestra sociedad en los próximos años.

Los avances de la Inteligencia Artificial a la hora de superar la capacidad humana en actividades como el ajedrez, el juego *Go*, y la traducción llegan ahora a los titulares, pero la Inteligencia Artificial está presente en la industria desde, al menos, la década de 1980. La razón por la cual recién ahora empiezan a verse sus aplicaciones es porque para utilizarlos se requieren tres cosas: una alto poder de cálculo, una gran capacidad de almacenamiento y el software apropiado. Así mismo, estas tecnologías tenían que tener un costo aceptable.

2.3.1. Gran capacidad de almacenamiento

Uno de los pilares más importante de *Data Science* es la capacidad de almacenar grandes volúmenes de información, la cual tiene que ser guardada de manera confiable y transferida a grandes velocidades.

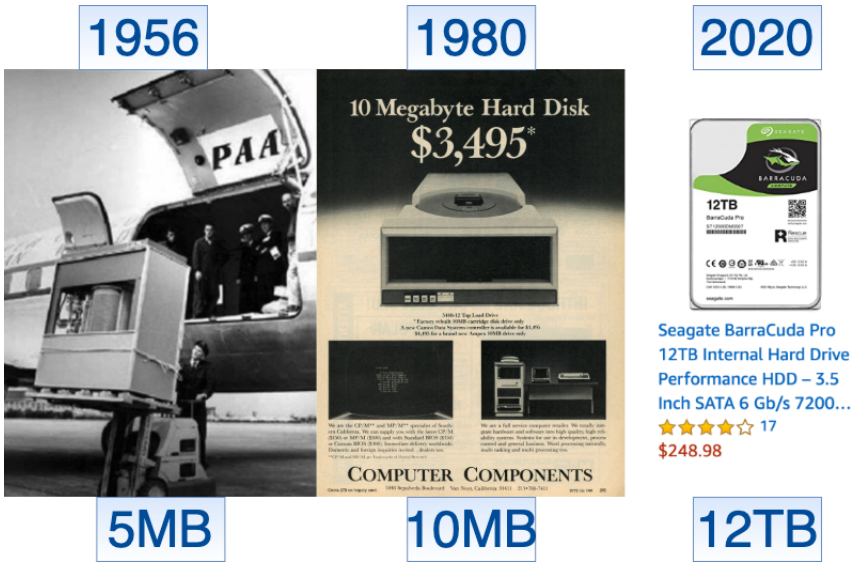

FIGURA 2.3: Almacenamiento - Visión histórica

En la imagen anterior se puede ver a la izquierda una unidad de almacenado de 5 mega bytes siendo cargada en un avión. En 1956 estas unidades eran alquiladas por 28000 dólares al mes (si se ajusta el precio por inflación). Casi 25 años después, 10 mega bytes (el doble de capacidad) se vendía por 3.495 dólares. En la actualidad, apenas 40 años después, por el 7 % de ese valor, se puede acceder a un disco rígido que provee 1.200.000 veces ese almacenado. Sin embargo, la ciencia está explorando en nuevas direcciones, prometiendo revolucionar el modo en el que almacenamos información: la utilización de ADN.

Estos dispositivos no sólo han minimizado drásticamente su tamaño sino también han aumentado su capacidad de almacenamiento exponencialmente. El siguiente cuadro presenta una comparativa entre discos rígidos, memorias flash y la utilización de ADN.

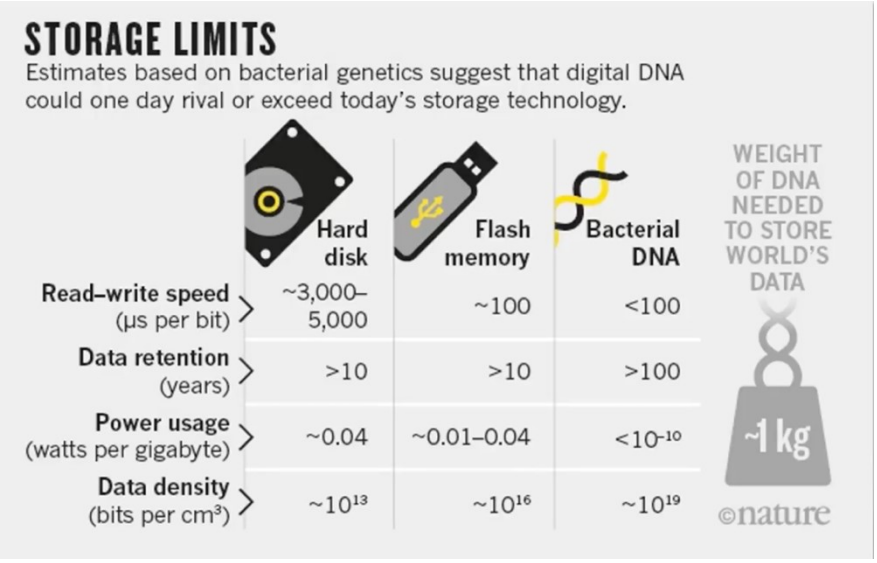

FIGURA 2.4: Medio de Almacenamiento - Comparativa

2.3.2. Alto poder de procesamiento

Para poder procesar los grandes volúmenes de datos de la actualidad se necesita un gran poder de cómputo para poder procesar. La Ley de Moore[3], que se sigue observando hasta la fecha, permite poner en perspectiva el aumento en el poder de cálculo, y permite ver cómo hasta hace unos pocos años las computadoras estaban muy lejos de lograr la capacidad computacional que poseen actualmente.

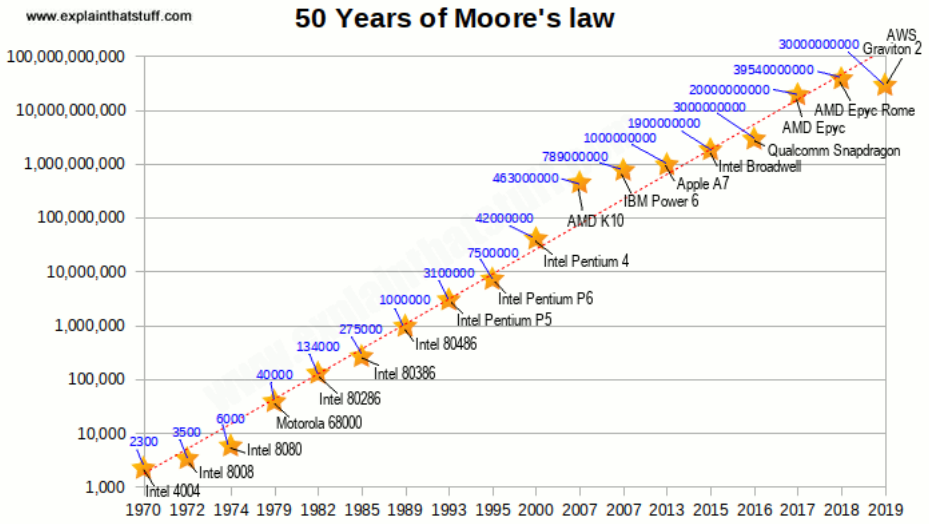

FIGURA 2.5: Ley de Moore

[3]La Ley de Moore establece, básicamente, que la capacidad de cómputo se duplica cada dieciocho meses.

2.3.3. Software requerido

El concepto de computadora inteligente surgió en 1950 con una prueba que Alan Turing planteó[4]. Desde entonces y durante los siguientes quince o veinte años tuvieron lugar algunos esfuerzos en el campo, tales como los primeros robots y computadoras como la *Mark I Perceptron* de 1960, capaz de aprender nuevas habilidades por ensayo y error. Sin embargo, no hubo ningún avance substancial hasta 1997, cuando *Deep Blue* derrotó al campeón ajedrecista *Garry Kasparov*. A partir de esos años, y hasta la actualidad, tanto *Machine Learning*, como Inteligencia Artificial y otras áreas relacionadas, han empezado a crecer cada vez más y adoptarse en todo tipo de campos.

Al reflexionar acerca de las causas detrás de la demora en la explosión de estas disciplinas se han mencionado las carencias en materia de procesamiento y almacenamiento. Pero también hay que destacar la falta de lenguajes de programación acordes, sistemas de bases de datos, procesamiento paralelo y distribuido y otras herramientas.

2.4. Resumen

En el presente capítulo se mencionó cómo los avances tecnológicos impactan la vida de las personas y los riesgos inherentes que éstos traen consigo. También se abordó el tema de los ataques informáticos y el cibercrimen en la actualidad, y cómo éstos se han ido incrementando y han evolucionado de manera alarmante.

Por último, se estudió el avance de la Inteligencia Artificial en los últimos años, y cuáles son los pilares que explicarían por qué su uso se ha venido popularizado los últimos años.

En el próximo capítulo se verán los conceptos relacionados a la seguridad informática, las distintas motivaciones de las personas que cometen los ataques informáticos y cuáles son los métodos que pueden ser utilizados para detectar estos ataques.

[4]El Test de Turing estipula que si una persona interactuando con una máquina no puede darse cuenta de que ésta no es una persona, entonces eso constituye evidencia para la máquina que sea considerada inteligente.

CAPÍTULO 3

Conceptos de Seguridad Informática

Cuando se habla de Seguridad Informática, se refiere a la práctica de preservar o defender todo aquel *software*, computadoras, servidores, dispositivos móviles, sistemas electrónicos, redes y bases de datos de ataques maliciosos. Para ello, es importante conocer no sólo los conceptos relacionados a dicho tema, sino también cuáles son los procesos, protocolos, métodos o herramientas que se pueden utilizar para hacer frente dicha amenaza.

En el siguiente capítulo, se describirán ciertos conceptos fundamentales relacionados a la Seguridad Informática. Se comenzará describiendo cómo pueden clasificarse los diferentes tipos de *hackers*; una descripción de las distintas clases de *malware* que pueden encontrarse, y finalmente, cuáles son los métodos para detectar estos programas.

3.1. Hackers

Se utiliza el término en inglés *"Hacker"* para referirse a aquella persona experta en computadoras, con habilidades y conocimientos técnicos específicos para resolver un problema. Los mismos pueden ser clasificados en las siguientes categorías:

- **White Hat Hackers**: expertos en seguridad que utilizan distintas técnicas, como penetration testing, para evaluar que tan segura es la información de una organización.
- **Back Hat Hackers**: esta es la categoría más genérica para referirse a los *hackers* en general. En este grupo se encuentran aquellos que crean virus, se infiltran en redes, etc.
- **Script Kiddies**: este es un término derogatorio que se usa para referirse a aquellos *hackers* con pocos conocimientos, que simplemente utilizan herramientas creadas por alguien más.

- **Hacktivists**: atacantes motivados por razones políticas y/o también religiosas, que desean exponer a otras personas por sus actividades ilícitas.
- **Hackers patrocinados por estados**: hackers contratados por gobiernos, que ponen a su disposición recursos ilimitados, para atacar otros estados u organizaciones.
- **Hackers espías**: contratados por empresas con el objetivo de infiltrar a la competencia para robar información.
- **Ciberterroristas**: *hackers* que, motivados por creencias políticas o religiosas, intentan crear miedo y caos en la sociedad, atacando infraestructuras claves.

Por otro lado, han proliferado las comunidades en donde los *hackers* comparten conocimiento y, además, distribuyen sus herramientas de forma gratuita, o paga. Incluso suelen encontrarse publicaciones en las que se ofrece dinero a cambio del descubrimiento de nuevas vulnerabilidades en determinado sistema operativo o software.

3.2. Malware

La palabra *malware* proviene de la abreviación de ***mal****icious soft****ware***, y puede ser utilizado para comprometer funciones del sistema, robo de información, saltear controles de acceso, o cualquier otra forma de causar daño al *host* sobre el cual se está ejecutando.

3.2.1. Tipos de Malware

Los malwares pueden dividirse en distintas categorías dependiendo de su propósito. A continuación, se describen algunas clases:

- ***Adware***: Cuya abreviatura proviene de *advertising-supported software*, es un tipo de *malware* cuyo único propósito es la visualización de publicidad. Un ejemplo común podría ser la utilización de ventanas emergentes en sitios web o aquellos que son ejecutados por *software*. Muchas veces las aplicaciones ofrecen versiones que son "gratuitas"pero que están atestadas de publicidades. Muchos de estos *adwares* son patrocinados por empresas de publicidad y utilizados como una herramienta para generar ganancias monetarias.
- ***Bot***: Los *bots* son programas escritos para ejecutar determinadas operaciones automáticamente. Mientras que algunos *bots* son creados con propósitos no dañinos (juegos en línea, subastas en Internet, concursos en línea, etc.), su utilización con fines maliciosos se ha estado incrementando. Estos *bots* pueden ser utilizados en *botnets* (conjuntos de computadoras que son controladas por terceros) para efectuar ataques de DDoS (*Distributed Denial of Service*), *spambots*

que muestran publicidades en sitios web, *web spiders* que recopilan información de servidores, y para la distribución de *malware* disfrazado como resultados de búsquedas en sitios de descarga.

- ***Bug***: En el contexto del *software*, un *bug* es una falla del programa que produce resultados no deseados. Estas fallas son normalmente resultado del error humano y en general se encuentran en el código fuente o en el compilado de un programa. Ciertos *bugs* menores simplemente afectarán el funcionamiento del programa y pueden llevar tiempo hasta ser detectados. Sin embargo, *bugs* más significativos pueden producir caídas del sistema o que los mismos dejen de funcionar. Los *bugs* más peligrosos son los que ponen en riesgo la seguridad del sistema, ya que estos pueden ser aprovechados para saltear controles de autenticación de usuarios, sobrescribir privilegios, o robar información.

- ***Ransomware***: El *ransomware* es en esencia una forma de *malware* que toma cautivo al sistema de la computadora afectada y solicita una recompensa para su recuperación. Este *malware* restringirá el acceso del usuario a dicha computadora ya sea, encriptando los archivos que se encuentran en el disco duro o bloqueando completamente el sistema y mostrando mensajes con la intención de forzar al usuario a pagar al creador del *malware* una recompensa para remover las restricciones y de este modo recuperar el acceso a su computadora.

- ***Rootkit***: Un *rootkit* es un tipo de *software* malicioso diseñado para acceder o controlar remotamente una computadora sin ser detectado por usuarios o programas de seguridad. Una vez que el *rootkit* ha sido instalado, el atacante podrá ejecutar archivos, acceder/robar información, modificar configuraciones del sistema, alterar programas (como por ejemplo programas de seguridad que puedan detectar el *rootkit*), instalar programas maliciosos, o controlar la computadora como parte de una *botnnet*. La prevención, detección o remoción de este tipo de *malware* suele ser tarea difícil, dada la naturaleza sigilosa con la que operan.

- ***Spyware***: *Spyware* es un tipo de *malware* cuya función es espiar la actividad del usuario sin su conocimiento, como por ejemplo, captar las entradas de teclado y colectar datos (información de cuentas, autenticaciones, datos financieros).

- ***Trojan Horse***: Un *trojan horse*, también conocido como *troyano*, es un tipo de *malware* que se disfraza a sí mismo como un archivo normal para engañar a los usuarios a descargarlos e instalarlos. Un *trojan* puede darle al atacante acceso remoto a la computadora infectada. Una vez haya ganado acceso, éste podrá robar datos del usuario, instalar otros programas maliciosos, modificar archivos y monitorear la actividad del usuario.

- **Virus**: Un virus es un tipo de *malware* que es capaz de copiarse a sí mismo y esparcirse a otras computadoras adjuntándose ellos mismos a otros programas,

y ejecutando código cuando el usuario inicia dichos programas. Los virus pueden ser utilizados para robar información, dañar la computadora que lo aloja y/o la red, crear *botsnets*, robar dinero y mostrar publicidad.

- ***Worm***: Los *worms* de computadoras se encuentran entre los tipos más comunes de *malware*. Se esparcen entre las computadoras explotando las vulnerabilidades de los sistemas operativos a través de la red. Estos *worms* causan daño a la red de la computadora que lo aloja consumiéndole ancho de banda y sobrecargando los servidores *web*. Los *worms* de computadoras pueden ser clasificados como un tipo de virus, pero poseen varias características que los distinguen. La mayor diferencia es que los *worms* tienen la habilidad de auto replicarse y esparcirse independientemente, mientras que los virus requieren de las actividades de los humanos para esparcirse, como ejecutar un programa o abrir un archivo.

- ***Backdoor***: El *backdoor* es un tipo de *malware* que provee una "puerta trasera" del sistema para los atacantes. En sí mismo no causa ningún daño, pero provee a los atacantes el acceso al sistema, de modo que éste pueda hacer lo que desee con él.

- ***Keylogger***: La idea detrás de este *malware* es registrar todas las teclas presionadas por el usuario, y, de este modo, almacenar todos los datos provistos, incluyendo contraseñas, números de tarjetas de crédito, y cualquier otra información sensible.

- ***Remote-Access Trojan (RAT)***: Este tipo de *malware* permite al atacante ganar acceso remoto al sistema y realizar cualquier modificación que éste desee.

3.3. Métodos de detección

Todas las técnicas de detección de malware pueden ser divididas en dos grandes categorías, ***signature-based*** y ***behavior-based***. A su vez, existen dos conceptos fundamentales relacionados al análisis, los cuales se clasifican en: análisis estático y análisis dinámico del *malware*. El análisis estático, como su nombre lo indica, es realizado "estáticamente", sin la ejecución de un archivo. Mientras que, en el dinámico, el programa es analizado durante su ejecución, por ejemplo, en una máquina virtual.

3.3.1. Análisis estático

El análisis estático puede ser visto como la "lectura" del código fuente, se analiza su sintaxis y la estructura del archivo de propiedades del *malware* en un intento por inferir o determinar si existe comportamiento malicioso. El análisis estático puede incluir diversas técnicas:

- **Escaneo mediante antivirus**: Un antivirus es una herramienta que puede ser utilizada para la detección de software malicioso. Estos, normalmente poseen una base de datos con secciones de códigos sospechosos (*file signatures*), como así también el análisis y la concordancia con patrones de comportamientos para identificar archivos potencialmente maliciosos (*heuristics*).

- ***Hashing***: Es un método muy común que se utiliza para identificar unívocamente un *malware*. A través de un programa se analiza el software malicioso, generándose una etiqueta (*hash*) que lo identifica. Entre las funciones de hash más comúnmente utilizadas se encuentran: *Message-Digest Algorithm 5 (MD5)* y *Secure Hash Algorithm 1 (SHA-1)*.

- **Búsqueda de *strings***: La búsqueda o extracción de *strings* puede ofrecer información de las funcionalidades de un programa y su actividad sospechosa. Por ejemplo, si un *malware* crea un archivo, el nombre de dicho archivo es almacenado como un *string* en binario, o si el *malware* resuelve un nombre de dominio controlado por el atacante, el nombre del dominio será almacenado como un *string*. La extracción de *strings* desde su binario puede contener referencias a nombres de archivos, URLs, nombres de dominio, direcciones IP, comandos de ataque, claves de registro, etc. Si bien la extracción en sí misma no otorga un entendimiento completo acerca del propósito y de las capacidades del archivo, sí puede ofrecer pistas que pueden ser utilizadas para entender de qué es capaz el *malware*.

- **Empaquetamiento y ofuscamiento del malware**: Los creadores de *malware* frecuentemente utilizan técnicas de empaquetamiento y ofuscamiento para hacer que sus archivos sean más difíciles de detectar y analizar. Los programas son ofuscados para esconder la ejecución del *malware*. Los programas empaquetados pertenecen a un subconjunto de los ofuscados, cuyo objetivo es comprimir el programa malicioso de modo que no pueda ser analizado.

- **Archivo ejecutable de formato portable**: Los metadatos y formatos de un archivo pueden revelar información importante acerca de la funcionalidad de un programa. Los archivos con formato *Portable Executable (PE)* son utilizados por el sistema operativo de Microsoft Windows®, estos archivos poseen una estructura de datos con información que es utilizada por el *loader* del sistema operativo para la ejecución de archivos. Los archivos PE contienen un encabezado que incluye información acerca del código, el tipo de aplicación, las funciones de librerías requeridas, y requerimientos de espacio. La información que pueden arrojar estos encabezados son de gran valor para el análisis de *malware*.

- ***Linkeo* de librerías y funciones**: Una forma de recopilar información acerca de un ejecutable es analizando la lista de funciones que son importadas. El código de las librerías puede ser linkeado estáticamente, en tiempo de ejecución o

dinámicamente. De realizarse estáticamente, en el encabezado de los archivos PE podría verse qué librerías han sido incluidas en el código. El problema es que, en general, el método más comúnmente utilizado por los atacantes es el linkeo en tiempo de ejecución o dinámico, de modo que las librerías que vayan a utilizarse no serán solicitadas hasta que el programa las necesite.

- **Desensamblado**: La técnica de desensamblado, o *disassembler*, consiste en tomar el código compilado en binario y convertirlo a código *assembler* utilizando ingeniería inversa. El mismo luego es analizado para inferir su lógica e intenciones. Esta técnica es la más común y la más confiable para el análisis estático.

El análisis estático, a menudo, es llevado a cabo con la ayuda de ciertas herramientas. Pero, más allá del simple análisis, éstas pueden proveer información sobre técnicas de protección que utilizan los *malwares*. La principal ventaja del análisis estático es la posibilidad de descubrir todos los posibles escenarios de comportamiento. Investigar el código en sí mismo permite al analista conocer en mayor detalle cómo se ejecuta el *malware*, y no limitarse a la situación actual. Incluso, este tipo de análisis es más seguro que el dinámico, ya que el código no necesita ejecutarse y por lo tanto no pone en riego al sistema. Pero su ejecución es más costosa en tiempo. Es por ello que, si bien resulta una técnica interesante, la misma no es realizada en ambientes dinámicos del mundo real, tales como antivirus, sino más bien con propósitos de investigación, como por ejemplo el desarrollo de *signatures* para los *malware zero-day*.

3.3.2. Análisis dinámico

A diferencia del análisis estático, el análisis dinámico permite observar el verdadero funcionamiento del *malware*, porque, por ejemplo, la sola existencia de un *action string* en un binario no significa que la acción en efecto vaya a ejecutarse. El análisis dinámico es también una forma eficiente de identificar la funcionalidad del *malware*. Por ejemplo, si el *malware* es un archivo de *log* de un *keylogger*, el análisis dinámico provee la posibilidad de localizar dicho archivo en el sistema, descubrir qué tipo de registros almacena, descifrar a dónde es enviada dicha información, etc. Este tipo de análisis interno es muy difícil de realizarlo sólo con técnicas de análisis estático.

Si bien el análisis dinámico es una técnica extremadamente poderosa, debería ser realizada una vez que se haya completado el análisis estático, ya que el mismo puede poner en riesgo a la red y al sistema. Las técnicas dinámicas también tienen sus limitaciones, ya que no todos los caminos posibles pueden llegar a ejecutarse mientras el *malware* está en funcionamiento. Por ejemplo, en el caso de un *malware* que se ejecuta por línea de comando que requiere argumentos, cada argumento podría ejecutar diferentes funcionalidades del programa, y mientras no se sepan cuáles son las opciones, no será posible examinar dinámicamente todas las funcionalidades que puede realizar el programa.

Existen muchos productos de software que pueden ser utilizados para llevar a cabo el análisis dinámico, pero quizá el más popular es el uso de tecnologías *sandbox*. Un *sanbox* es un mecanismo de seguridad que permite ejecutar programas inseguros en un ambiente seguro, sin dañar la integridad del sistema real que se quiere proteger. Los *sandboxes* se componen de ambientes virtualizados, que a menudo ofrecen la simulación de una red de servicios, ofreciendo un modo seguro de ejecutar el *software* o *malware* que se desea testear.

3.3.3. Signature-based vs behavior-based

El método de análisis *signature-based* es un método estático que se basa en *signatures* predefinidas. Estos pueden ser archivos *fingerprints*, por ejemplo, *hashes* generadas con MD5 o SHA1, *strings* estáticos, archivos de *metadata*, entre otros. La detección del *malware*, en este caso, sería llevado a cabo de la siguiente manera: con la llegada de un nuevo archivo al sistema, el mismo es estáticamente analizado por el *software* del antivirus. Si existe alguna coincidencia con cualquiera de las *signatures* ya registradas, se dispara una alerta, indicando que el archivo es considerado sospechoso. Muchas veces, este tipo de análisis ya resultan suficiente, dado que muchos *malware* son detectados basándose en su valor de *hash*.

Sin embargo, los creadores de *malware* han comenzado a desarrollar programas que, de alguna manera, son capaces de cambiar su *signature*. Esta característica en los *malwares* es referida como polimorfismo. Por lo tanto, dada esta condición polimórfica, no pueden ser detectados solamente utilizando técnicas de detección basadas en *signatures*, hasta que la nueva *signature* haya sido creada. Esta situación llevó a las empresas desarrolladoras de antivirus a utilizar nuevas técnicas de detección. El análisis *behavior-based*, también conocido como *heuristics-based*, es un método en el cual se observa cómo se comporta el *malware* durante su ejecución, buscando señales de comportamiento malicioso: modificaciones de archivos de *host*, claves de registro, establecimiento de conexiones sospechosas, etc. En sí mismas, cada una de estas acciones, no representan necesariamente señales de *malware*, pero combinadas pueden elevar el nivel de sospecha acerca del archivo. Existe cierto umbral en el nivel de sospecha definido, por lo que cualquier *malware* que exceda este nivel dispará una alerta.

El nivel de la precisión de la detección de *malware* basado en comportamiento (*behavior-based*) dependerá de la implementación. Las más populares utilizan ambientes virtuales, como por ejemplo un *sandbox* para ejecutar el archivo y monitorear su comportamiento. Si bien este método consume más tiempo, es más seguro, ya que el archivo es chequeado antes de su ejecución. La principal ventaja del método de detección basado en comportamiento (*behavior-based*), es que, en teoría, puede identificar no sólo familias de *malware*, sino también, ataques *zero-day* (*malware* que

aún no ha sido identificado), y virus polimórficos. Sin embargo, si se tiene en cuenta el alto grado de esparcimiento del *malware*, tales análisis no resultan adecuados cuando se trata de *malwares* nuevos o polimórficos.

3.3.4. La necesidad de *Machine Learning*

Como se mencionó anteriormente, los detectores de *malware* basados en *signatures* pueden ser muy efectivos si el *malware* es uno ya conocido, que ya ha sido descubierto por alguna herramienta antivirus. Sin embargo, no resultan útiles para la detección de aquellos virus polimórficos, capaces de cambiar su *signature*. Por su parte, la precisión de los detectores basados en comportamiento (behavior-based), no siempre resulta ser la adecuada para la detección, dando como resultado cantidades de falsos positivos y falsos negativos.

La necesidad por encontrar nuevos métodos de detección está dada por el alto grado de propagación que poseen los virus polimórficos. Esto ha llevado a que se comenzaran a explorar nuevas alternativas capaces de brindar una solución a este problema. Métodos de detección y clasificación utilizando técnicas de *data mining* y *machine learning* han arrojado muy buenos resultados en este campo.

3.4. Resumen

Como se pudo ver durante el desarrollo de este capítulo existen diferentes motivaciones que se encuentran detrás de los ataques informáticos perpetuados por los *hackers*, entre las que se encuentran la ganancia económica, el daño a infraestructuras y la satisfacción personal.

También se realizó el estudio de los diferentes tipos de *malware* y cómo pueden clasificarse.

Por último, se analizaron cuáles son las técnicas de detección de *malware* que son utilizadas hoy en día, y por qué técnicas basadas en *machine learning* y *data mining* resultan una buena alternativa para la realización de dicha tarea.

El capítulo que se encuentra a continuación tendrá como objetivo realizar un estudio indicando en qué consiste el proceso del *data mining* y cuáles son las etapas que lo componen. También se estudiarán las técnicas utilizadas para realizar el preprocesamiento de los datos.

CAPÍTULO 4

Data Mining y Tratamiento de los Datos

Los últimos años no sólo han mostrado un importante incremento en el número y la complejidad de los ataques informáticos, sino también la cantidad y disponibilidad de los datos generados diariamente. Las capacidades de distintas disciplinas como el *data mining*, *machine learning*, estadísticas, entre otras, son necesitadas para abordar los desafíos de la ciberseguridad.

Data mining o minería de datos, es la extracción, o como su nombre lo indica, la "minería"de conocimiento a partir de una gran cantidad de datos. Los patrones o reglas descubiertas por estas técnicas pueden luego ser utilizadas para realizar predicciones respecto a nuevos datos. Las técnicas del *data mining* utilizan una combinación de estadísticas matemáticas, inteligencia artificial y reconocimiento de patrones, de modo de agrupar o extraer comportamientos o entidades. Por lo tanto, el *data mining* es un campo interdisciplinario que emplea el uso de herramientas de análisis de modelos estadísticos, algoritmos matemáticos y métodos de *machine learning* para descubrir patrones válidos, relaciones en un gran conjunto de datos, y otras características que aún no han sido descubiertas, lo cual resulta útil para encontrar técnicas y procedimientos utilizados por los *hackers* para vulnerar sistemas y obtener información.

En el siguiente capítulo se describirá en qué consiste el proceso del *data mining* y cuáles son los pasos para llevarlo a cabo. Se abordarán también conceptos relacionadas al tratamiento y preprocesamiento de los datos. Por último, se explicará de manera breve cuáles son las etapas del relacionadas al *machine learning* y qué compone cada una.

4.1. El proceso del *Data Mining*

Antes de comenzar el proceso del *data mining*, es importante determinar qué es lo que se quiere lograr implementándolo, y es lo que se conoce como "entendimiento del negocio"o (*business understanding*). Esta fase consiste en investigar los objetivos y requerimientos, y decidir si el *data mining* puede ser aplicado para alcanzarlos, determinando qué tipo de datos deben ser recolectados para construir un modelo desplegable.

La siguiente fase, consiste básicamente en el "entendimiento de los datos", o *data undertanding*, en donde el conjunto de datos inicial es analizado y estudiado para determinar si es apropiado para un futuro procesamiento. Si la calidad de datos es deficiente o pobre, quizá sea necesario recolectar nuevos datos basado en algún criterio más riguroso.

Las siguientes tres etapas son: la preparación de los datos, el modelado, y la evaluación. La fase de preparación involucra tareas de preprocesamiento de los datos crudos, para que, a partir de éstos, los algoritmos de *machine learning* puedan producir un modelo. Esta etapa de preprocesamiento puede incluir actividades que requieran la construcción de modelos, ya que muchas herramientas de preprocesado construyen modelos internos de los datos para transformarlos. De hecho, la preparación de los datos y el modelado, son etapas que van de la mano. Muchas veces se requiere iterar entre ellas, ya que los resultados obtenidos durante el modelado suelen dar una nueva perspectiva que puede afectar las técnicas de preprocesamiento elegidas.

La última fase, y quizá la más importante, ya que de esto depende el éxito de modelo, es la etapa de evaluación. Si durante la etapa de evaluación se determina que el modelo es pobre, será necesario reconsiderar el proyecto entero y regresar a la fase de "entendimiento del negocio "para identificar objetivos más fructíferos para la recolección de los datos. En cambio, si la precisión del modelo es suficientemente alta, entonces el siguiente paso será realizar un despliegue. Lo que normalmente puede significar la integración a un sistema mayor, o funcionar como un sistema en si mismo.

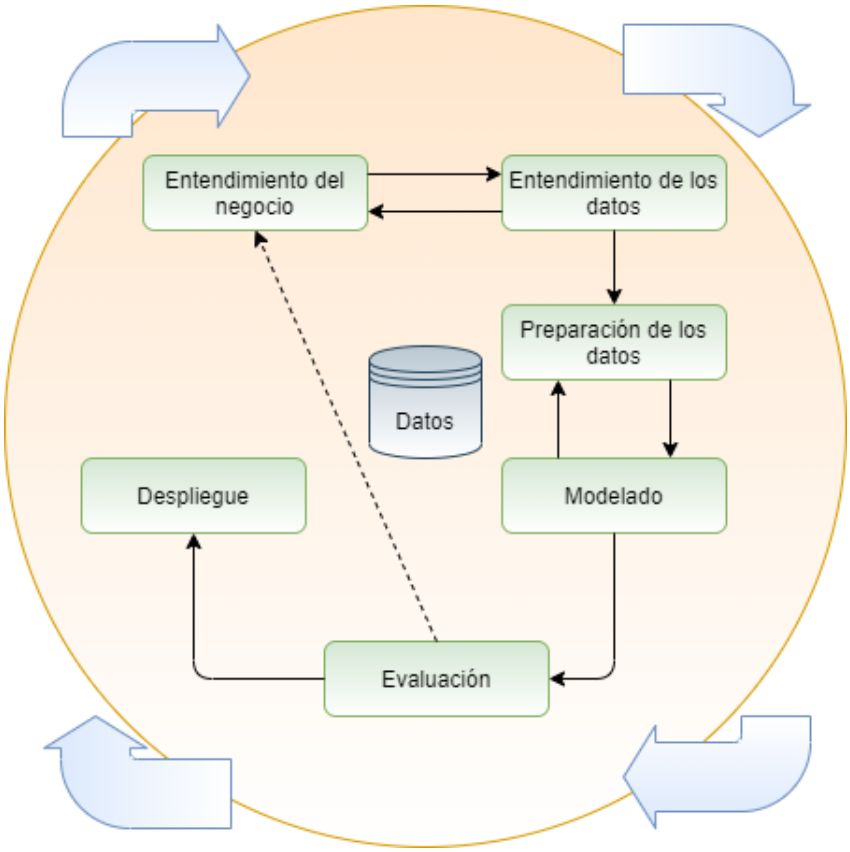

FIGURA 4.1: El proceso del *data mining*

4.2. Obtención de los datos

Para comenzar con el proceso del *data mining* será necesario obtener los datos, estos deberán ser recolectados y extraídos del mundo real. Dado que dichos datos podrán ser obtenidos de diferentes fuentes, es posible que posean distintos formatos. Los formatos para datos más conocidos y la forma más común de recolectarlos son:

- ***CSV***: Los archivos CSV (*Comma Separated Values*) son el formato de datos más comúnmente utilizado. Es también uno de los formatos más antiguos y todavía uno de los preferidos por los diferentes sistemas. Este tipo de archivos puede contener diferentes tipos de datos separados por coma, y pueden o no contener encabezados. Una variante de los CSV son los TSV, en donde el delimitador en lugar de ser una coma es un espacio tabular.

- ***JSON***:La *Java Script Object Notation* (JSON) surgió como una alternativa de los archivos XML. Es un formato de texto totalmente independiente del lenguaje con ciertas convenciones establecidas. Un archivo/objeto JSON es simplemente una colección de pares de clave/valor. Tal estructura de pares tiene su correspondiente representación en la mayoría de los lenguajes como "diccionarios".

- ***XML***: XML o *eXtensible Markup Language* es un lenguaje de marcas que define reglas para codificar sus datos/documentos. Al igual que los JSON, son legibles por humanos, es independiente de la plataforma y es simple de usar.
- ***Web Scraping***: Es una técnica que se utiliza para detectar y extraer información de la Web, principalmente de páginas web. El proceso de *web scraping* puede ser realizado manualmente copiando los datos, o utilizando técnicas automáticas para recorrer y extraer información de las páginas. Esta información puede ser luego utilizada por herramientas de análisis y almacenada. El proceso de *web scraping* se puede resumir de la siguiente manera: Un robot o *crawler* solicita a los servidores web un conjunto de direcciones predeterminadas, las analiza, identifica los enlaces externos que ésta pueda contener y los añade a la lista de direcciones a visitar, repitiendo el proceso en tanto le sea posible. Una vez obtenidas las páginas, se realiza la extracción de la información. La tarea de *scraping* consiste en utilizar técnicas como expresiones regulares, extracción basada en *XPath*, o etiquetas específicas, entre otras, para acotar la búsqueda de la información requerida en la página.
- ***SQL***: Las bases de datos datan de los años '70 y se utilizan para representar grandes volúmenes de información almacenados en forma relacional. Los datos se encuentran disponibles en tablas, o en alguna otra forma de estructura de datos. Si bien existen diferentes formas de trabajar con las bases de datos, quizás la más común dentro de este campo, es realizando las consultas SQL directamente.

4.2.1. Tipos de datos

En la sección previa se describió los distintos formatos y formas de extraer la información de ellos. Cada uno de estos formatos permiten representar datos de distintos tipos. Estos tipos de datos en su forma original forman los *features* de entrada que utilizarán los algoritmos de *Machine Learning*. A continuación, se explicarán los tipos más importantes de datos con los que se puede llegar a trabajar:

- **Numérico**: Es el más simple de los tipos de datos disponible. Los datos numéricos representan información escalar acerca de entidades que están siendo observadas, por ejemplo, el número de visitas de una página web, el precio de un producto, el peso de una persona, etc. Para la manipulación de los datos numéricos, se utilizan técnicas tales como normalización, discretización (*binning*, se trata de convertir un valor numérico en un valor nominal ordenado), cuantificación, entre otras, para la transformación de datos numéricos de acuerdo con los requerimientos.
- **Texto**: Es un tipo de dato compuesto de contenido alfanumérico, desestructurado y uno de los más comunes. Los datos textuales, cuando representan contenido del lenguaje humano contienen estructuras gramaticales implícitas

y significado. Este tipo de dato requiere un esfuerzo adicional para transformarlo y entenderlo.

- **Categórico**: Este tipo de dato se encuentra situado entre el tipo de dato numérico y el tipo texto. Las variables de tipo categóricas están asociadas a categorías de entidades con las que se están trabajando. Por ejemplo, si el tipo de cabello de una persona puede ser negro, castaño, rubio o pelirrojo; o la situación económica como de clase baja, media o alta. Los valores pueden ser representados como numéricos o alfanuméricos según se considere. De acuerdo a sus características, las variables categóricas pueden ser:
  - **Nominal**: Estos definen únicamente la categoría de los datos, sin tener en cuenta ningún tipo de orden. Por ejemplo, para el color de cabello negro, castaño, rubio o pelirrojo, donde las categorías no poseen ningún tipo de orden específico.
  - **Ordinal**: Define categorías, pero también establece un orden según reglas del contexto. Por ejemplo, gente categorizada por la situación económica en bajo, medio, alto, pueden ser consideradas en ese orden.

Es importante destacar que las operaciones matemáticas estándar como la suma, la resta, la multiplicación y la división, no tienen significado para las variables categóricas, aunque sintácticamente sea viable hacerlo (como es el caso de variables categóricas numéricas).

4.3. Preprocesamiento de Datos

Una vez obtenidos los datos, se procederá a su limpieza y transformación. Dentro de las tareas que involucra esta etapa se encuentran:

- **Filtrado de datos**: La limpieza del *dataset* involucra tareas de remoción y manipulación de datos erróneos, faltantes, imprecisos, valores atípicos (*outlier*), etc. También requiere de tareas de estandarización de nombres de atributos, para hacerlo más intuitivo y legible.
- **Casteo de tipos**: El *casteo* de tipos o conversión a los tipos apropiado de datos es una de las partes más importantes en esta etapa. A menudo, los datos son convertidos a tipos de datos incorrectos cuando son extraídos de su fuente original. Diferentes sistemas y plataformas manejan los tipos de datos de manera distinta, por lo que darles a los datos el tipo correcto es una tarea muy importante.
- **Transformación y adaptación de estructuras de datos**: Muchas veces, las estructuras de datos creadas para contener al *dataset* tienen que ser modificadas con el objetivo de facilitar uno o más pasos intermedios del procesamiento del

modelo. Nuevas columnas, o incluso nuevas estructuras transitorias pueden ser creadas para facilitar cálculos auxiliares, identificar determinadas observaciones y en casos donde se necesita cruzar dos o más estructuras de datos y no se dispone de medios obvios para hacerlos (como podría ser un identificador).

- **Manejo de valores faltantes o nulos**: Los valores faltantes pueden acarrear diversos problemas, desde ser interpretados incorrectamente por los algoritmos hasta errores en los cálculos y/o resultados finales. Un método muy común para rellenar los valores faltantes es la utilización del cálculo de la media.
- **Eliminación de duplicados**: Si bien es cierto que cuantos más datos se dispongan, mejor, también se debe tener presente que los datos duplicados en el *dataset* no agregan ningún valor adicional, por lo que deberían ser eliminados.
- **Manipulación de datos categóricos**: Los atributos de tipo categórico se refieren a aquellos datos alfanuméricos que pueden tomar un número limitado de valores. Por ejemplo, un *dataset* que contiene una columna género, cuyos valores son F (femenino) y M (masculino), estos datos, como sólo pueden tomar esos dos valores, son considerados categóricos.
- **Normalización de valores**: El proceso de normalización consiste en la escalado de los valores de los atributos. A este proceso de normalización también se lo conoce como *feature scaling*.
- **Manipulación de *strings***: Cuando lo que se quiere procesar es el lenguaje natural, éste consta de sus propias etapas que componen el proceso:
  - ***Tokenization***: Se divide el *string* en unidades que lo componen. Por ejemplo, dividir una sentencia en palabras o palabras en caracteres.
  - ***Stemming y lemmatization***:Consiste en normalizar las palabras para obtener su raíz o forma canónica. Mientras que el *stemming* es un proceso heurístico para lograr la forma canónica, la *lemmatization* utiliza reglas gramaticales y vocabulario para llegar a la raíz.
  - **Remoción de *stopword***:Los textos contienen ciertas palabras que aparecen con alta frecuencia pero que no agregan mucha información (signos de puntuación, conjunciones, etc.). Estas palabras/frases son removidas para reducir la dimensión y complejidad de los datos.

4.4. Selección e ingeniería de atributos

Una vez los datos hayan sido limpiados y transformados, se seleccionarán aquellas propiedades que fueron consideradas relevantes para formar lo que se conoce como "atributo" o *feature*, y serán representados dentro del conjunto de datos como columnas. Los atributos pueden ser de dos tipos: **inherentes**, son aquellos que se

obtienen directamente del *dataset*, sin tener que realizar ningún tipo de cálculo ni ingeniería; y los **derivados**, aquellos que se obtienen a partir de atributos existentes. Por ejemplo, a partir de la fecha de nacimiento de una persona se puede obtener su edad.

4.4.1. Ingeniería de atributos en datos categóricos

Los atributos (*features*) categóricos representan un conjunto finito de valores distintos. Estos valores pueden ser de tipo texto o numérico. Las cuales a su vez pueden pertenecer a variables categóricas nominales y ordinales. Dentro de los atributos categóricos nominales no existe el concepto de orden, mientras que en los ordinales sí.

Cuando los valores son alfanuméricos, es necesario transformar estos atributos a valores numéricos, dado que un modelo matemático no puede trabajar con valores alfanuméricos. Esta transformación se puede realizar utilizando un esquema denominado *dummy variable* (variables tontas), en la cual, si se tienen m valores de categoría se crearán m – 1 columnas, las cuales se completarán con valor 0 o 1, según corresponda. Excepto cuando se trate de la categoría no representada, para la cual se completarán con ceros para todas las demás.

4.4.2. Normalización de atributos

Muchas veces es necesario trabajar con valores numéricos que son muy diferentes entre sí. Si se utilizan como datos de entrada al modelo estos valores tal y como están, pueden resultar en modelos sesgados o cuyas magnitudes son demasiado altas. Algunas técnicas que se utilizan para realizar la normalización son:

- **Escalado estándar**: Intenta estandarizar cada valor de la columna que contiene el atributo, eliminando el valor medio y escalando la varianza a 1 de cada uno de los valores. A este proceso también se lo conoce como centrado y escalado, y puede ser denotado matemáticamente como:

$$SS(X_i) = \frac{X_i - \mu_x}{\sigma_x} \tag{4.1}$$

  Donde, a cada valor del atributo X se le resta la media μ_x y este resultado es dividido por la desviación estándar σ_x.

- **Escalado min-max**: Se transforma y escala cada valor del atributo tal que dicho valor esté dentro del rango [0-1]. En términos matemáticos sería:

$$MMS(X_i) = \frac{X_i - min(X)}{max(X) - min(X)} \tag{4.2}$$

Donde, se escala cada valor del atributo X sustrayéndole el valor mínimo (X) y dividiendo el resultado por la diferencia entre los valores máximo y mínimo del atributo.

- **Escalado robusto**: Una gran desventaja de la escala Min-Max es que muchas veces la presencia de *outliers* (valores atípicos) afectan la escala para cualquier valor del atributo. La escala robusta utiliza medidas estadísticas para escalar los atributos y no verse afectado por los *outliers*. Matemáticamente esta escala se puede representar cómo:

$$RS(X_i) = \frac{X_i - media(X)}{IQR_{(1,3)}(X)} \tag{4.3}$$

Donde, se escala cada valor del atributo X sustrayéndole la media de X y dividiendo el resultado por el *IQR (Inter-Quartile Range)* de X, el cuál sería el rango (diferencia) entre el primer cuartil y el tercer cuartil.

4.4.3. Selección de atributos

Muchas veces no resulta conveniente trabajar con *datasets* que cuentan con quizá cientos de atributos. Grandes conjuntos de atributos conducen a modelos complejos y difíciles de interpretar, y posiblemente *overfitting*. Por lo tanto, el objetivo es seleccionar un número óptimo de atributos que sean representativos del problema que se quiere abordar, y de este modo evitar las dificultades antes mencionadas.

Las estrategias para la selección de atributos pueden ser divididas en tres grandes áreas:

- **Métodos de filtrado**: Estas técnicas seleccionan atributos basándose puramente en métricas como la correlación, información mutua, etc. Estos métodos no dependen de los resultados obtenidos de ningún modelo, y normalmente buscan la relación de cada variable con la variable que se quiere predecir. Entre los más populares se encuentran los *Threshold-Based Method* (métodos basados en umbral) y tests estadísticos.

- **Métodos de envoltorio**: Se estudia la interacción entre múltiples atributos y, mediante eliminaciones recursivas se intenta obtener un subconjunto de estos atributos que den como resultado un modelo más eficiente. Los métodos de selección hacia atrás y selección hacia adelante son los más populares en esta categoría.

- **Métodos embebidos**: Estas técnicas combinan los beneficios de los otros dos métodos, asignándoles un puntaje (*score*) a cada atributo basado en la importancia. Árboles de decisión y *Random Forests* son los más populares dentro de esta categoría.

Entre las técnicas más representativas para la selección de atributos, se encuentran:

- **Threshold-based Method**: Esta estrategia de selección de atributos permite establecer un umbral para limitar la cantidad de atributos que posee del modelo. Estos umbrales pueden ser utilizados de diferentes formas, por ejemplo, establecer un mínimo y/o un máximo, e incluso es posible utilizar la varianza de modo que aquellos atributos cuya varianza sea baja sean removidos.
- **Métodos estadísticos**: Consiste en la selección de atributos basados en la utilización de tests estadísticos. Existen varios test estadísticos que pueden ser usados para regresión y clasificación que incluyen, información mutua, ANOVA (análisis de varianza) y tests *Chi-Square*. Basados en los puntajes obtenidos de estos tests, es posible seleccionar los atributos que arrojen los mejores resultados.
- **Eliminación recursiva de atributos**: La *Recursive Feature Elimination*, también conocida como RFE, es una técnica de selección de atributos basados en envoltorios, la cual permite con la ayuda del modelo de estimación del *Machine Learning* rankearz asignar puntajes a los atributos para luego eliminar recursivamente aquellos que tengan el menor hasta que se llegue un número específico preestablecido.
- **Selección basada en el modelo**: Modelos basados en árboles como los de decisión o modelos de ensamble como *random forest* (o árboles de ensamble) pueden ser utilizados no sólo para modelar sino también para la selección de atributos. Estos modelos pueden establecer la importancia de los atributos mientras son construidos seleccionando aquellos que posean la mejor puntuación y descartando aquellos de puntuación irrelevante.
- **Extracción de atributos con *Principal Component Analysis* (PCA)**: Es un método estadístico que utiliza procesos de álgebra lineal para transformar un conjunto de atributos de alta dimensionalidad a uno de menor dimensionalidad, con un mínimo de pérdida de información.

4.5. Visualización de los datos

Finalmente, tiene lugar la visualización de los datos. Éste es un punto muy importante, ya que la utilización de ayudas visuales, como gráficos, imágenes o mapas, constituyen una herramienta de mucho valor, tanto para la exploración y validación de los datos disponibles, así como también para la detección y corrección temprana de errores que, de otra manera, podrían propagarse a los modelos de *machine learning*, generando resultados inesperados o errores difíciles de identificar y corregir.

4.6. Resumen

El *data mining*, o minería de datos, es un proceso de identificación de información relevante extraída, la mayoría de las veces, de grandes volúmenes de datos, con el objetivo de descubrir patrones y tendencias, estructurando la información obtenida de un modo comprensible para su posterior utilización.

Este proceso es iterativo y culminará cuando la información obtenida satisfaga las expectativas de conocimiento esperadas, de no ser así el proceso se repetirá utilizando nuevas variables y adoptando técnicas distintas a las usadas en procesos anteriores hasta obtener un modelo de datos deseado.

El capítulo que se encuentra a continuación enunciará conceptos de *machine learning*, comenzando con su definición para luego abordar características más específicos relacionados a dicho tema.

CAPÍTULO 5

Conceptos de Machine Learning

De acuerdo a la definición clásica dada por el pionero en Intenligencia Artificial Arthur Lee Samuel, *machine learning* es un conjunto de métodos que da a las computadoras "la habilidad de aprender sin ser programadas explícitamente".

En otras palabras, un algoritmo de *machine learning* descubre y generaliza las características subyacentes a los datos que observa. Con este conocimiento, el algoritmo puede "inferir"las propiedades de aquellas muestras que aún no ha visto. Todas estas características obtenidas de los datos, formalizados matemáticamente es lo que se conoce como "modelo".

En la detección de *malware*, estas muestras que no han sido vistas estarían representadas por archivos `ASM` y `bytes`, los cuales podrían ser benignos o malignos (*zero-day malware*).

En el siguiente capítulo, se expondrán conceptos relacionados al *machine learning*, comenzando con su definición, las etapas que lo componen y cómo es su funcionamiento en términos matemáticos.

5.1. Definición

Originalmente, el término *machine learning* fue acuñado por Arthur Lee Samuels en 1959, un pionero en el área de la inteligencia artificial y juegos de computadora, y lo definió como "la habilidad que tienen las computadoras de aprender sin ser explícitamente programadas".

En 1997, Tom Mitchel formalizó una defínción en términos matemáticos y racionales indicando que:

> *"Se dice que un programa de computadora aprende con una experiencia E si, para un conjunto de tareas T que realiza con una medición de performance P, dicha performance es incrementada por la experiencia E."*

En donde, una tarea *T* consiste básicamente en definir las tareas que se ejecutarán para resolver un problema del mundo real, entre las cuales podrían ser clasificación o categorización, la regresión, agrupamiento o *clustering*. La experiencia *E* la obtienen los algoritmos de *machine learning*, o modelos, a partir de lo que aprenden del conjunto de datos. Este proceso de ir ganando experiencia es iterativo y es conocido como "entrenamiento" del modelo. En cuanto a la performance *P*, es una medida cuantitativa, o métrica, para determinar que tan bien el algoritmo o modelo está ejecutando la tarea *T*, con experiencia *E*. Medidas típicas utilizadas son la precisión, la exactitud, etc.

5.2. Surgimiento del *Machine Learning*

Como se mencionó en la sección referida al avance de la Inteligencia Artificial, la idea de máquinas conscientes e inteligentes no es algo nuevo, sin embargo, gracias a los avances de estos últimos años, es posible procesar grandes volúmenes de datos a una escala sin igual. Es así que se podría decir que vivimos en una Era de Información, siendo uno de los principales desafíos de las empresas y organizaciones el poder obtener información a partir de todos estos datos, que les permita tomar mejores decisiones (*data-driven decisions*).

Para poder tomar decisiones inteligentes a partir de una gran cantidad de datos no alcanza con los paradigmas de programación tradicionales. El programador debería contar con un conocimiento muy amplio del dominio sobre el cual está trabajando para poder determinar todas las correlaciones existentes entre las distintas variables de sus datos. Luego, debería invertir mucho tiempo en la codificación de conjuntos de reglas extremadamente complejos para intentar llevar a cabo cualquier tipo de análisis (un sistema basado en reglas) sobre los datos disponibles. Esta aproximación presenta varias deficiencias. Por un lado, ciertas relaciones podrían ser menos evidentes y podrían ser pasadas por alto. Por otro lado, este tipo de sistemas son muy costosos de mantener. Cualquier cambio que se introduzca, ya sea para crear una nueva regla, modificar o eliminar una existente, es costoso y puede introducir errores.

Así surgió el concepto de *machine learning* o aprendizaje automático, como una necesidad de tomar decisiones más rápidamente y de mejor calidad. A diferencia de los paradigmas tradicionales, basados en reglas, *machine learning* utiliza un conjunto de datos, denominados observaciones, para la construcción de un modelo. Este modelo utilizará dichos datos para deducir todas los posibles patrones y correlaciones existentes entre ellos. A partir de este conocimiento adquirido, el modelo será capaz de predecir valores de salida para nuevas observaciones no vistas anteriormente.

Este proceso de aprendizaje del modelo consta, básicamente, de dos etapas:

- *Training* (entrenamiento): su objetivo es aprender a partir de un conjunto de datos conocidos. Para ello se deben utilizar conceptos matemáticos (principalmente de Álgebra Lineal y Estadística) y algoritmos de aprendizaje.
- *Testing* (evaluación): a partir de lo "aprendido" en la etapa anterior, en esta etapa se predicen, o infieren, resultados de acuerdo con nuevos datos.

Estas dos etapas de entrenamiento y evaluación, también son conocidos como aprendizaje y predicción, respectivamente.

5.3. Etapas del proceso de *Machine Learning*

El proceso de *machine learning* podría ser generalizado en tres grandes tareas:

- **Representación**: Consiste en representar el problema utilizando un lenguaje formal. En esta primera etapa es donde se seleccionará el algoritmo de *machine learning* que se utilizará. Por ejemplo, si observando los datos se determina que se tiene un problema de regresión, entonces es probable que se se elija Regresión Lineal como modelo para representarlo. Existen diferentes tipos de modelos, los cuales pueden ser clasificados de acuerdo con sus categorías y nomenclaturas. Muchos de ellos se basan en el algoritmo de aprendizaje que utilizan o en el método que emplean para construirlos. Por ejemplo, un modelo puede ser lineal o no lineal, paramétricos o no paramétricos, supervisado, no supervisado o semi-supervisado, modelo de ensamble o incluso un modelo basado en *deep learning*. Es en esta etapa donde también se seleccionan los parámetros / pesos / coeficientes del modelo que serán utilizados.
- **Evaluación**: Una vez decidida la representación y los posibles modelos, se necesitará algún criterio para evaluarlos y, de este modo, poder seleccionar el mejor dentro de un conjunto de candidatos. En general se utilizan métricas que retornan algún valor de performance numérico que ayude a decidir la efectividad del candidato.
- **Optimización**: La etapa final del proceso es la optimización. La optimización se puede describir como la búsqueda a través de todas combinaciones de posibles hiperparámetros, para encontrar aquella que dé como resultado el modelo más óptimo.

5.4. El conjunto de datos

El conjunto de datos que utilizarán los algoritmos está compuesto por una serie de observaciones sobre el dominio que se está estudiando. A su vez, estas observaciones están conformadas por variables independientes, también conocidas como

input variables o *features*, que son las distintas características propias de la observación; y una variable dependiente denominada también como *target*, *label* o *output*, que representa aquella característica del dominio que se desea estudiar.

Formalmente, las variables independientes se denotan con la letra X y un subíndice que las identifica. De esta manera, $X_1, X_2, ..., X_n$ conforman la totalidad de las variables independientes de nuestro conjunto de datos. Por otro lado, la variable dependiente es comúnmente denotada con la letra Y. Así, podemos formular la siguiente ecuación:

$$Y = f(X) + \epsilon \tag{5.1}$$

La fórmula anterior expresa a la variable dependiente Y en términos de una función f de las variables independientes $X_1, X_2, ..., X_n$ que es fija, pero desconocida. También se incluye un término de *error*, que es independiente de X y cuya media, como se verá más adelante, es cero. f representa la información sistemática que la variable X genera respecto de Y pero, como ya se ha mencionado, esta función f suele ser desconocida, por lo que debe ser estimada a partir de la información de las observaciones que se disponen.

5.5. Tipos de estimación

Existen dos grandes razones que justifican la necesidad de estimar f: **predecir** e **inferir**.

5.5.1. Predicciones

En muchos escenarios se dispone de un gran número observaciones para las distintas variables independientes X. Sin embargo, el valor de Y puede ser difícil de obtener. Siguiendo bajo la asunción que el término de error ϵ tiende ser cero, es posible formular:

$$\hat{Y} = \hat{f}(X) \tag{5.2}$$

En donde $\hat{f}$ representa la estimación de la función f y en donde $\hat{Y}$ es la predicción que se obtiene de Y. En este contexto, la forma exacta de $\hat{f}$ no es necesariamente importante y puede verse como una caja negra. Lo que realmente importa es que produzca predicciones lo suficiente aproximadas a Y.

Por ejemplo, se puede suponer que las variables independientes $X_1, X_2, \ldots, X_n$ representan las distintas características de la muestra de sangre de un paciente, siendo la variable dependiente Y el riesgo que el paciente sufra una severa reacción adversa

a determinado medicamento en particular. De este modo, resulta importante poder estimar Y a fin de evitar el suministro de la droga a aquellos pacientes para los que la predicción arroje un valor alto. Dicha predicción $\hat{Y}$ tendrá una precisión que estará marcada por dos valores: el **error reducible** y el **error irreducible**.

Como se ha mencionado anteriormente, la estimación $\hat{f}$ de la función f no será perfecta. El error que se produzca en esta estimación es lo que se denomina **error reducible**, ya que el modelo utilizado puede ser potencialmente mejorado para aumentar la precisión y reducir el error. Sin embargo, por mucho que se mejore el modelo, siempre habrá un pequeño error. Esto se debe a que la variable dependiente Y está también definida por el término de error ϵ. En el ejemplo anterior, este error está asociado a factores tales como el estado de salud general del paciente en un día particular o pequeñas variaciones en la fabricación del medicamento. Todos estos factores, que no pueden ser medidos por el modelo, son conocidos como **error irreducible**, ya que no puede ser eliminados.

Retomando la Ecuación 5.2, podemos plantear la siguiente ecuación:

$$E(Y - \hat{Y})^2 = E[f(X) + \epsilon - \hat{f}(X)]^2 = \underbrace{(f(X) - \hat{f}(X))^2}_{reducible} + \underbrace{Var(\epsilon)}_{irreducible} \tag{5.3}$$

El primer término de la expresión representa la esperanza del cuadrado de la diferencia entre el valor real y el predicho para f. Este término es muy importante, ya que conforma la base de una técnica para cálculo de errores en diversos modelos, conocida como *Mean Squared Error*.

5.5.2. Inferencias

En ciertos casos el foco está puesto no en predecir un valor, sino en entender la relación que existen entre las variables independientes y la variable dependiente, es decir, cómo Y cambia en función de $X_1, X_2, \ldots, X_n$. En este contexto, $\hat{f}$ no puede ser vista como una caja negra ya que debe conocerse la forma exacta que posee. Los puntos más importantes que deben ser estudiados son:

- Qué variables independientes están asociadas con la dependiente y en qué grado. Este análisis puede determinar que sólo un pequeño subconjunto de los atributos están realmente relacionados con la variable dependiente. Esto, además de proveer valiosa información, permitiría reducir la complejidad del modelo.
- Cuál es la relación entre las variables independientes y la variable independiente. No sólo es importante descubrir aquellos atributos que guardan una alta correlación con la variable dependiente. También es necesario entender la

naturaleza de dicha relación. Incrementos en determinadas variables independientes pueden conducir a incrementos o decrementos de la variable *target*. Otras variables independientes pueden tener el efecto opuesto.

- Qué tan compleja es dicha relación. Una vez identificadas las correlaciones más importantes y establecido de qué manera influyen, es importante determinar si son lo suficientemente simples como para ser modeladas a través de ecuaciones lineales, o si la relación es más compleja y otro tipo de modelo es necesario.

Ejemplos típicos de inferencia se observan en áreas como finanzas, *marketing* y ventas, cuando se estudian las distintas características que hacen que un producto se venda mejor, o cuando se quiere analizar cómo influye en la rentabilidad de un negocio factores como su ubicación, proximidad con la competencia, precios y esquemas de descuentos.

5.6. Métodos de estimación de f

En términos generales, existen dos formas de estimar f, que pueden ser caracterizados como métodos paramétricos y métodos no paramétricos.

5.6.1. Método paramétrico

El método paramétrico consta de dos pasos:

1. Se realiza alguna asunción en cuando a la forma de f. Una primera asunción bastante frecuente es que f tiene una forma lineal. De esta manera, se puede expresar f de la siguiente manera:

$$f(X) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_n X_n \tag{5.4}$$

   Dado que los distintos valores de las variables independientes $X_1, X_2, \ldots, X_n$ son conocidos, la tarea de estimar f se reduce significativamente. En lugar de tener que estimar una función $n - dimensional$ aleatoria, sólo es necesario estimar los $n + 1$ coeficientes.

2. Una vez que se dispone de un modelo, el siguiente paso consiste en utilizar la información de las observaciones que se poseen para entrenar el modelo. En el caso de un modelo lineal como el presentado en la Ecuación 5.4, la tarea consiste en estimar los coeficientes $\beta_0, \beta_1, ..., \beta_n$ a modo tal que:

$$Y \approx \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_n X_n \tag{5.5}$$

5.6.2. Método no paramétrico

A diferencia de los métodos paramétricos, los no paramétricos no realizan ninguna asunción en cuanto a la figura de f, sino que el eje está puesto en encontrar las estimaciones de f que más se aproximen al valor real. Esto presenta una ventaja importante, dado que, a no asumir linealidad, los modelos no paramétricos tienen la posibilidad de adaptarse mejor a un grupo más variado de formas de f.

5.7. El balance entre precisión e interpretabilidad

Existen distintos tipos de métodos de estimación de f. Algunos producen modelos más restrictivos, como el caso de las regresiones lineales que ya se han mencionado, que sólo son capaces de producir funciones lineales; y otros menos restrictivos, que se adaptan mejor a una familia más amplia de problemas. Los modelos menos flexibles, si bien son más fáciles de entender y visualizar, producen resultados menos acertados. En teoría es razonable suponer que uno siempre elegiría aquellos modelos que producen mejores estimaciones, pero, en la práctica, éste no es siempre el caso.

Existen diversas razones por las que se podría preferir un método inflexible. Por ejemplo, en los casos de inferencias, como lo que se estudia es la correlación entre las variables independientes y la dependiente, puede ser preferible sacrificar cierto nivel de precisión con el objetivo de obtener un modelo más simple de entender y transmitir.

5.8. Evaluación de la precisión de un modelo

Muchas veces la utilización de algún método de aprendizaje puede lograr muy buenos resultados para un conjunto determinado de datos, y otros que pueden incluso resultar mejor para ese conjunto y peor para otros. Por lo que resulta muy importante poder establecer, dado el conjunto de datos que se tiene, qué método produce el mejor resultado.

5.8.1. Calidad del ajuste (*Quality of fit*)

Una de las principales medidas que se pueden tomar para evaluar la performance de un modelo frente a un conjunto de observaciones consiste en evaluar qué tan aproximadas fueron las predicciones con respecto a los valores reales. En los modelos de regresión la técnica usada más frecuentemente es la del cálculo del error cuadrático medio (*Mean Squared Error*) dado por la siguiente fórmula:

$$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{f}(x_i))^2 \tag{5.6}$$

Donde $\hat{f}(x_i)$ es la predicción de $\hat{f}$ para la i-ésima observación. El MSE (por sus siglas en inglés) será pequeño mientras que los valores de las predicciones se mantengan relativamente cercanos a los verdaderos valores de y.

El error cuadrático medio puede ser computado sobre el *training set* (lo que se conoce como *training* MSE) para poder tener una primera impresión de qué tan preciso es el modelo. Sin embargo, este cálculo no es realmente importante, dado que lo que realmente interesa es conocer qué tan acertadas serán las predicciones frente a información nueva.

Matemáticamente, $\hat{f}$ es estimada a partir de un *training set* con observaciones $(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$. Así se obtienen estimaciones $\hat{f}(x_1), \hat{f}(x_2), ..., \hat{f}(x_n)$. Si estas estimaciones son bastante cercanas a $y_1, y_2, ..., yn$, se tendrá un *training* MSE bajo. Sin embargo, el interés no está en saber si $\hat{f}(x_1) \approx y_1$, sino en determinar si $\hat{f}(x_0) \approx y_0$, siendo (x_0, y_0) una observación nueva, no utilizada durante el entrenamiento del modelo. Luego de procesar un número considerable de observaciones no utilizadas durante la fase de entrenamiento, se puede calcular el *test* MSE como:

$$Ave(y_0 - \hat{f}(x_0))^2 \tag{5.7}$$

Donde (x_0, y_0) son todas las observaciones nuevas y *Ave* es el cálculo del promedio (*average*) de dichas observaciones. La función anterior es, simplemente, otra forma de expresar el cálculo del error cuadrático medio como se mostró en la Ecuación 5.6.

Desafortunadamente, no siempre se cuenta con un *test set*, por lo que debe elegirse otra alternativa para calcular el *test* MSE. En estos casos lo que se suele hacer es tomar el *training* MSE, asumiendo que el modelo arrojará para el *test set* predicciones con un MSE semejante al obtenido durante la etapa de entrenamiento. Pero existe un problema fundamental con esta estrategia, ya que no hay ninguna garantía de que un modelo que produjo en la fase de entrenamiento el MSE más pequeño, también lo haga durante la etapa de test.

5.8.2. Calidad de ajuste en clasificación

En la sección anterior se habló de la calidad del ajuste para los modelos de regresión, sin embargo muchos de los conceptos enunciados también aplican para los modelos de clasificación, con una sóla modificación y es que ahora y_i ya no es numérica. Por ejemplo, si lo que se busca es estimar f en las bases de las observaciones de entrenamiento $(x_1, y_1), ..., (x_n, y_n)$, en donde ahora $y_1, ..., y_n$ son cualitativos; la forma más común para cuantificar la precisión del estimador $\hat{f}$ es el *error rate* del entrenamiento, lo que representa la proporción de errores si se utilizara el estimador $\hat{f}$ en las observaciones del conjunto de entrenamiento, y se encuentra dada por:

$$\frac{1}{n}\sum_{i=1}^{n} I(y_i \neq \hat{y}_i) \tag{5.8}$$

En donde, $\hat{y}_i$ es el *label* de la clase predicha para la observación i-écima, utilizando el estimador $\hat{f}$. Y $I(y_i \neq \hat{y}_i)$ es un *indicador de variable* que es 1 si $y_i \neq \hat{y}_i$ y 0 si $y_i = \hat{y}_i$. Si $I(y_i \neq \hat{y}_i)$ es igual a 0, entonces la i-écima observación fue clasificada correctamente por el método de clasificación, caso contrario, fue clasificada incorrectamente. Por lo tanto, la ecuación 5.8 computa la fracción de clasificaciones incorrectas. Esta ecuación, hace referencia al *training error rate* porque es computada basada en los datos que fueron usados para entrenar el clasificador.

Al igual que en el caso de la regresión lineal, el interés esta centrado en la proporción de errores que resultan de aplicar el clasificador a las observaciones de test que no son utilizadas en el entrenamiento. El *test error rate* asociado con el conjunto de observaciones de test de la forma (x_0, y_0) está dado por:

$$Ave(I(y_0 \neq \hat{y}_0) \tag{5.9}$$

En donde, $\hat{y_0}$ es el *label* de clase predicho que resulta de aplicar el clasificador a las observaciones de test con predicador x_0. Un buen clasificador será aquel para el cual el error en el test sea el más pequeño.

5.8.3. *Overfitting y Underfitting*

Cuando se está entrenando un modelo es deseable que el mismo tenga el menor *train* MSE posible. Sin embargo, si se continúa aumentando la flexibilidad del modelo con el sólo objetivo de reducir este valor, el modelo trabajará de manera intensa para aprender todo lo que puede de las observaciones que se la han dado, y empezará a adaptarse para incorporar en él pequeñas variaciones en las observaciones, que pueden estar dadas por fluctuaciones azarosas.

5.8.4. Balance entre sesgo y varianza

Estas dos medidas se relacionan con la capacidad de ajuste y la generalización del modelo. Cuando se logra un buen ajuste, la diferencia entre los datos reales y la estimación del modelo es pequeña, en cuyo caso el sesgo también será pequeño; pero esto buenos resultados van de la mano con el aumento de la complejidad del modelo. Cuando se aumenta la complejidad del modelo este se vuelve sensible a las pequeñas variaciones en los datos de entrada, fluctuando en función de los datos, por lo que la varianza aumenta, razón por lo cual será muy importante encontrar un balance entre el sesgo y la varianza.

En términos más formales, el sesgo y la varianza pueden ser descriptos como:

- **Varianza**: se refiere a la cantidad en la que $\hat{f}$ variaría si se hubiera entrenado al modelo con *training set* diferente. Dado que el modelo aprende a partir de los datos que le son suministrados en la etapa de entrenamiento, es natural suponer que distintos datos producirán un modelo distinto. Si se observa una varianza alta, esto quiere decir que pequeños cambios en el *training data* tendrán un alto impacto en la estimación de $\hat{f}$. Por lo general se asume que cuanto más flexible es el método, más alta es su varianza. Luego de haber analizado la Subsección 5.8.3 esto debería resultar intuitivo. Más flexible es el método, más tratará de ajustarse a cada una de las observaciones disponibles.
- **Sesgo**: es el error inherente que existe en un modelo al querer representar un problema complejo de la realidad. En otras palabras, el sesgo presenta una medida para evaluar qué tan bien un método se adapta a la realidad que está intentando modelar. En este caso, la relación con la flexibilidad del modelo es inversamente proporcional: un método más flexible producirá un sesgo menor.

De este modo, el *test* MSE está dado por la velocidad de crecimiento/decrecimiento de cada una de estas dos propiedades, lo que puede ser expresado matemáticamente de la siguiente manera:

$$Bias\big(\hat{f}(x)\big) = E\big(\hat{f}(x) - f(x)\big) \tag{5.10}$$

La figura Figura 5.1 permite observar gráficamente el impacto de la varianza y el sesgo en un modelo. Una varianza alta producirá valores más dispersos. Por otro lado, un sesgo elevado producirá valores más alejado del centro (el valor real).

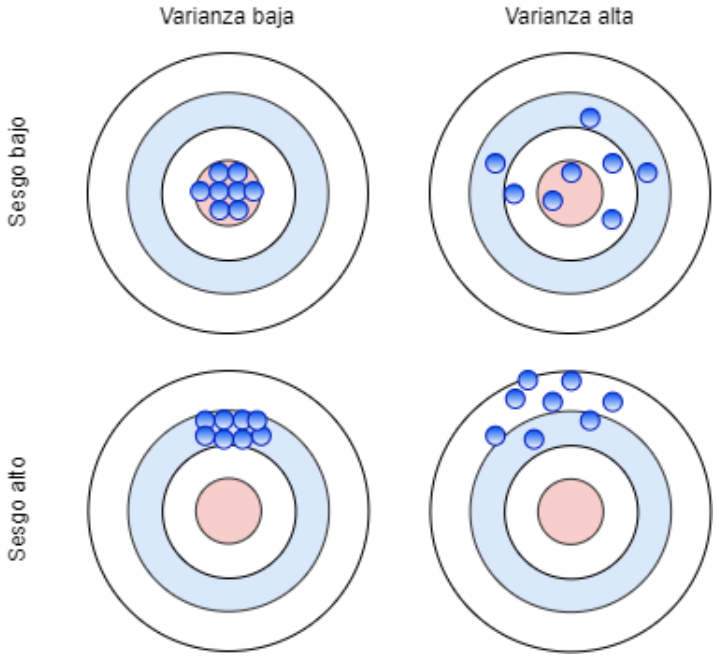

FIGURA 5.1: Sesgo y varianza

5.9. Clasificación de los métodos de aprendizaje

Los métodos de aprendizaje pueden ser clasificados en cuatro grandes categorías:

1. **Aprendizaje supervisado**: Los métodos o algoritmos de aprendizaje supervisado realizan su entrenamiento con los datos de entrada (conjunto de entrenamiento) y sus respectivos datos de salida (conocido como etiqueta o *label*). Expresado en términos matemáticos, la idea consiste en que el algoritmo pueda aprender las relaciones que existen entre las variables independientes $X_1, X_2, ..., X_n$ y la variable dependiente Y, y de este modo utilizar el aprendizaje para realizar predicciones sobre nuevos datos o datos desconocidos.

2. **Aprendizaje no supervisado**: El aprendizaje supervisado requiere que el modelo sea entrenado con una serie de observaciones tanto de las variables independientes, como de la dependiente. Sin embargo, en muchos casos la información de la variable dependiente es desconocida, por lo que es imposible entrenar un modelo. Son en estas situaciones en las que un modelo no supervisado es extremadamente útil. Estos algoritmos intentan aprender cuáles son las relaciones, patrones o estructuras internas inherentes a los datos sin ningún tipo de ayuda o supervisión, como en el caso del aprendizaje supervisado. El aprendizaje no supervisado se concentra más en intentar extraer concocimiento o información útil de los datos, más que en predecir salidas.

3. **Aprendizaje semisupervisado**: Los métodos de aprendizaje semisupervisados son una combinación de los métodos supervisados y no supervisados. Estos métodos normalmente utilizan para su entrenamiento una gran cantidad de

datos que no poseen etiqueta, y una pequeña que si. Existen múltiples técnicas disponibles en la forma de métodos generativos, gŕaficos basados en métodos, métodos basados en heurística, etc.

4. **Aprendizaje por refuerzos**: En el caso de los métodos de aprendizaje por refuerzos, lo que se tiene es un agente que se quiere entrenar en un ambiente determinado y durante un período de tiempo, de modo de ir mejorando su performance basándose en las acciones que éste ejecuta sobre dicho ambiente. Normalmente, el agente comienza con un conjunto de estrategias o reglas para interactuar con el ambiente. Al observar dicho ambiente, toma acciones particulares basándose en las reglas o políticas y observando el estado actual del ambiente. De acuerdo a la acción que tomó, el agente obtiene una recompensa o una penalización. Este es un proceso iterativo en el que el algoritmo irá aprendiendo y modificando su estrategia de ser necesario, de modo de obtener la recompensa deseada.

5.10. Categorización de los métodos de aprendizaje

Es posible categorizar los métodos de aprendizaje de acuerdo al tipo de salida que se desea obtener de los algoritmos de *machine learning*.

1. **Clasificación**: Los modelos de clasificación se encuentran dentro de los métodos de aprendizaje supervisado, en donde el objetivo principal es predecir una categoría o etiqueta para cada uno de los datos de entrada basándose en lo que el modelo ha aprendido en la etapa de entrenamiento. Estas etiquetas de salida también son conocidas como clases o etiquetas de clase, las cuales por naturaleza son categóricas, por lo que no poseen orden y son discretas. Por lo tanto, cada etiqueta de salida pertenecerá a una clase o categoría discreta específica. Existe una amplia variedad de algoritmos que pertenecen a esta familia, pero quizás los más importantes son:

   - Modelos lineales como por ejemplo *Logistic Regression* (regresión logística), Naïve Bayes, y *Support Vector Machines*
   - Modelos no pramétricos como los k vecinos más cercanos (*k-nearest neighbors*).
   - Métodos basados en árboles como árboles de decisión (*Decision Trees*).
   - Métodos de ensamble como *Random Forest* y *Gradient Boosted Machines* (*boosting*).
   - Redes neuronales.

   Los modelos de clasificación también pueden ser caracterizados por la clase a la que pertenece el dato de salida y la cantidad.

- **Clasificación binaria**: Cuando lo que se tiene son dos categorías para diferenciar, entonces el problema es de clasificación binaria. Un ejemplo de clasificación binaria podría ser el "Problema de clasificación de emails". En este problema lo que se desea es distinguir entre dos categorías: "Es Spam" o "No es Spam".
- **Clasificación multi-clases**: Se considera una extensión al problema de clasificación binario. En este caso se tienen más de dos categorías o clases al que el dato puede pertenecer. Por ejemplo, un problema de clasificación de multi-clases es determinar la categoría de dígitos del cero al nueve que han sido escritos a mano. Con lo cual, lo que se tiene es un problema de clasificación de diez clases.
- **Clasificación multi-etiqueta**: Estos problemas de clasificación normalmente involucran datos que pueden pertenecer a más de una categoría o poseer más de una etiqueta o *label*. Un ejemplo es la predicción de la categoría de artículos de novedades que pueden tener múltiples etiquetas, como política, ciencia, religión, etc.

2. **Regresión**: Al igual que los modelos de clasificación, éstos pertenecen a la categoría de algoritmos de aprendizaje supervisados, pero en lugar de predecir un valor discreto, estos modelos predicen valores reales (continuos). Los métodos basados en regresiones son entrenados a partir de un conjunto de observaciones, cada una conformada por un conjunto de variables independientes y un valor continuo como dependiente. Así el modelo hará uso de estos valores para aprender las relaciones que existen entre los *features* y su *target*. A partir de este conocimiento el modelo sabrá como predecir nuevos valores continuos cuando le sean suministradas nuevas observaciones, no vista anteriormente. Los modelos de regresiones más importantes son:

   - **Regresión lineal simple (*Simple Linear Regression*)**: En este tipo de modelos sólo se tiene una variable independiente y una dependiente. La variable dependiente es un valor real que normalmente sigue una distribución normal. En los modelos de regresión, se asume que existe una relación lineal entre la variable independiente y la dependiente.
   - **Regresión lineal múltiple (*Multiple Linear Regression*)**: Es considerada una extensión del modelo de regresión lineal simple, donde se incluye más de una variable independiente. Al igual que el modelo lineal simple, la predicción es un valor real que sigue una distribución normal.
   - **Regresión no lineal (*No Linear Regression*)**: Un modelo de regresión en el cual los coeficientes no son lineales, puede ser considerado un modelo de regresión no lineal. Por ejemplo, considérese $Y = \beta_0 + (\log \beta_1)x^2 + \varepsilon$. Estos modelos son difíciles de aprender, por lo que no son muy utilizados.

3. ***Clustering***: El clustering pertenece a los modelos de algoritmos no supervisados, cuyo proceso consiste en agrupar datos "similares" que no hayan sido previamente etiquetados o clasificados. Las salidas de este tipo de modelos son grupos de datos segregados que son similares entre sí, pero diferentes a los miembros de los demás grupos. La mayor diferencia que existe con los modelos supervisados es que aquí no se tienen los datos previamente clasificados para entrenar al modelo, por lo que se utiliza todo el conjunto de datos como entrada. Los modelos de *clustering* pueden ser de diferentes tipos, de acuerdo con sus metodologías o principios:

   - ***Clustering* basado en partición**: El método de *clustering* basado en partición definirá una noción de similaridad. Esta similaridad es una característica que se deriva de los atributos de entrada luego de aplicadas funciones matemáticas. Luego, una vez encontradas estas similitudes, se agrupan los datos que son similares en un solo grupo y separados de aquellos que son diferentes. Los modelos de *clustering* basados en partición, en general son desarrollados utilizando técnicas recursivas para clasificarlos. Por ejemplo, se comienza con una porción arbitraria de los datos y, basados en alguna medida de similitud, se continúa reasignando datos hasta que se llegue a un punto estable según algún criterio. Ejemplos de estas técnicas son: *K-means*, *K-medoids*, *CLARANS*, etc.
   - ***Clustering* jerárquico**: Este tipo de modelos se diferencian del *clustering* basado en partición por la manera en la que son desarrollados y por cómo trabajan. Dentro del paradigma del *clustering* jerárquico, se comienza con, o bien todos los datos en un solo grupo (*divisive clustering*), o todos los datos en diferentes grupos (aglomerativo). Según el punto de entrada elegido, se continúa dividiendo el gran grupo en grupos más pequeños o *clusters* basados en algún criterio de similitud, o bien se puede continuar juntando diferentes grupos o *clusters* en grupos más grandes basados en el mismo criterio. El proceso concluye cuando se llega a una condición preestablecida.
   - ***Clustering* basado en densidad**: Ambos métodos antes mencionados son fuertemente dependientes de la noción de distancia. Lo que conduce a estos algoritmos a encontrar *clusters* de datos de forma esférica. Esto puede ser un problema si los datos se encuentran ubicados arbitrariamente. Esta limitación podría ser resuelta si en lugar de tener en cuenta el concepto de distancia, se utilizara el concepto de "densidad" de los datos para desarrollar el modelo. Entonces, la metodología para encontrar puntos ya no consiste en encontrar puntos en la próximos a uno en particular, sino más bien a determinar áreas que contengan puntos. Este tipo de modelos no resultan simples de interpretar como los de métricas de distancia,

pero ayudan con *clusters* que no son necesariamente de forma esférica. Ejemplos de estos modelos son *DBSCAN* y *OPTICS*.

5.11. Resumen

Se comenzó el capítulo definiendo *machine learning* y la razón de su surgimiento. También se mencionó cómo se compone el conjunto de datos que se utiliza para dichos algoritmos. Seguidamente, se abordaron conceptos relacionados a los tipos de estimación que se pueden presentar, los métodos que existen, la evaluación en la precisión de un modelo, y el balance entre las medidas sesgo y varianza. Por último se estudiaron cuáles son los distintos métodos de aprendizaje que pueden encontrarse y cómo se encuentran categorizados.

En el capítulo que se encuentra a continuación se abordará en mayor detalle los métodos y técnicas pertenecientes a algunos de los diferentes modelos de clasificación existentes, y cuáles son las métricas que se utilizan para su evaluación.

CAPÍTULO 6

Modelos de Clasificación

Desde una perspectiva del *machine learning*, tanto el problema de la detección de *malware*, como el de identificación de familias para cada una de las muestras, pueden ser considerados problemas de clasificación. En el caso de la detección de *malware*, donde lo que se intenta identificar es si una muestra es, en efecto, un programa malicioso, la clasificación es binaria: la muestra es o no es un *malware*. En el caso de clasificación de familias, el problema es multi-clase, dado que debe determinarse a cuál de las nueve familias pertenece la muestra.

A continuación se dará un marco teórico a los métodos y técnicas pertenecientes a esta categoría de modelos, algunas de las cuales serán utilizados en esta investigación.

6.1. *Logistic Regression*

Logistic Regression es un algoritmo que puede ser utilizado tanto para resolver problemas de regresión como clasificación. *Logistic Regression*, también conocido como *Logic Regression*, es comunmente utilizado para estimar la probabilidad que una instancia pertenezca o no a una clase en particular. Si la probabilidad estimada es mayor al 50 % entonces el modelo predice que la instancia pertenece a la clase (llamada clase positiva, con etiqueta ”1”), o predice que la clase no pertenece (en cuyo caso la clase será negativa, con etiqueta ”0”). Esto es lo que la convierte un clasificador binario.

Es muy simple de utilizar y comprender, y resulta muy efectivo para problemas en los cuales el conjunto de variables de entrada son bien conocidas y además se encuentran fuertemente correlacionadas con las salidas. Aunque no resulta tan efectivo cuando las variables de entrada no son bien conocidas, o cuando las relaciones entre dichas variables son muy complejas.

6.2. *k-Nearest Neighbors*

El algoritmo de k-Nearest Neighbors es quizás uno de los más sencillos de implementar. Dado un conjunto de entrenamiento de datos etiquetados y una función de distancia, *k-NN* clasifica un punto `X` basándose en los `k` elementos de dicho conjunto que se encuentren más cerca de X.

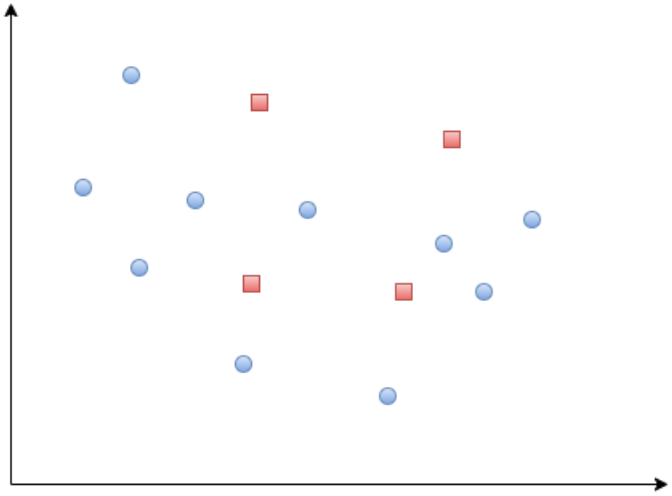

FIGURA 6.1: Conjunto de datos etiquetado

Por ejemplo, si se considera el gráfico de la Figura 6.1, dicho conjunto de entrenamiento consta de diez elementos de tipo círculo azul, y cuatro de tipo cuadrados rojos.

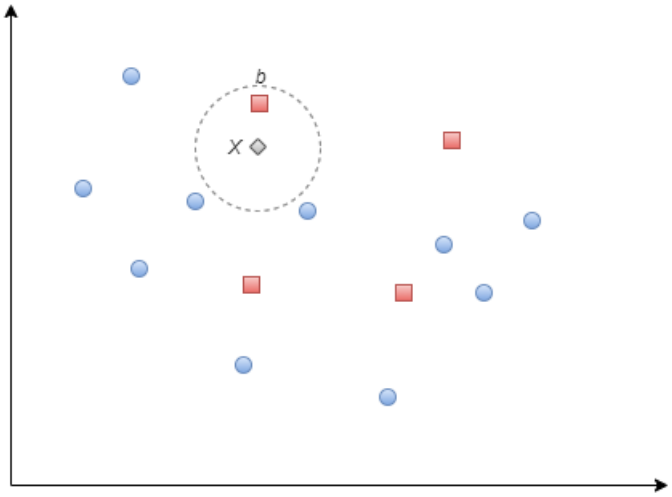

FIGURA 6.2: (a) 1-*nearest neighbor* (1-NN)

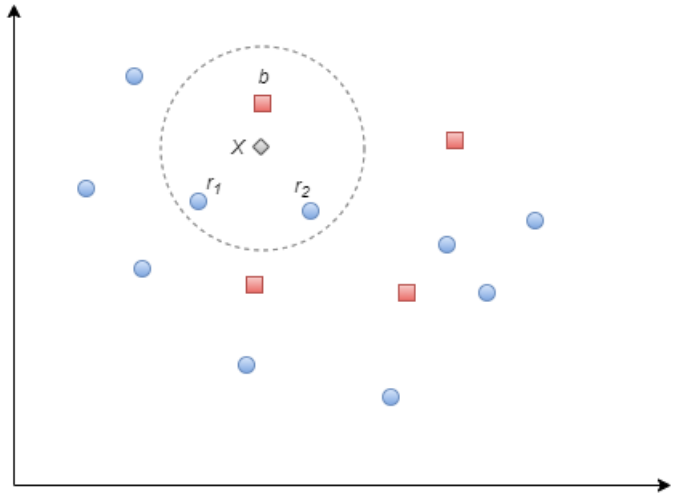

FIGURA 6.3: (b) 3-*nearest neighbor* (3-NN)

Si se quisiera clasificar el "diamante gris" que es etiquetado como X en la Figura 6.2 utilizando 1-NN (donde k = 1), dado que, el punto más cercano a X es el cuadrado rojo etiquetado como b, X sería clasificado como un cuadrado rojo. Por otro lado, si se quisiera clasificar X utilizando 3-NN, como se muestra en la Figura 6.3, se puede observar que existen tres puntos cercanos a X (con respecto a la distancia de Euclides), un cuadrado rojo etiquetado como b, y dos círculos azules etiquetados como r_1 y r_2 respectivamente. Dado que la mayoría de los puntos más cercanos a X son círculos rojos, éste sería clasificado del mismo tipo.

El número de vecinos más cercanos (k) y la medida de distancia son componentes clave para el algoritmo de *k-Nearest Neighbors*. Un valor pequeño de k resultará en una baja precisión, sobre todo en conjuntos de datos con mucho ruido, dado que cada instancia del conjunto de entrenamiento tiene un alto peso durante el proceso de decisión. Un valor grande de k disminuye la performance del algoritmo. Además, si el valor es muy grande, el modelo puede hacer un sobreajuste (*overfitting*), dificultando la separación entre clases, lo cual también resultaría en menos precisión. Una buena regla es utilizar un k menor a la raíz cuadrada de n, siendo n el número total de patrones de entrenamiento.

Existen diferentes métricas para el cálculo de la distancia entre vecinos más cercanos, siendo las más conocidas la distancia Hamming, distancia Manhattan y distancia Minkowsky. La distancia de Euclides es el método más utilizado cuando se trata de variables continuas. En donde, si se toman dos observaciones en un espacio n-dimensional $x_1 = (x_{11}, ..., x_{1n})$ y $x_2 = (x_{21}, ..., x_{2n})$, la distancia de Euclides entre estos dos puntos está dada por:

$$dist(x_1, x_2) = \Big(\sum_{i=1}^{n}(x_{1i} - x_{2i})^2\Big)^{0,5} \tag{6.1}$$

La distancia de Euclides funciona muy bien para problemas en donde los atributos son del mismo tipo. Para atributos de diferente tipo, es recomendable utilizar la distancia Manhattan.

Si bien su facilidad y sencillez de implementar son un aspecto positivo en este tipo de algoritmos, no resultan aptos para todos los casos. Si el conjunto de datos se encuentra desigualmente distribuido, el algoritmo de *k-Nearest Neighbors* no tendrá una buena perfomance. Esto se debe a que, si una clase domina ampliamente a las demás, es muy probable tener más vecinos de esa clase debido a su gran tamaño, lo que llevará a tener predicciones incorrectas.

6.3. *Naïve Bayes*

Naive Bayes es un algoritmo de clasificación de *machine learning* que utiliza el teorema de Bayes. El mismo puede ser utilizado tanto para problemas de clasificación binario como para multi-clases. El objetivo principal de este método es tratar cada atributo independientemente, evaluando la probabilidad de cada uno de ellos sin tener en cuenta ninguna correlación, y hacer la predicción basándose en el teorema de Bayes. Este es el motivo por el cual es llamado "naïve" (ingenuo, en español), ya que en problemas del mundo real siempre existe algún tipo de correlación entre los atributos.

Para comprender mejor el algoritmo de Bayes, es necesario introducir algunos conceptos:

- **Probabilidad de clase**: Es la probabilidad de una clase en el *dataset* o conjunto de datos. En otras palabras, si se selecciona de manera aleatoria un elemento del *dataset*, es la probabilidad de pertenecer a cierta clase.
- **Probabilidad condicional**: Es la probabilidad de un valor de un atributo dada una clase.

En donde, la probabilidad de clase es calculada simplemente como el número de observaciones en la clase, dividido el número total de observaciones.

$$P(C) = \frac{\text{cantidad de instancias en C}}{\text{cantidad de instancias total}} \tag{6.2}$$

La probabilidad condicional es calculada como la frecuencia de cada valor de atributo, dividido por la frecuencia de instancias en esa clase.

$$P(V|C) = \frac{\text{cantidad de instancias con V y C}}{\text{cantidad de instancias con V}} \tag{6.3}$$

Dadas las probabilidades se podrá ahora calcular la probabilidad de una instancia perteneciente a una clase y de esta manera tomar decisiones utilizando el teorema de Bayes:

$$P(A|B) = \frac{P(A|B)P(A)}{P(B)} \tag{6.4}$$

Las probabilidades de cada ítem pertenecientes a todas las clases son comparadas, y la clase con la más alta probabilidad es seleccionada como resultado.

La ventaja de utilizar este método es su simplicidad y facilidad de entendimiento. Además, posee buena performance en conjunto de datos con atributos irrelevantes, ya que las probabilidades de que estas contribuyan a las salidas son bajas. Por lo que las mismas no son tomadas en cuenta cuando se realizan las predicciones. Más aún, este algoritmo usualmente resulta en una buena performance en términos del uso de recursos, dado que sólo necesita calcular las probabilidades de los atributos y clases, por lo que no hay necesidad de encontrar ningún coeficiente como en otros algoritmos.

6.4. *Support Vector Machines*

Support Vector Machines (SVM) es un algoritmo de *machine learning* que es utilizado generalmente para resolver problemas de clasificación. El objetivo principal de un SVM es encontrar un hiperplano que separe las clases de la mejor manera posible. Donde un hiperplano es definido como un subespacio de una dimensión menos que el espacio en el cual se está trabajando. Por ejemplo, si los datos con los cuales se están trabajando viven en un espacio bidimensional, un hiperplano es simplemente una línea. Si el hiperplano existe, se dice que los datos son **linealmente separables**.

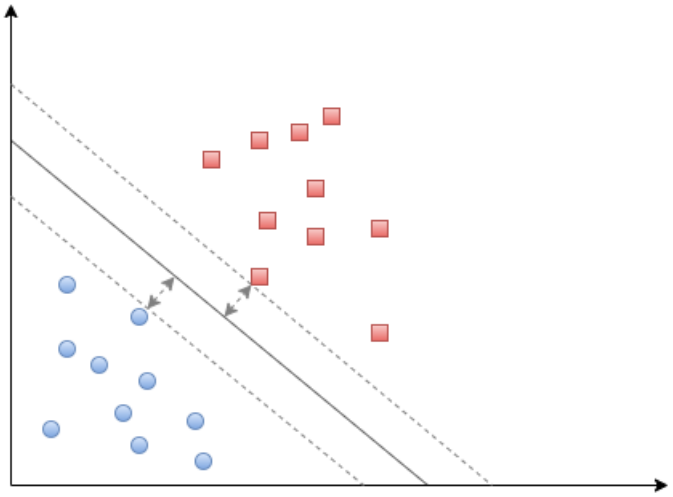

FIGURA 6.4: Maximización del margen

A la hora de elegir un hiperplano para el SVM, lo que se busca es uno que maximice el margen, donde el margen se define como la distancia mínima entre el hiperplano y cualquier elemento del *training set*. Si se observa la Figura 6.4, las flechas son las que representan el margen.

Los SVM generalmente resultan en una muy buena precisión, incluso con conjunto de datos pequeños. Adicionalmente, si el *dataset* es pequeño, el modelo se ejecutará rápidamente, pero los tiempos de ejecución se disparan conforme el tamaño del conjunto de datos aumenta.

6.5. *Decision Trees*

Al igual que los *Support Vector Machines*, los *Decision Trees* (árboles de decisión) son algoritmos de *machine learning* versátiles, que pueden ejecutar tanto tareas de regresión como de clasificación, e incluso obtener múltiples salidas. Son algoritmos muy poderosos, capaces de ajustarse a complejos conjuntos de datos.

Un *Decision Tree* se construye dividiendo de manera recursiva el conjunto de datos en secuencias de sub-conjuntos basado en preguntas del estilo si-entonces (*if-then*). El conjunto de entrenamiento consiste de pares (x, y) donde $x \in I\!R^d$, por lo que d se corresponde con el número de atributos disponibles y en donde y es la correspondiente etiqueta. El método de aprendizaje dividirá el conjunto de entrenamiento en grupos basándose en x mientras intenta mantener las asignaciones en cada grupo tan uniformes como sea posible. Para lograrlo, el método de aprendizaje debe elegir un atributo y un umbral asociado a dicho atributo sobre el cual dividirá los datos. El proceso de entrenamiento continuará hasta encontrar una condición de fin.

Los *Decision Trees* son fáciles de construir, permiten la creación de complejos procesos de decisión y sus resultados son muy intuitivos de interpretar. Pero pueden fácilmente producir *overfitting*, expandiendo el árbol con ramas que reflejan los *outliers* en el conjunto de datos. Una manera de tratar el sobreajuste es "podar.[el] modelo, ya sea evitando el crecimiento de ramas superfluas (*pre-pruning*), o removiéndolas luego de que el árbol haya crecido (*post-pruning*).

6.6. Métodos de ensamble

Para poder resolver un problema, los métodos de ensamblado entrenan múltiples modelos de aprendizaje. A diferencia de los métodos típicos, que producen un único modelo luego del entrenamiento, el objetivo es construir un conjunto de modelos apenas mejores que "tirar la moneda", cuyos resultados puedan ser *combinados* de acuerdo a un criterio para alcanzar una performance casi perfecta.

De este modo, un *ensamble* está compuesto por modelos de aprendizaje *básicos* o *débiles*, que implementan un algoritmo tal como un árbol de decisión o una red neuronal. Si todos los modelos utilizan el mismo algoritmo básico, se dice que el *ensamble* es **homogéneo**, mientras que, si distintos modelos ejecutan algoritmos de distinto tipo, el *ensamble* es conocido como **heterogéneo**.

Existen dos principales metodologías de métodos de ensamble:

- *Bagging*. Dentro de esta metodología el principal objetivo es construir varios estimadores independientemente y luego promediar sus predicciones. Al utilizar esta técnica lo que se logra es una drática reducción del error por lo que este método siempre buscará crear modelos lo más independientes posible. Un ejemplo de algoritmos que utilizan estos métodos son los *Random Forest*.
- *Boosting*. A diferencia del método *bagging*, el *boosting* construirá estimadores base de manera secuencial con el objetivo de reducir el sesgo del estimador combinado. Su motivación es convertir modelos de aprendizaje débiles en modelos robustos. Ejemplos que utilicen esta técnica son: *AdaBoost*, *Gradient Boosting*, *XGBoost*, entre otros.

6.6.1. *Random Forest*

Random Forest es un algoritmo de aprendizaje supervisado flexible y fácil de usar, capaz de producir muy buenos resultados aún sin haber realizado el ajuste de sus parámetros. El "*forest*"(bosque) que construye, es un ensamble de varios *Decision Trees*, que normalmente es entrenado utilizando el método *bagging*.

El método *bagging* (*Boostrap Aggregation*) es una combinación de varios modelos de aprendizaje cuyo objetivo consiste en reducir la alta varianza del *Decision Tree*. Su funcionamiento consiste básicamente en crear varios subconjuntos de datos del conjunto de entrenamiento original (*training set*) elegidos de manera aleatoria con reemplazo. Luego, cada subconjunto se utiliza para entrenar los árboles de decisión, dando como resultado el ensamble de diferentes modelos. El promedio de las predicciones de los diferentes árboles es lo que se utiliza, que resulta más robusto que el de un sólo árbol de decisión.

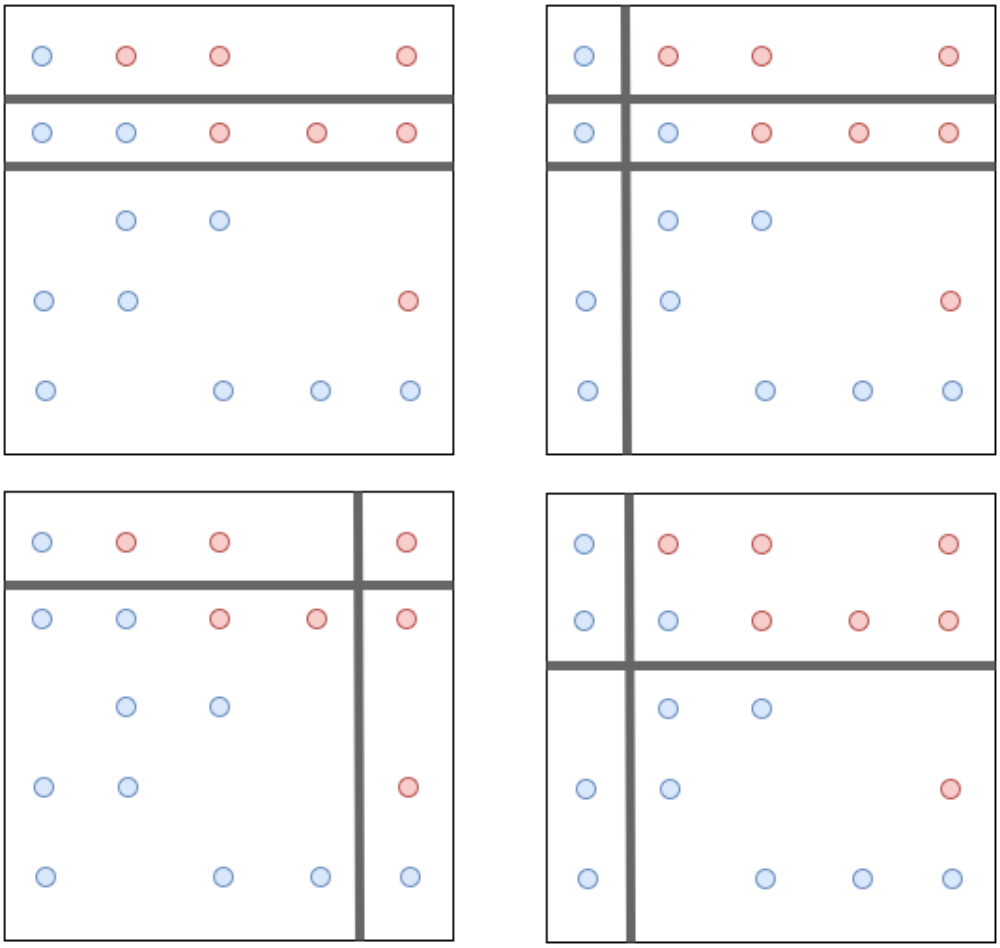

FIGURA 6.5: Constitución de los árboles de decición de un *random forest* y cómo particiona el conjunto de entrenamiento.

La principal ventaja de los clasificadores *Random Forest* es que requieren muy poco ajuste (*tuning*), y aún así son capaces de proveer una forma de alcanzar un buen equilibrio entre el sesgo y la varianza utilizado el promedio y la aleatoriedad. Además, son rápidos y fáciles de entrenar en paralelo, y eficaces al momento de predecir. A diferencia de los sencillos árboles de decisión, los *random forest* no son intuitivos y pueden resultar mucho más difíciles de interpretar.

6.6.2. *XGBoost*

XGBoost[1], cuyo nombre proviene de *e**X**treme **G**radient **B**oosting*, es una implementación del algoritmo de aprendizaje automático *Gradient Boosting* desarrollado para funcionar de manera altamente eficiente, flexible y portable.

Gradient Boosting es una técnica de *machine learning* para resolver problemas de regresión y clasificación, el cual produce un modelo de ensamble débil, normalemente un árbol de decisión. Este modelo es construido de misma manera que lo hacen los

[1] https://xgboost.readthedocs.io/en/latest/

algoritmos de *boosting*, en donde nuevos modelos son agregados a los ya existentes para corregir los errores. Estos modelos son incorporados de manera secuencial hasta que se detecte que no hay más mejoras por realizar. La técnica *Gradient Boosting* utiliza el descenso del gradiente para minimizar la pérdida cuando agrega nuevos modelos, de ahí su nombre.

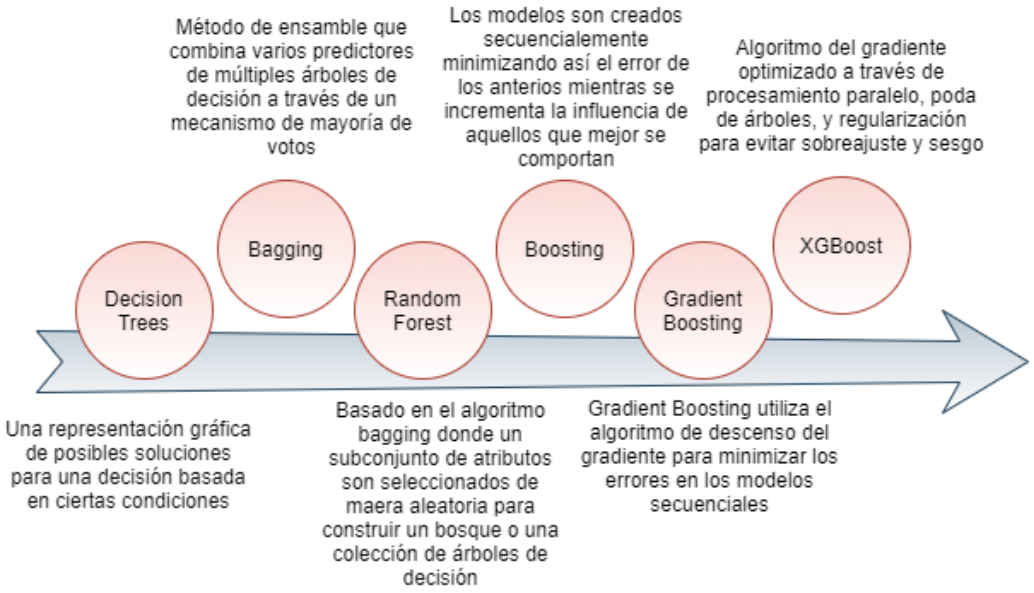

FIGURA 6.6: Evolución de XGBoost desde un Decision Trees.

6.7. Redes Neuronales Artificiales y Redes Neuronales Profundas

Tanto las Redes Neuronales Artificiales (*Artificial Neural Network*), como las Redes Neuronales Profundas (*Deep Neural Network*) pueden ser utilizadas para realizar tareas de clasificación y obtener muy buenos resultados. Si bien pueden resultar más complejas de programar, existen muchas herramientas, o *frameworks* como TensorFlow[2], desarrolladas con el fin de ayudar al programador con dicha tarea.

Las Redes Neuronales han tomado su inspiración en el proceso de aprendizaje que ocurre en el cerebro humano. Se componen de una red de funciones llamadas parámetros, que le permitirán a la red aprender, los cuales, a su vez, se pueden ajustar (*tuning*) a si mismos mediante el análisis de los datos. Cada uno de estos parámetros, también conocidos como neuronas, es una funcioón que produce una salida luego de haber recibido una o más entradas. Luego, estas salidas se pasarán a la siguiente capa de neuronas, la cual las utilizará como entrada en su función y generará su propia salida. Estas nuevas salidas se enviarán a la siguiente capa y así el proceso continuará sucesivamente hasta haber considerado todas las neuronas que

[2] https://www.tensorflow.org/

conforman la red y las neuronas terminales hayan recibido su entrada. Las salidas de estas neuronas terminales será el resultado final del modelo.

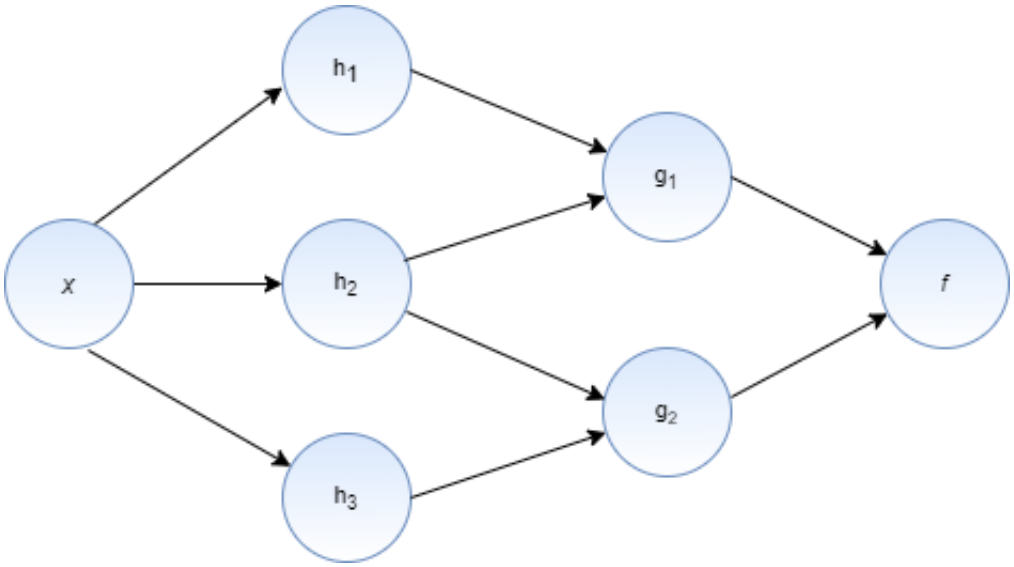

FIGURA 6.7: Representación visual de una Red Neuronal simple

Las Redes Neuronales pueden ser muy efectivas para problemas de alta dimensionalidad, son capaces de lidiar con complejas relaciones entre variables, conjuntos de categorías no exhaustivas y complejas funciones relacionadas a la entrada y salida de variables. Cuentan con poderosas opciones de *tuning* para evitar el *overfitting* y *underfitting*.

Si bien estas redes pueden resultar muy robustas y poderosas, también pueden resultar complejas y difíciles de implementar (aunque esto puede resolverse fácilmente si se utiliza un *framework*). Pueden no ser intuitivas y requieren de una mano experta para ser ajustadas. En algunos casos pueden requerir de grandes conjuntos de datos para ser efectivas.

6.8. Evaluación de modelos de clasificación

6.8.1. Matriz de confusión

La matriz de confusión es una de las formas más conocidas de evaluar modelos de clasificación. Aunque la matriz en sí misma no es una métrica, su representación puede ser utilizada para definir una variedad de métricas, que pueden resultar muy importantes dependiendo del escenario. La matriz de confusión puede ser utilizada tanto para modelos de clasificación binaria como así también de multi-clases.

La matriz de confusión se crea a partir de la comparación de la etiqueta que predijo con la etiqueta actual que posee la muestra. Este proceso se repite para todos los datos del *dataset*, y los resultados son representados en una matriz de dimensión

dos. En donde, la primera columna contiene la suma total de verdaderos negativos (resultados que el modelo predijo eran negativos y en efecto lo son), y falsos negativos (resultados que el modelo predijo eran negativos, pero no lo eran); y una segunda columna que contiene la suma total de los falsos positivos (resultados que el modelo predijo eran positivos, pero no lo eran) y verdaderos positivos (resultados que el modelo predijo eran positivos y eran positivos).

Como se dijo anteriormente, la matriz en sí no es una métrica, pero a partir de los resultados que ella arroja se pueden calcular ciertas métricas que resultan muy útiles. Se considera TP (verdaderos positivos), TN (verdaderos negativos), FP (falsos positivos), FN (falsos negativos), entonces:

- ***Accuracy***: Es una de las métricas de performance más populares dentro de los modelos de clasificación. Define la proporción de predicciones correctas realizadas sobre el modelo.

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{6.5}$$

- ***Precision***: Es otra métrica que se deriva de la matriz de confusión. Se define como el número de predicciones hechas que son realmente correctas sobre el total de predicciones positivas.

$$Precision = \frac{TP}{TP + FP} \tag{6.6}$$

- ***Recall***: Es una medida del modelo que identifica el porcentaje de datos relevantes. Se define como el número de instancias de la clase positiva que fueron correctamente predichas.

$$Recall = \frac{FP}{TP + FN} \tag{6.7}$$

- ***F1 Score***: En muchos casos lo que se quiere es una optimización balanceada entre la precisión y la sensibilidad.

$$\text{F1 Score} = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{6.8}$$

6.8.2. *Receiver Operating Characteristic Curve*

ROC por sus siglas en inglés, es un concepto que proviene originalmente del uso de Radares. Este concepto se puede extender a modelos de clasificación binarios y multi-clases. Y puede ser interpretada como la efectividad con la cual el modelo puede distinguir entre una señal verdadera y el ruido en los datos.

La curva ROC se crea dibujando la fracción de verdaderos positivos versus la fracción de falsos positivos. Es mayormente aplicable en clasificadores de *scoring*. Clasificadores de *scoring* son un tipo de clasificador el cual retorna un valor de probabilidad o *score* por cada clase de *label*. La curva ROC funciona de la siguiente manera:

1. Se ordenan las salidas del clasificador ordenados por su *score* (o la probabilidad de pertenecer a la clase positiva).
2. Se comienza en la coordenada $(0, 0)$.
3. Por cada valor x de la lista ordenada se pregunta:
   a) Si x es positiva, se mueve $\frac{1}{pos}$ hacia arriba.
   b) Si x es negativa, se mueve $\frac{1}{neg}$ hacia la derecha.

   Donde *pos* y *neg* son las fracciones de positivos y negativos respectivamente. Luego se traza una línea diagonal, lo que indicará si nuestra curva ROC se encuentra por encima quiere decir que el modelo de clasificación es mejor que el promedio.

6.9. Resumen

En el presente capítulo se explicó en mayor detalle algunos de los métodos existentes de *machine learning* para clasificación. Se comenzó explicando el *Logistic Regression*, un método simple que puede ser utilizado tanto para regresión como para clasificación binaria. Luego se estudiaron algoritmos como *K-Neares Neighbors*, *Naive Bayes* y *Support Vector Machines*, todos estos métodos resultan simples de implementar y de comprender. A continuación, se abordó el algoritmo *Decision Tree*, que, al igual que los *Support Vector Machines* son algoritmos versátiles que pueden ejecutar tareas tanto de regresión como de clasificación. También se explicaron algunos métodos de ensamble como el *Random Forest* y la implementación del algoritmo de *Gradient Boosting*, *XGBoost*. Por último, se describieron las Redes Neuronales Artificiales y las Redes Neuronales Profundas, las cuales también pueden ser utilizadas para taras de clasificación.

Como paso final, se mencionaron y describieron cuáles son las técnicas más utilizadas para evaluar modelos de clasificación, entre las cuales se destacan la matriz de confusión y la curva ROC.

Parte II

Implementación

CAPÍTULO 7

Data Mining: Generación del *dataset*

Este capítulo se enfoca en el estudio de las decisiones de diseño e implementación de técnicas utilizadas para el análisis del conjunto de datos disponible. De este modo, se comenzará describiendo la estructura y formato de dichos datos crudos originales, para luego pasar a enumerar aquellos atributos que han sido considerados más relevantes. Una vez identificados dichos *features*, se mencionarán y describirán los métodos y técnicas utilizadas para extraerlos. Esta información obtenida será la que luego conformará el *dataset* que se utilizará para los algoritmos de clasificación.

7.1. Conjunto inicial de datos

El conjunto inicial de datos con el que se realizó el experimento fue obtenido del sitio *Kaggle*[1]. El mismo se compone de un conjunto de archivos de tipo `ASM` y su correspondiente archivo de tipo `BYTES`, equivalentes a aproximadamente once mil *malwares*. Cada uno de estos archivos pertenece a una de nueve clases o familias de virus distinta.

7.1.1. Archivos `ASM`

Los archivos `ASM` son programas escritos en lenguaje ensamblador, almacenados bajo la extensión `.asm`, el cual guarda una relación cercana con las instrucciones en código máquina.

[1] https://www.kaggle.com/c/malware-classification/

```
.text:004010AF 8B EC mov     ebp, esp
.text:004010B1 83 EC 44 sub     esp, 44h
.text:004010B4 8D 15 C4 91 57 00        lea     edx, unk_5791C4
.text:004010BA 81 FA BF 1D 00 00        cmp     edx, 1DBFh
.text:004010C0 72 ED jb      short loc_4010AF
.text:004010C2 52 push    edx
.text:004010C3 FF B2 E8 02 00 00        push    dword ptr [edx+2E8h]
.text:004010C9 57 push    edi
.text:004010CA FF 72 50 push    dword ptr [edx+50h]
.text:004010CD E8 42 11 00 00 call    sub_402214
.text:004010D2 83 C4 0C add     esp, 0Ch
.text:004010D5 5A pop     edx
.text:004010D6 52 push    edx
.text:004010D7 BA 00 00 00 00 mov     edx, 0
.text:004010DC 52 push    edx
.text:004010DD E8 07 00 00 00 call    sub_4010E9
.text:004010E2 FF 35 24 30 40 00        push    ds:GetPriorityClass
.text:004010E8 C3 retn
.text:004010E8 sub_4010AC      endp ; sp-analysis failed
```

FIGURA 7.1: Extracto del archivo `0AguvpOCcaf2myVDYFGb.asm`

Nota: Los encabezados PE (*Portable Executable*) han sido eliminados por cuestiones de seguridad. Estos encabezados poseen una estructura particular formados por una serie de secciones que le indican al *dynamic linker* cómo deberá mapear el archivo en memoria.

7.1.2. Archivos `BYTES`

Los archivos BYTES son archivos que contienen el código de máquina ejecutable en representación hexadecimal. Estos archivos se almacenan bajo la extensión `.bytes`.

```
10001000 FF 25 F8 80 00 10 FF 25 F4 80 00 10 FF 25 D8 80
10001010 00 10 FF 25 E8 80 00 10 55 8B EC 83 EC 10 A1 00
10001020 90 00 10 83 65 F8 00 83 65 FC 00 53 57 BF 4E E6
10001030 40 BB 3B C7 BB 00 00 FF FF 74 0D 85 C3 74 09 F7
10001040 D0 A3 04 90 00 10 EB 60 56 8D 45 F8 50 FF 15 6C
10001050 80 00 10 8B 75 FC 33 75 F8 FF 15 70 80 00 10 33
10001060 F0 FF 15 18 80 00 10 33 F0 FF 15 28 80 00 10 33
10001070 F0 8D 45 F0 50 FF 15 74 80 00 10 8B 45 F4 33 45
10001080 F0 33 F0 3B F7 75 07 BE 4F E6 40 BB EB 0B 85 F3
10001090 75 07 8B C6 C1 E0 10 0B F0 89 35 00 90 00 10 F7
100010A0 D6 89 35 04 90 00 10 5E 5F 5B C9 C3 56 68 80 00
100010B0 00 00 FF 15 9C 80 00 10 8B F0 56 FF 15 98 80 00
100010C0 10 85 F6 59 59 A3 5C E4 00 10 A3 58 E4 00 10 75
100010D0 05 33 C0 40 5E C3 83 26 00 E8 D8 03 00 00 68 DA
100010E0 14 00 10 E8 BC 03 00 00 C7 04 24 F3 13 00 10 E8
```

FIGURA 7.2: Extracto del archivo `0AnoOZDNbPXIr2MRBSCJ.bytes`

7.1.3. Las familias de Malware

Como se mencionó previamente, cada uno de los archivos *malware* disponibles pertenecen a una de nueve familias (o clases) de *malware*. Esta información se encuentra disponible en un archivo, el cual incluye para cada muestra la clase la que pertenece.

Entre las nueve familias se encuentran:

1. **Ramnit**, perteneciente a la familia de tipo troyanos, infecta archivos ejecutables tales como .exe y .html, y puede esparcirse fácilmente a través de dispositivos removibles como por ejemplo memorias USB.

2. **Lollipop**, es un tipo de adware o PUP (Potentially Unwanted Program), el cual se caracteriza por instalar toolsbar o mostrar publicidades por medio de ventanas emergentes.

3. **Kelihos_ver3**, de tipo botnet, el cual normalmente se encuentra relacionado con el robo de bitcoins y el envío de spam.

4. **Vundo**, de tipo troyano conocido por utilizar popups y publicidad de programas de antivirus maliciosos.

5. **Simda**, perteneciente a la familia de backdoors, encargado de robar información de usuarios tales como nombres de usuario, contraseñas y certificados.

6. **Tracur**, de la familia de troyanos, y es capaz de redireccionar búsquedas web a sitios con publicidad fraudulenta, además puede descargar y ejecutar archivos, como por ejemplo otros malwares.

7. **Kelihos_ver1**, botnet relacionado al envío de spam masivo.

8. **Obfuscator.ACY**, utiliza técnicas para ocultar decarga de archivos y robo de información.

9. **Gatak**, perteneciente a la familia de los troyanos, el cual se encargará de recopilar información en la PC en la que se está ejecutando para luego enviársela a un *hacker*.

7.2. Análisis y selección de características más relevantes

Con los archivos ya descargados, el siguiente paso consiste en teorizar qué características puede llegar a ser útiles para identificar programa similares. Para esto es necesario estudiar tanto la estructura de cada archivo (tamaño y fisonomía), como su funcionamiento: instrucciones que ejecuta y uso de librerías.

A continuación se presentan aquellas características que fueron consideradas más relevantes.

- **Librerías DLL**. Las `DLLs` (*Dynamic-link Library*) son librerías de enlace dinámico que se cargan bajo demanda durante la ejecución de un programa. Los archivos correspondientes se almacenan bajo la extensión `.dll`. Las referencias a estas librerías pueden resultar de interés bajo el supuesto que el mismo tipo de *malware* hace uso del mismo tipo de `DLLs`.

- **Códigos de operación**. Un código de operación identifica el tipo de instrucción a ejecutar. De forma similar al caso anterior, se presupone que miembros de una misma familia de *malware* ejecutan el mismo tipo de operaciones.

- **Secciones**. El código `.ASM` se encuentra dividido en distintas secciones, como `HEADER`, `.text` y `.rdata`. De este modo, las secciones describen, en términos generales, la estructura de un archivo. Es posible que miembros de una misma familia compartan la misma estructura, por lo que deben ser estudiadas.

- **Códigos de Operación - N Gramas**. Un `N Grama` es una secuencia contigua formada por `n` items de alguna muestra de texto, audio, numérica, o de cualquier otra fuente que se esté investigando. Tomando los códigos de operación, anteriormente mencionados, se desea determinar si miembros de la misma familia ejecutan la misma secuencia de instrucciones (n-gramas de códigos de operación).

- **Tamaños de archivos**. El tamaño del archivo podría ser un dato representativo de la clase a la que pertenece dicho *malware*, ya que archivos de la misma clase podrían tener tamaños similares.
- **Tamaños de archivos comprimidos**. Al igual que el tamaño de archivo, el tamaño del archivo una vez comprimido, y también su *ratio* de compresión, podría resultar en un dato de interés.
- ***Snapshots* de los primeros 1024 *bytes***. Se propone estudiar el primer `kilo byte` de cada archivo con el propósito de determinar si *malware* pertenecientes a una misma familia comparten cierta fisonomía.

7.3. Extracción de los Datos

Una vez que se han identificado aquellos aspectos de los archivos que desean ser estudiados, la siguiente tarea consiste en la elaboración de un conjunto de procesos que permita la extracción de estos datos.

A continuación se describen los distintos procesos que conforman un *pipeline* de tareas de minería de datos que fueron ejecutadas para la construcción de un *dataset* único a partir de miles de muestras de *malware*.

7.3.1. DLLs, Secciones y Códigos de Operación

Partiendo del supuesto que los archivos de una misma familia son similares en cuanto a estructura y comportamiento, se estudian los códigos de operación, sección y uso de `DLLs`, con el fin de determinar la veracidad de este supuesto.

De este modo, para la extracción de estos atributos se diseñó e implementó un *pipeline* de procesos, el cual tiene como objetivo final determinar cuáles son los *features* más relevantes para cada familia, así como también contabilizar las ocurrencias de estos atributos relevantes para cada una de las muestras disponibles. Este *pipeline* puede descomponerse en los siguientes pasos:

1. Calcular la cantidad de ocurrencias de cada atributo para cada una de las muestras.
2. Determinar los atributos más relevantes para cada una de las familias.
3. Crear una lista única de atributos relevantes para el total de familias.
4. Construir una "tablaçon la cantidad de ocurrencias de los atributos más relevantes para todas las muestras del conjunto de datos.

7.3.1.1. Procesamiento de archivos `ASM`

El primer paso en el proceso de minería de estos *features* consiste en tomar los archivos `ASM` y recorrerlos línea por línea en búsqueda de la posible ocurrencia de atributos de interés, los cuales son totalizados. De esta manera, se construye una estructura que contiene, por cada archivo `ASM` disponible, la cantidad de veces que cada *feature* fue encontrado. La figura 7.3 ilustra este primer paso del proceso.

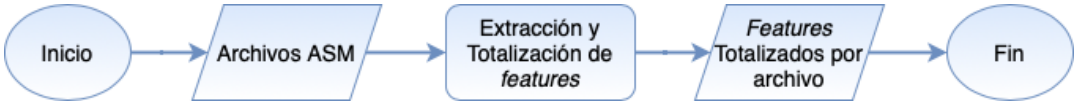

FIGURA 7.3: Procesamiento de Archivos `ASM`

El resultado de este proceso es un diccionario (o mapa) que, por cada muestra, totaliza la cantidad de cada una de las ocurrencias encontradas en el mismo, como se ilustra a continuación:

```
archivo1: {
   seccion1: 200
   seccion2: 200
   seccion5: 200
},
...
archivo5: {
   seccion1: 100
   seccion2: 200
   seccion5: 200
   seccion6: 1500
}
```

Esta estructura es guardada en disco utilizando *MessagePack*[2], el cual provee un formato eficiente de serialización más liviano y veloz que `JSON`. Almacenar la información de esta manera evita tener que convertirla a una representación tabular, la cual puede generar un archivo con redundancia de campos o con mucha dispersión.

Identificación de DLLs

La lógica para la detección de *DLL*s debe ser capaz de detectar la cadena `.dll` sin ser sensible a mayúsculas ni minúsculas. A su vez, debe evitar procesar líneas correspondientes a comentarios, tal como se ilustra en la siguiente imagen.

[2] https://msgpack.org

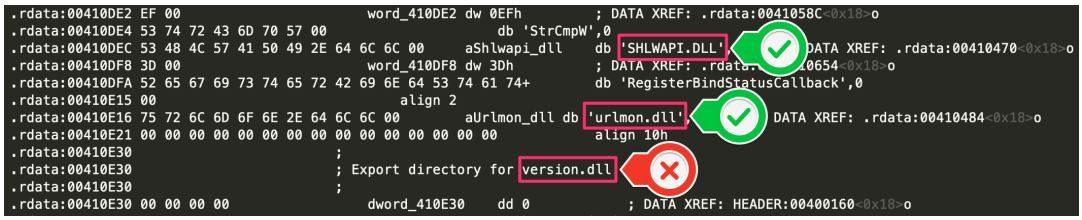

FIGURA 7.4: Extracción de DLLs de Archivos ASM

Identificación de Códigos de Sección

Cada línea comienza con un prefijo que identifica el tipo de sección a la que la línea en cuestión pertenece. Este código siempre es seguido por un dos puntos. De este modo, extraer el código de sección es tan simple como obtener la cadena a la izquierda del primer dos puntos.

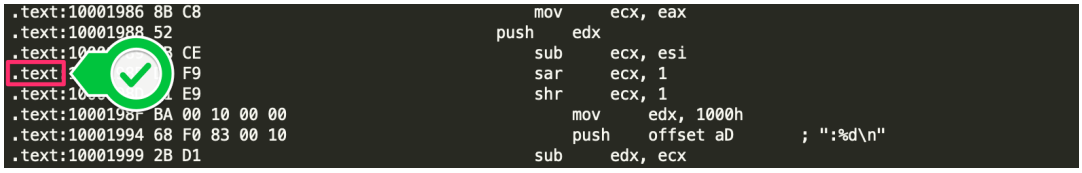

FIGURA 7.5: Extracción de Códigos de Sección de Archivos ASM

Identificación de Códigos de Operación

Se asume como código de operación a la primer palabra de una línea, en tanto y en cuanto no se trate de un comentario.

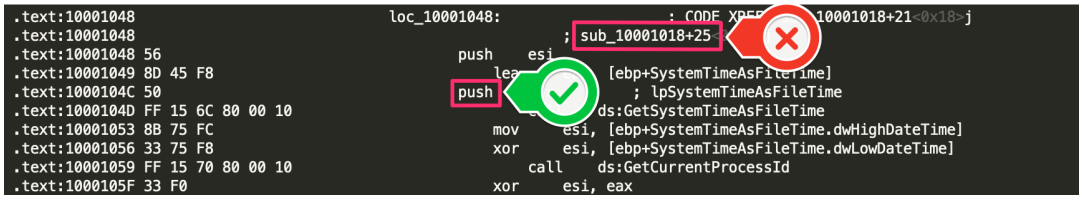

FIGURA 7.6: Extracción de Códigos de Operación de Archivos ASM

7.3.1.2. Totalización de Ocurrencias y Cálculo de Proporciones

Una vez procesados los archivos ASM y obtenido un total de occurrencias de cada *feature* por archivo, se debe determinar cuáles de estos *features* son los que caracterizan realmente a cada una de las familias. Una solución simple a este problema podría ser totalizar las ocurrencias de cada atributo (por familia), para luego quedarse con aquellos que produjeron cuentas más altas. Sin embargo, este enfoque puede llevar a problemas e inconsistencias en presencia de valores anómalos.

Para graficar la situación se propone la siguiente tabla, la cual presenta la cantidad de líneas cada sección encontrada en cada uno de siete archivos de ejemplo.

	seccion1	**seccion2**	**seccion3**	**seccion4**	**seccion5**	**seccion6**
archivo1	200	200	0	0	200	0
archivo2	200	200	0	0	200	0
archivo3	200	200	100	0	100	0
archivo4	200	200	0	1500	100	0
archivo5	100	200	0	0	200	1500
archivo6	200	200	0	0	200	0
archivo7	200	100	0	0	200	0
TOTAL	**1300**	**1300**	**100**	**1500**	**1200**	**1500**

CUADRO 7.1: Ejemplo de totalización de secciones usando sumas de ocurrencias

Con una rápida inspección de la tabla anterior podemos observar que los siete archivos contienen un número elevado de líneas pertenecientes a **seccion1**, **seccion2** y **seccion5**. También se observan valores atípicos para **seccion4** en **archivo4** y **seccion6** en **archivo5**. Suponiendo que el objetivo es obtener las cuatro secciones más relevantes basándonos en la cantidad de ocurrencias, éstas serían **seccion4**, **seccion6**, **seccion1** y **seccion2**. Esto evidencia dos tipos de problemas:

- Sensibilidad a valores atípicos: **seccion4** y **seccion6** aparecen tan sólo en un archivo cada uno, pero en una cantidad extremadamente alta. Estos valores inusuales hacen que sean identificados como los *features* más importantes.
- Omisión de *features* relevantes: como corolario de la sensibilidad a valores atípicos mencionada en el punto anterior, atributos que deberían ser capturados, como **seccion5**, son desplazados y no son identificados como importantes para la correspondiente familia.

Para solucionar estos dos problemas es necesario comprender que lo que hace más relevante a un *feature* no es la **cantidad** de ocurrencias, sino qué **proporción** estas cantidades representan sobre el total de ocurrencias de cada archivo. Así, se puede expresar la tabla 7.1 utilizando proporciones de la siguiente manera.

	seccion1	seccion2	seccion3	seccion4	seccion5	seccion6
archivo1	0,333	0,333	0	0	0,333	0
archivo2	0,333	0,333	0	0	0,333	0
archivo3	0,333	0,333	0,166	0	0,166	0
archivo4	0,1	0,1	0	0,75	0,05	0
archivo5	0,05	0,1	0	0	0,1	0,75
archivo6	0,333	0,333	0	0	0,333	0
archivo7	0,4	0,2	0	0	0,4	0
TOTAL	**1,883**	**1,733**	**0,166**	**0,75**	**1,716**	**0,75**

CUADRO 7.2: Ejemplo de totalización de secciones usando proporciones

Como puede observarse, el uso de proporciones produce resultados que capturan de mejor manera la relevancia de cada atributo.

Habiendo establecido una estrategia mediante la cual se determinarán los atributos más importantes, el siguiente paso en el *pipeline* consiste en consumir el archivo generado en el paso anterior con los *features* totalizados por archivo y convertir estos en proporciones y acumular estos valores por familia.

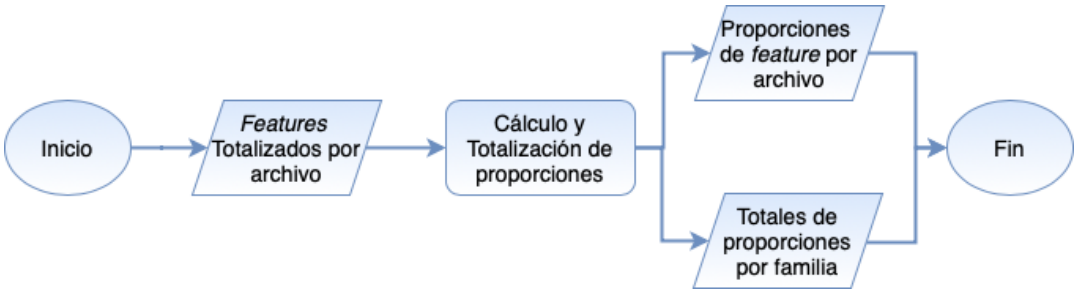

FIGURA 7.7: Cálculo y Totalización de Ocurrencias

De este modo, el proceso genera dos archivos de salida. Uno análogo al resultado del proceso anterior, pero con las cantidades de ocurrencias expresadas en proporciones, similar al contenido del cuadro 7.2. Este archivo será de gran utilidad para el último paso del proceso de extracción de atributos . El segundo archivo contiene una totalización de las proporciones anteriormente mencionadas por cada una de las nueve familias de *malware* que se disponen. Este archivo será utilizado por el siguiente paso del proceso.

7.3.1.3. Determinación de los *features* más relevantes

Una vez que se ha ejecutado el paso anterior, se dispone de las proporciones de ocurrencias de cada *feature* por archivo. De este modo, es posible totalizar estos valores por familia, lo que permite determinar cuáles son los atributos más relevantes

para cada una de las clases de *malware* disponible.

La siguiente figura presenta un diagrama del proceso de obtención de los atributos más relevantes para cada una de las familias de *malware* disponibles.

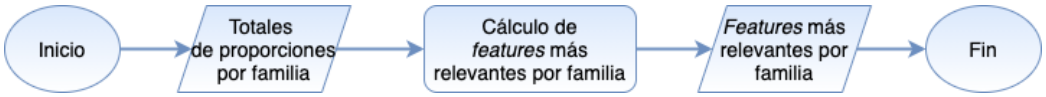

FIGURA 7.8: Determinación de lo *features* más importante para cada familia

Dada la gran cantidad de atributos disponible, es necesario poder reducir este número sin sacrificar demasiada *performance* para los modelos de *machine learning*. Para ello, se debe escoger un *punto de corte* adecuado. Esta tarea puede realizarse con facilidad si se vuelcan las cantidades de cada atributo en un gráfico de barras. Adicionalmente, cabe destacar que se elegirá el mismo número para todos los *features*, con la intención de producir un conjunto de datos balanceado.

Las imágenes 7.9, 7.10 y 7.11 permiten visualizar los atributos más relevantes para cada clase.

Comenzando con la utilización de las librerías `DLL`, podemos distinguir rápidamente cada clase, podemos observar que, independiente del orden, generalmente las primeras cinco librerías son:

- `kernel32.dll`: módulo del *kernel* de *Windows* que, cuando arranca el sistema, es cargada en una área protegida de memoria para que no pueda ser modificada por ningún usuario o proceso.
- `user32.dll`: librería que contiene funciones de la *API* de *Windows* que tienen que ver con la interfaz de usuario.
- `advapi32.dll`: provee una *API* con funciones relacionadas con llamadas al *registry* del sistema y funciones de seguridad.
- `msvcrt.dll`: módulo parte de la librería *Microsoft C Runtime*, que contiene funciones estándar, como `printf`, `memcpy` y `cos`.
- `gdi32.dll`: librería con funciones para la *Windows GDI (Graphical Device Interface)*, la cual asiste al sistema de ventanas en la creación de objetos bidimensionales.

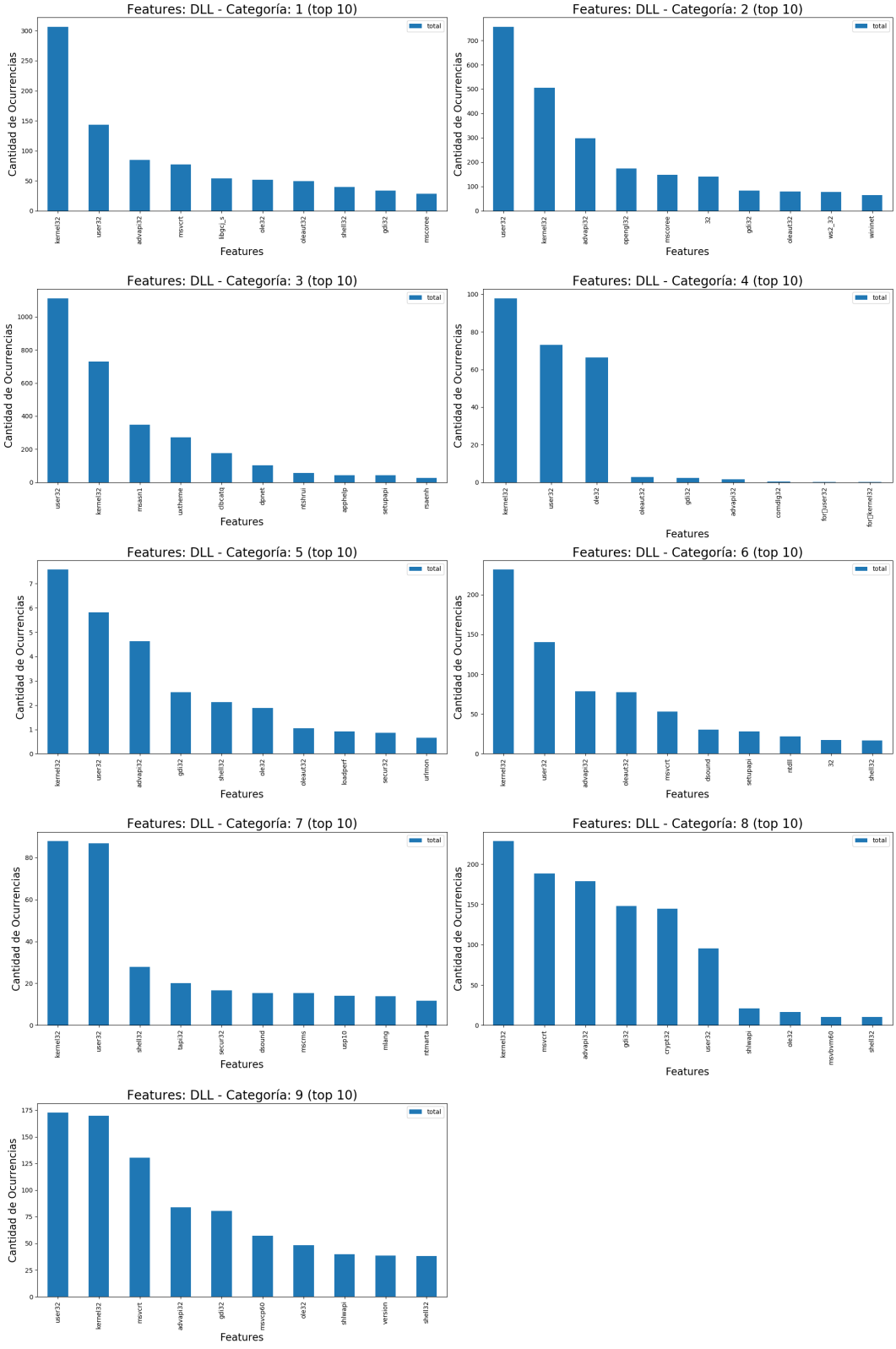

FIGURA 7.9: Top 10 DLLs más relevantes para cada clase de *malware*

Similar a los resultados observados para el uso de librerías *DLL*, para los códigos de operaciones podemos identificar cinco valores como los más recurrentes entre los primeros puestos. Ellos son:

- `mov`: operación que, por lo general, toma dos operandos y copia el valor de uno al otro.
- `push`: función que recibe una lista de registros y los apila en el *stack* en orden descendiente.
- `call`: permite la invocación de la subrutina etiquetada como parámetro.
- `dd`: operación que se usa para declarar un dato de cuatro *bytes*.
- `db`: operación utilizada para la declaración de un dato de un *byte*.

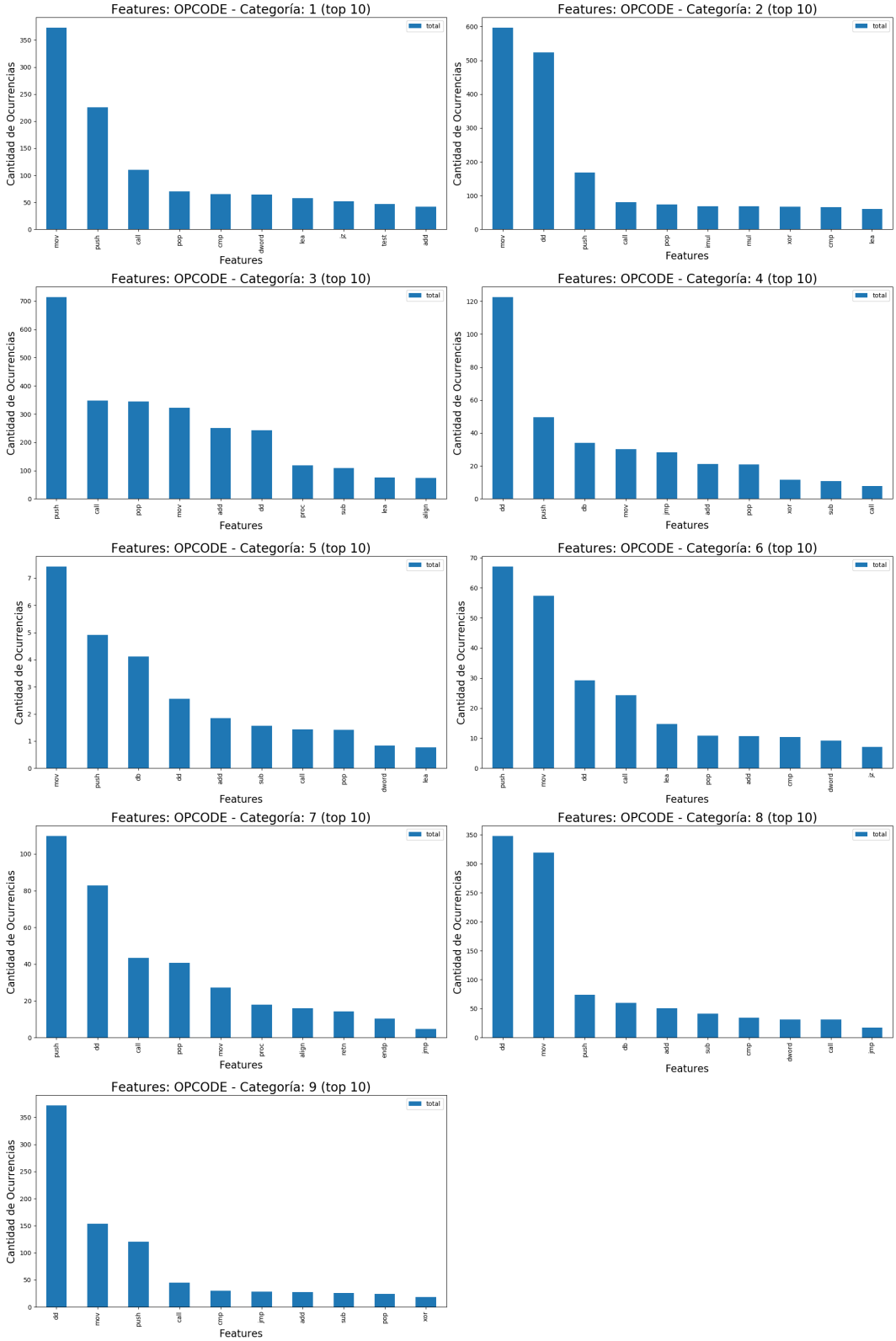

FIGURA 7.10: Top 10 códigos de operación más relevantes para cada clase de *malware*

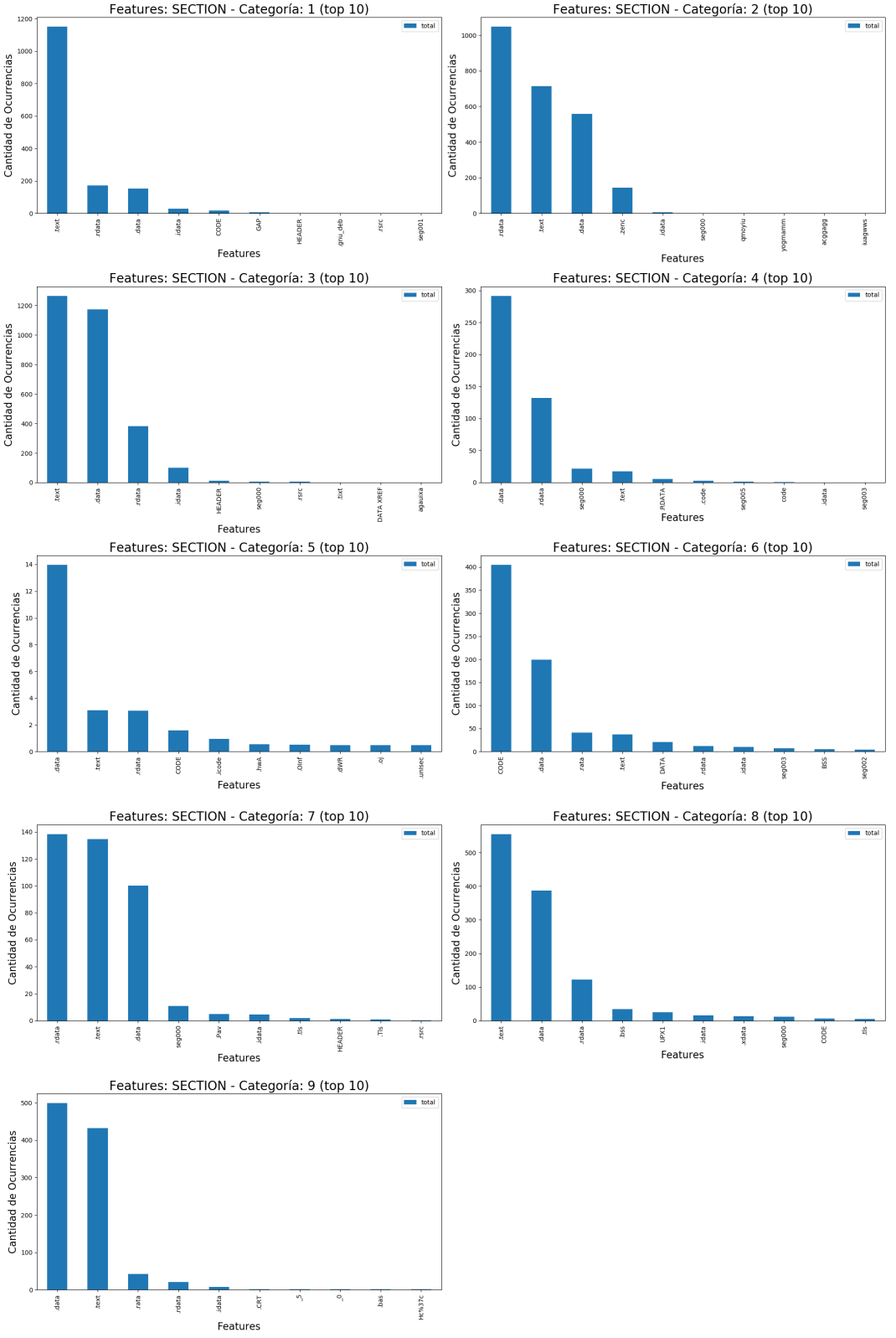

FIGURA 7.11: Top 10 códigos de sección más relevantes para cada clase de *malware*

Estudiando más de cerca los códigos de sección obtenidos, se distingue que, en la mayoría de lo casos los tres códigos más frecuentes son:

- `.data`: sección dedicada a la inicialización de datos y variables.
- `.rdata`: similar a `.data`, con al distinción de ser de sólo lectura.
- `.text`: sección que contiene instrucciones.

7.3.1.4. Consolidación de Resultados

Habiendo determinado los atributos más importantes para cada una de las familias, el único paso restante consiste en construir una lista única de atributos, sin repeticiones. Esta lista es utilizada en conjunto con el archivo generado en el segundo paso, correspondiente a las proporciones de cada *feature* en cada archivo (como se describe en la figura 7.7), para la elaboración de un *dataset* que contiene, para cada archivo, la proporción de ocurrencias de cada uno de los atributos que son relevantes para cualquiera de las familia. La figura 7.12 representa este último paso del proceso de *data mining*.

FIGURA 7.12: Obtención de proporciones de ocurrencias relevantes por archivo

7.3.1.5. Resumen

La obtención de los *features* más importantes para DLLs, códigos de operación y de sección fue un trabajo extenso y laborioso que implicó la combinación de distintas tareas. Requirió primero el estudio de la estructura de los archivos ASM para poder determinar cómo localizar y extraer los datos deseados. Luego se procedió a la construcción y ejecución de distintos algoritmos para la minería concreta de los datos, donde tuvieron que sortearse inconvenientes relacionados principalmente con problemas de *encoding* de los archivos. Por otro lado, tuvo lugar la elaboración de herramientas para validar la información generada a través de gráficos y tablas, y dar soporte para la toma de decisiones. Finalmente, se tuvo que escoger una estrategia para identificar y seleccionar los atributos más relevantes de cada una de las familias de *malware*.

Una visión global de todas las tareas involucradas en el proceso se ofrece en la figura 7.13

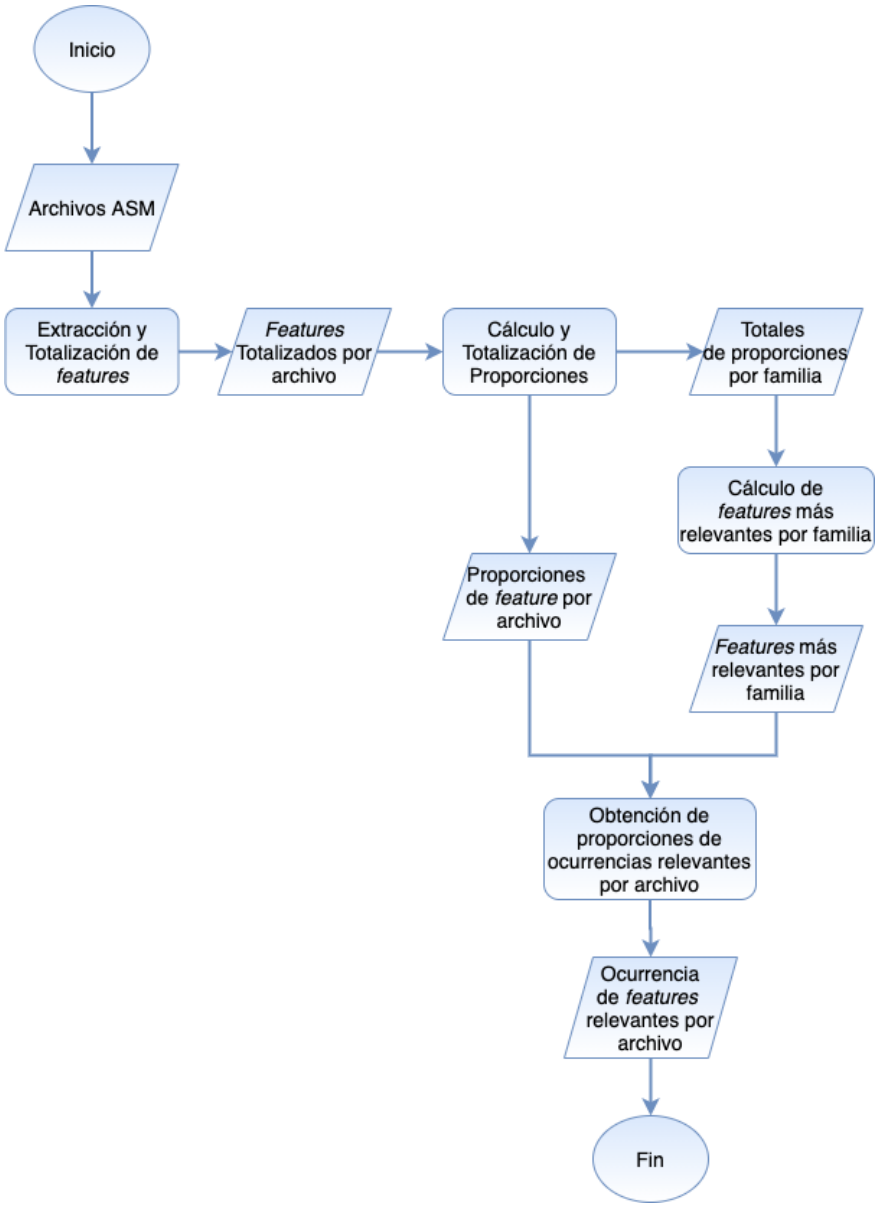

FIGURA 7.13: Obtención de proporciones de ocurrencias relevantes por archivo para DLLs, Códigos de operación y de sección.

7.3.2. *Snapshots* de Archivos ASM

El trabajo realizado en la subsección anterior implica un análisis minucioso de cada lína de los archivos ASM para extraer la información de interés. En este punto, sin embargo, el enfoque se centra en el estudio de la fisonomía de cada uno de los archivos.

De forma similar a la foto de identificación en un documento o un pasaporte, se propone *sacar una foto* en escala de grises de los primeros 1024 *bytes* de cada archivo ASM y utilizar estas imágenes de 32 x 32 *bytes* para entrenar una red neuronal que sea capaz de, por cada muestra analizada, asignar una probabilidad de que la misma pertenezca a cada una de las familias.

7.3.2.1. Captura de los *snapshots*

El primer paso en la de minería de datos en este punto involucró la generación de *snapshots* a partir de los primeros 1024 *bytes* de cada archivo `ASM`. De este modo, se utilizó la librería de *Python* `imageio`[3] para la creación de imágenes de 32 x 32 *bytes* en escalas de grises. La figura 7.14 presenta cinco muestras para cada familia (una por fila). Los *snapshots* se presentan en una escala de 125 % de su tamaño original para su mejor visualización.

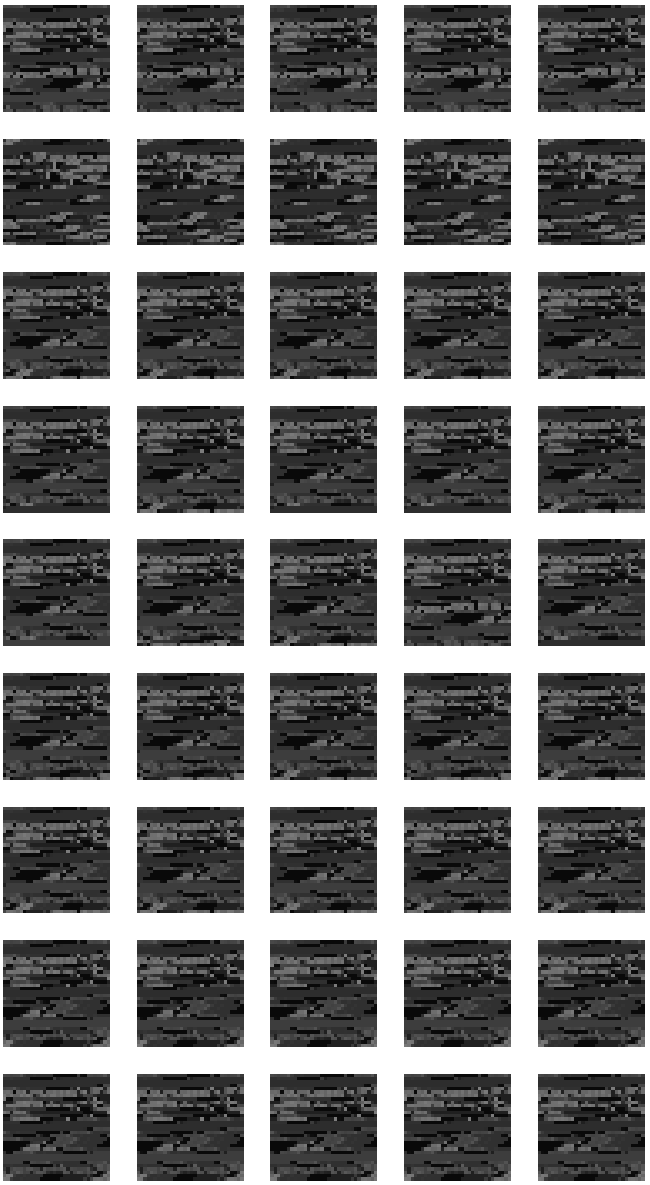

FIGURA 7.14: 5 *snapshots* de 32 x 32 *bytes* de cada familia de *malware*

[3] https://pypi.org/project/imageio/

7.3.2.2. Entrenamiento de la Red Neuronal

Una vez generada la totalidad de las imágenes, se procedió a la construcción de una red neuronal utilizando *TensorFlow*. El objetivo de este proceso no es el de obtener la estimación más precisa posible, sino el de poder observar, a grandes rasgos, la similitud entre archivos pertenecientes a una misma clase. Teniendo que ser capaz de poder realizar estimaciones para cada una de las muestras, la red fue construida utilizando *k-fold*. La técnica de *k-fold* permite dividir el conjunto de datos en *k* partes iguales (conocidas como *folds* o *splits*) y ejecutar sobre ellas *k* corridas. En cada corrida se toma un *split* distinto como conjunto de *test* y a los restantes como conjunto de datos para entrenar el modelo. Para la construcción de la red se utilizó un *k* de 5.

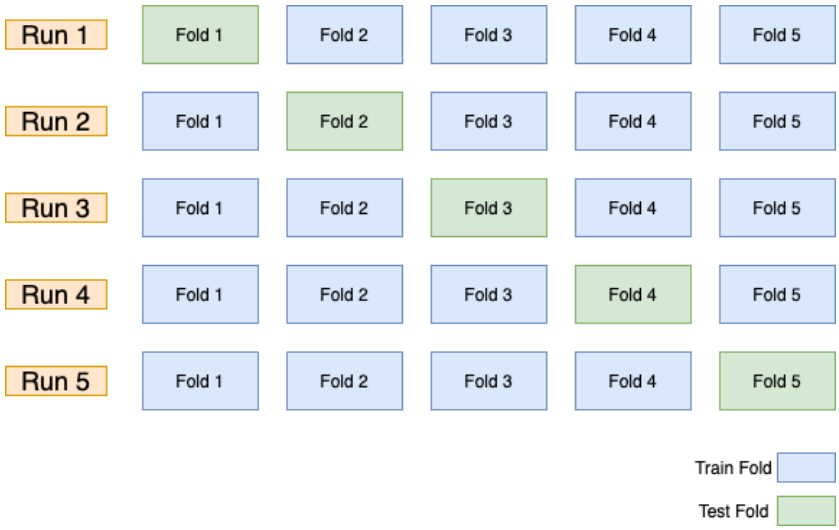

FIGURA 7.15: *k-fold* de 5 utilizado para clasificar los *snapshots*

Una vez ejecutada la red, se obtuvo una precisión de alrededor de 0.62, produciendo resultados con el siguiente formato:

sample	1	2	3	4	5	6	7	8	9
0qV43bnlcBDPF1WlG6tJ	4.79E-06	**0.47899768**	**0.52099687**	8.63E-09	2.08E-08	2.44E-08	1.14E-08	1.11E-07	4.98E-07
C3ZfGBVmThlcWFQDEMvw	1.70E-07	0.00083031	**0.9991696**	2.85E-10	6.53E-10	1.18E-09	3.05E-10	5.75E-09	1.49E-08
db03PLecsviEz6TMZCSn	5.32E-09	**0.99999917**	8.83E-07	5.68E-12	1.83E-11	8.85E-12	1.37E-11	3.63E-11	4.50E-10
Bp9yICsU7lPLvcuhY5Qt	1.24E-20	3.48E-08	1.22E-05	3.87E-20	1.63E-20	1.76E-21	1.17E-23	**0.99998784**	3.99E-11
6q2Y5BOx8RoluXjA10se	1.25E-06	4.38E-05	**0.6143582**	1.93E-06	4.08E-07	6.00E-07	4.01E-07	0.00120943	**0.38438404**

FIGURA 7.16: Muestra de los resultados de la red neuronal para clasificar *snapshots*

7.3.3. Tamaños de Archivos y *Compression Rate*

Continuando con el estudio de las muestras desde el punto de vista de su fisonomía, se procedió a determinar el tamaño del archivo `ASM` y también del archivo `bytes` para cada *malware* que conforma el conjunto de datos. Adicionalmente se calcularon los tamaños de ambos archivos una vez comprimidos mediante `gzip`[4].

[4] *gzip*, o *GNU Zip*, es una técnica de compresión estandar.

Partiendo del supuesto que archivos de una misma familia deben tener un tamaño similar, la minería de estos atributos se realiza con el objetivo de determinar si, en efecto, los distintos tamaños y *compression rate* pueden ser útiles para la identificación de cada *malware* dentro de su correspondiente familia.

Así, se construyó un *script* que, una vez ejecutado produjo un archivo con un formato como el que se ilustra en la siguiente imagen.

sample	asm_compressed_size	asm_size	bytes_compressed_size	bytes_size
01kcPWA9K2BOxQeS5Rju	14352	81988	242366	712704
04EjIdbPV5e1XroFOpiN	587488	4449524	264765	890880
05EeG39MTRrI6VY21DPd	129206	980898	166806	534528
05rJTUWYAKNegBk2wE8X	1834142	13320173	760245	2925056
0AnoOZDNbPXIr2MRBSCJ	127553	935311	164890	593920

FIGURA 7.17: Extracción de los tamaños de archivos y tamaño de archivos comprimidos

7.3.4. N-gramas

El último conjunto de atributos extraído de los archivos *malware* se trata de los 2, 3 y 4-gramas mas importantes, formados por las ocurrencias de los códigos de operación más relevantes. Esta fue, quizás, la tarea más extensa de todas.

Obtener *n-gramas* a partir de una secuencia de valores significa recorrer la misma con una ventana de tamaño *n*, desplazándose una posición a la vez, registrando el valor observado.

El objetivo al construir los *n-gramas* es analizar las operaciones ejecutadas pero, en lugar de enfocarnos en ellas individualmente, nos centramos en estudiar las secuencias de instrucciones que se ejecutan juntas. Se pretende determinar si miembros de una misma familia de *malware* tienden a ejecutar la misma serie de operaciones. Es importante tener en cuenta que se trabaja únicamente con los códigos de operaciones más relevantes para minimizar el número de combinaciones posibles.

De este modo, el trabajo a ser realizado se puede dividir en dos tareas. Por un lado tenemos la construcción de los *n-gramas* basados en los códigos de operación más importantes. Por otro lado se debe realizar la identificación y obtención de los *n-gramas* más relevantes para cada familia.

La primer tarea, la de la obtención de los *n-gramas*, se hace en base a los códigos de operación más relevantes, como fue mencionado en la subsección 7.3.1. Así, los archivos `ASM` deben ser recorridos nuevamente, para armar los *n-gramas*. El diagrama 7.18 grafica este proceso.

FIGURA 7.18: Proceso de extracción de n-gramas

Una vez extraídos los *n-gramas*, se procede al filtrado de aquellos más relevantes, de forma similar al trabajo realizado con otros *features* como DLLs, códigos de sección y operación. A continuación se presentan las imágenes 7.19, 7.20 y 7.21 con los diez *2*, *3* y *4-gramas* más relevantes.

Cabe destacar que la decisión de utilizar solamente los código de operación más relevantes se tomó con la intención de reducir los números de combinaciones posibles. Aún así, se obtuvieron un gran número de *n-gramas* distintos. Por ejemplo, la familia 1 posee 19258 *4-gramas* distintos.

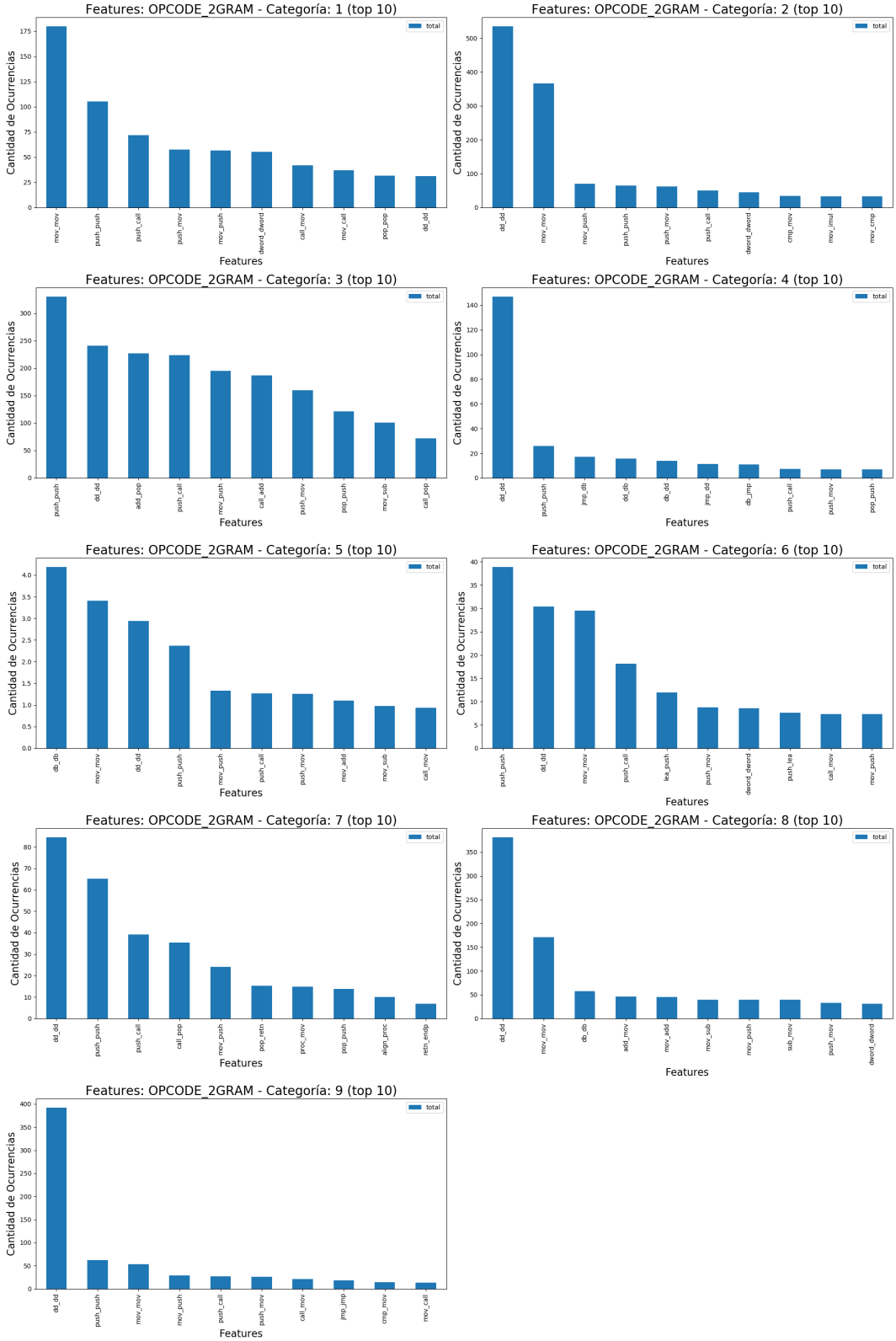

FIGURA 7.19: Top 10 2-gramas más relevantes para cada clase de `malware`

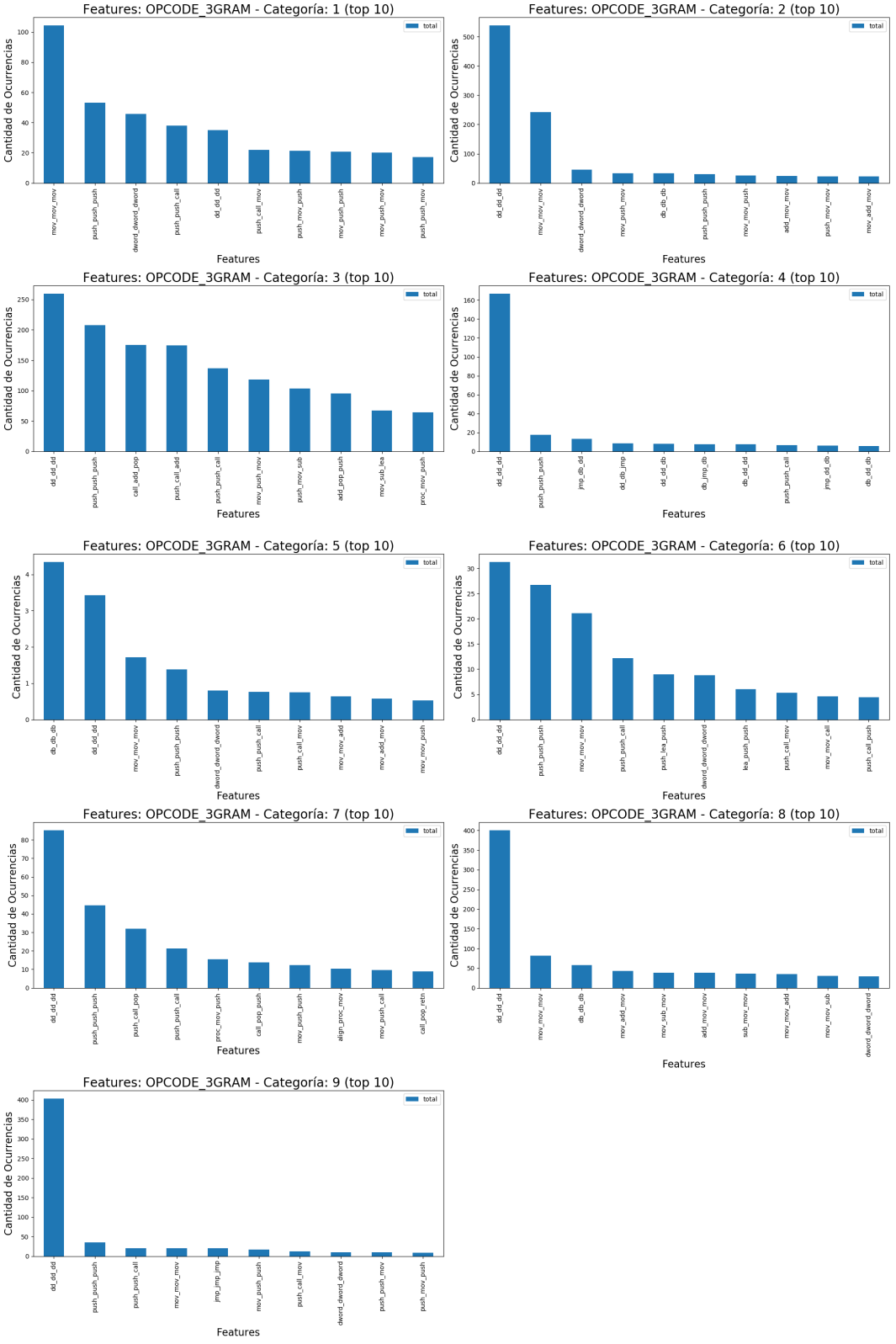

FIGURA 7.20: Top 10 3-gramas más relevantes para cada clase de `malware`

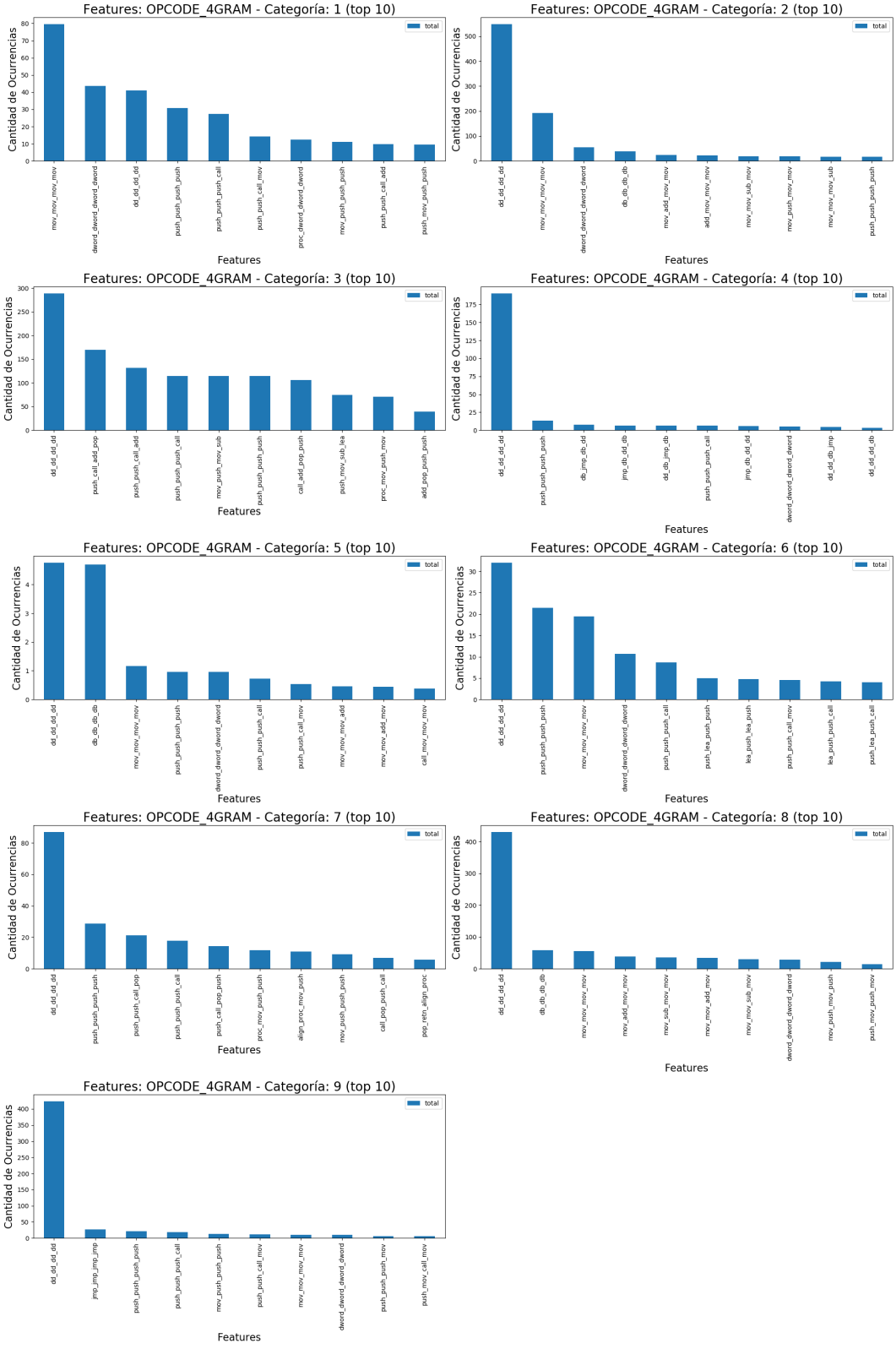

FIGURA 7.21: Top 10 4-gramas más relevantes para cada clase de `malware`

7.4. Detalles Técnicos

Dada la cantidad de atributos a extraer y la diversidad de los mismos, un aspecto fundamental a la hora de implementar los algoritmos de *data mining* fue tratar de logar la mejor abstracción posible, para minimizar la repetición de código. Por ende, la construcción de los procesos mencionados en la sección anterior tuvo como punto central la implementación de dos clases (con sus correspondientes subclases).

- `CategoryFeatureProcessor`: esta clase es responsable de, a partir del nombre de un atributo, como `DLL`, y el número de una categoría, realizar la extracción del *feature* correspondiente de todos los archivos de esa familia. Gracias al patrón de diseño *template method*, las subclases concretas proveen la implementación necesaria para obtener la información según el atributo que se esté analizando.
- `LineParser`: el mismo tipo de diseño mencionado en el punto anterior se aplicó para abstraer el proceso de análisis de una línea individual de un archivo, en búsqueda de la información de interés, como el código de operación, sección o la referencia a una librería `DLL`.

La imagen 7.22 ofrece un diagrama de clases en el que se puede observar en mejor detalle estas clases y su relación.

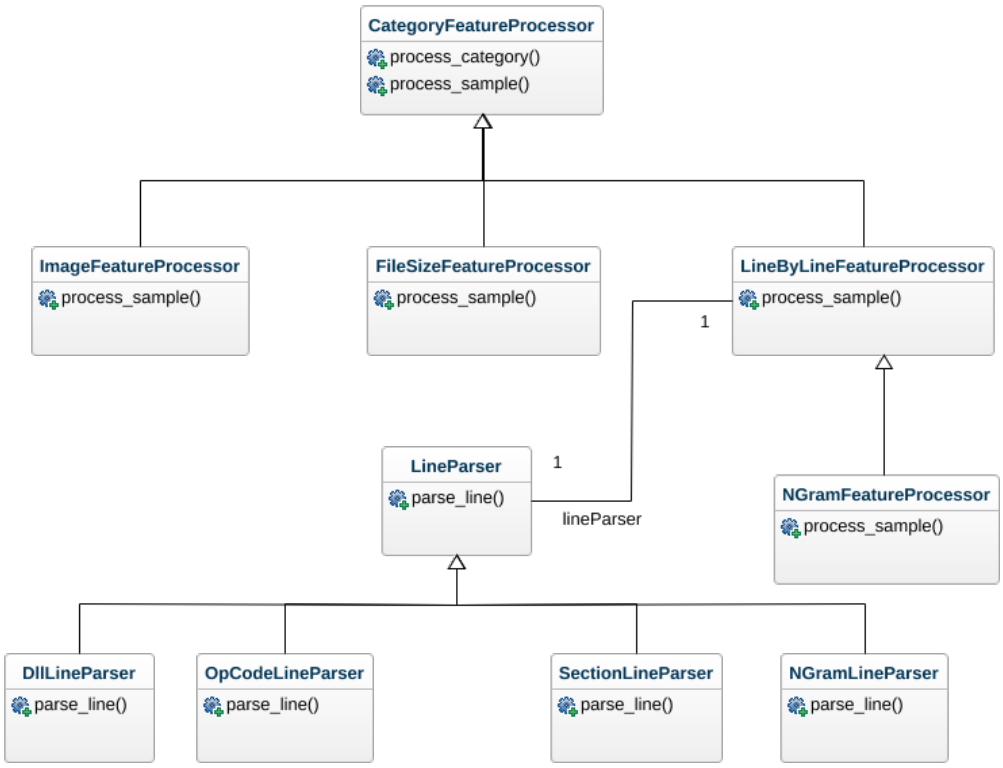

FIGURA 7.22: Extracción de *features* de Archivos `ASM`

Cabe recordar que el estudio de las referencia a librerías, los códigos de sección y de operación, la construcción de *n-gramas* y la clasificación de lo *snapshots* requirieron procesos adicionales. Estos algoritmos ya fueron abordados cuando se discutieron cada uno de lo *features*.

7.5. Conclusión

El proceso de *data mining* fue una tarea muy compleja y demandante. Requirió una considerable inversión de tiempo en comprender la naturaleza y estructura de los archivos de *malware* original antes de poder empezar a trabajar. Luego, fue necesaria la elaboración de diversos tipos de algoritmos que fueran capaz de extraer la información correspondiente a un número muy diverso de atributos; así como también idear estrategias para cálculo y almacenado de resultados intermedio. Adicionalmente, se debieron crear herramientas de apoyo que permitieran validar que los datos extraídos fueran correctos, así como también graficar la información obtenida para poder determinar relevancias y puntos de corte.

El resultado de este proceso es un archivo `csv` con 242 atributos, que servirá como *dataset* para los algoritmos de *machine learning* descriptos en el próximo capítulo.

CAPÍTULO 8

Análisis Exploratorio y Preprocesamiento de los datos

El proceso de *Data Mining* finalizó con la construcción de un *dataset* producto de la extracción de los distintos atributos que fueron identificados como de interés para el análisis y clasificación de *malware*. Sin embargo, el archivo `csv` resultante no está listo aún para ser utilizado para entrenar los modelos de *machine learning*, sino que es necesario llevar a cabo un número de tareas adicionales para mejorar la calidad del mismo.

A través del Análisis Exploratorio de los Datos (**EDA** por sus siglas en inglés), se obtendrá un mejor entendimiento de los datos recolectados mediante métricas y gráficas. A su vez, este análisis permitirá el tratamiento de datos nulos o datos faltantes, determinar la importancia de las variables para proceder a su selección y extracción, y el estandarizado o escalado de los atributos. Se hizo un uso extensivo de distintas librería de *Python* como *Pandas*[1], *matplotlib*[2] y *scikit-learn*[3]

Cabe destacar que este capítulo ofrecerá algunos conceptos teóricos adicionales no cubiertos en el Marco Teórico, por ser demasiados específicos del trabajo realizado en esta sección.

8.1. Estructura y Contenido del *dataset*

Antes de comenzar cualquier tipo de análisis de los datos que conforman el *dataset*, es necesario estudiar la propia estructura del archivo. Esto permite adquirir cierta dimensión del volumen de datos con el que se trabajará. Utilizando *Pandas* se cargó el archivo `csv` en un *dataframe* y, así, se observó que el *dataset* resultante posee **10868** filas, correspondientes a cada una de las muestras disponibles, y **242** columnas con cada uno de los atributos extraídos. Enfocándonos en las columnas, se observan:

[1] https://pandas.pydata.org
[2] https://matplotlib.org
[3] https://scikit-learn.org/stable/index.html

- **39 códigos de sección**: HEADER, .text, .idata, .rdata, .data, .rsrc, .bss, .gnu_deb, .tls, CODE, DATA, BSS, GAP, ; DATA XREF, .CRT, seg001, seg002, .code, .zenc, seg000, yogmamm, iuagwws, qmoyiu, acggagg, .tixt, agauixa, code, .RDATA, seg003, seg005, .icode, .unIsec, .hwA, .OInf, .dWR, .oj, .rata, .Tls, .Pav, UPX1, .xdata, _0, _5, Hc%37c y .bas
- **43 librerías DLL**: kernel32, user32, advapi32, msvcrt, ole32, oleaut32, gdi32, shlwapi, version, urlmon, shell32, mlang, wininet, mscoree, secur32, comdlg32, libgcj_s, setupapi, ntdll, uxtheme, crypt32, ws2_32, apphelp, tapi32, msvcp60, dsound, mscms, msasn1, dpnet, ntmarta, opengl32, ntshrui, 32, usp10, clbcatq, rsaenh, forkernel32, foruser32, loadperf y msvbvm60
- **23 códigos de operación**: proc, dword, push, mov, sub, lea, call, pop, add, align, test, jz, xor, cmp, endp, db, jmp, retn, dd, imul y mul
- **34 2-gramas**: dword_dword, push_push, push_mov, mov_push, push_lea, mov_call, call_mov, mov_mov, lea_push, push_call, call_add, mov_cmp, cmp_mov, call_pop, pop_pop, proc_mov, mov_add, pop_retn, align_proc, dd_dd, dd_db, jmp_db, add_mov, retn_endp, sub_mov, db_dd, db_db, add_pop, mov_imul, mov_sub, jmp_dd, pop_push, jmp_jmp y db_jmp
- **43 3-gramas**: dword_dword_dword, mov_push_push, push_lea_push, push_push_call, push_call_add, push_push_push, mov_mov_push, push_call_mov, push_push_mov, push_mov_push, mov_push_mov, push_mov_mov, mov_mov_mov, mov_mov_add, push_call_pop, mov_push_call, dd_dd_dd, dd_dd_db, mov_add_mov, push_call_push, call_pop_retn, sub_mov_mov, mov_mov_call, lea_push_push, db_dd_db, call_add_pop, proc_mov_push, align_proc_mov, db_dd_dd, push_mov_sub, mov_sub_mov, add_mov_mov, db_db_db, mov_mov_sub, call_pop_push, mov_sub_lea, jmp_jmp_jmp, db_jmp_db, dd_db_jmp, jmp_db_dd, add_pop_push y jmp_dd_db
- **47 4-gramas**: dword_dword_dword_dword, lea_push_lea_push, push_lea_push_call, push_push_push_push, push_mov_push_mov, mov_push_mov_mov, mov_mov_mov_add, mov_push_mov_push, pop_retn_align_proc, push_mov_push_push, push_mov_call_mov, dd_dd_dd_dd, dd_dd_dd_db, mov_mov_mov_mov, proc_dword_dword_dword, push_push_push_mov, push_push_push_call, push_push_call_mov, call_mov_mov_mov, lea_push_push_call, push_lea_push_push, push_call_add_pop, proc_mov_push_mov, align_proc_mov_push, mov_push_push_push, mov_push_mov_sub, mov_add_mov_mov, push_push_call_add, push_push_call_pop, proc_mov_push_push, db_db_db_db, mov_mov_add_mov, mov_mov_sub_mov, mov_sub_mov_mov, add_mov_mov_mov, mov_mov_mov_sub, push_call_pop_push, call_pop_push_call, jmp_jmp_jmp_jmp, push_mov_sub_lea, dd_db_jmp_db, db_jmp_db_dd, jmp_db_dd_dd, call_add_pop_push, add_pop_push_push, jmp_db_dd_db y dd_dd_db_jmp
- **9 probabilidades**: estas columnas corresponden a la clasificación de los *snapshots* de los primeros 1024 *bytes* de cada archivo ASM.

- **4 tamaños de archivo**: columnas que contienen el tamaño de cada archivo `ASM` y `bytes` y sus versiones comprimidas.

Considerando las curvas presentadas en las secciones 7.3.1.3 y 7.3.4, la cantidad de atributos para cada *feature* parece ser razonable. Dos casos, quizás, resulten de interés: el bajo número de códigos de operación y el número relativamente alto de códigos de sección.

Si bien veintitres códigos de operación pueden parecer un número relativamente bajo, sobre todo si se compara con otros *features*, en realidad es una cantidad que resulta razonable, dado que la cantidad de códigos de operación para `x86` varía entre ochenta y cien, según la versión.

En cuanto a los códigos de sección, es interesante analizar los gráficos de barras en la figura 7.11. En ella puede observarse que, para la mayoría de las clases, los valores son relativamente pequeños luego de los primeros cuatro o cinco códigos. Además, como ya fue mencionado anteriormente, se identifican tres o cuatro códigos encabezando las gráficas para todas las familias. En este contexto, obtener un total de treinta y nueve códigos distintos habla de cómo cada familia hace uso de secciones distintas del resto.

8.2. Valores nulos o datos faltantes

La presencia de datos faltantes o nulos en un *dataset* suele tener un impacto negativo en la *performance* de lo modelos que se entrenen en base a él. Por este motivo la primera tarea que suele llevarse a cabo en todo preprocesamiento de datos es la de búsqueda e identificación de valores nulos, o faltantes, y su correspondiente corrección.

Como se describió en la sección 4.3, existen diversas técnicas para el tratamiento de datos nulos. Para seleccionar la más adecuada es imperativo comprender el origen de los valores faltantes.

8.2.1. Análisis de valores nulos

Gracias a distintas funciones de *Pandas* pudo determinarse que la naturaleza de los valores nulos se debían a dos razones:

1. Archivos que, por problemas de *encoding* e integridad no pudieron ser analizados, por ende estas filas estaban prácticamente vacías, salvo por las columnas correspondientes a lo tamaños de archivo. En total se identificaron un poco más de 800 muestras.

2. Archivos que no registraban ocurrencias para atributos que resultaron relevantes para otras familias.

El problema 2 se debe, en todos los casos, al modo en el que se obtuvieron los atributos más relevantes relacionados con códigos de operación, de sección, uso de librerías `DLL` y *n-gramas*. Para ilustrar esta situación, se puede imaginar que, trabajando con `DLL`s, se identificaron las librerías `dll1` y `dll2` como las más relevantes para la familia **Ramnit** y las librerías `dll1` y `dll3` como las más importantes para la familia **Lollipop**. Si sólo se estuvieran analizando estas dos familias, el conjunto de atributos relevantes estaría conformado por `dll1`, `dll2` y `dll3`. Adicionalmente, podría agregarse el escenario en que ninguna muestra de la familia **Ramnit** hace uso de `dll3`. Al ejecutar la consolidación de lo resultados, descripta en la subsección 7.3.1.4, se obtendría un *dataset* donde la columna correspondiente a `dll3` estaría vacía para todos las muestras correspondientes a la familia **Ramnit**.

Antes de proceder a la resolución de los valores nulos vale la pena mencionar que el estudio de estos valores faltantes brindó un mayor entendimiento con respecto a la relevancia real de cada uno de los atributos. Por ejemplo, librerías como `kernel32`, `gdi32` y `ole32` son utilizadas por la mayoría de los archivos (únicamente 296 no lo hacen). Secciones como `HEADER`, `.rdata` y `.text` aparecen en todos los archivos. De forma análoga, los atributos con mayor cantidad de valores faltantes corresponden, en todos los casos, a códigos de sección tales como `.oj`, `.OInf` y `.unIsec`, con 10826 valores nulos. Esto habla de cómo estas secciones, si bien pueden haber resultado relevantes para una de las familias, no fueron realmente utilizadas por el resto.

8.2.2. Resolución de valores nulos

Como se describió en la subsección anterior, los datos faltantes tienen dos orígenes distintos. Para la resolución del primer problema (archivos corruptos), no hubo otra alternativa más que eliminar las filas correspondientes. Mientras que para el segundo problema (columnas sin mediciones), se completaron las celdas con cero, dado que, en este contexto, un valor nulo representa cero ocurrencias.

8.3. Distribución de los valores

Habiendo aplicado una estrategia para solucionar el problema de los campos nulos, se prosigue a estudiar la distribución de los valores para cada *feature*. Nuevamente, podemos utilizar una función de *Pandas* sobre los datos para obtener los mínimos y máximos de cada columna, así como el valor medio y desviación estándar, y la medición para los distintos cuartiles.

Para ejemplificar la información obtenida se seleccionan los tres atributos más importantes (de acuerdo al análisis realizado posteriormente en la sección 8.4) para su estudio:

Atributo	mean	std	min	25 %	50 %	75 %	max
HEADER	0.0017	0.0026	0	0	0.0003	0.0032	0.0578
proc_mov_push_mov	0.0081	0.0112	0	0	0.0002	0.0203	0.0404
add_pop	0.0231	0.0361	0	0	0.0013	0.0549	0.1204

CUADRO 8.1: Estadísticas para los tres atributos más importantes

A partir de la tabla anterior pueden realizarse diversas observaciones:

- Basándonos en la columna **max** que ninguno de los tres atributos es utilizado de manera desproporcionada.
- En todos los casos al menos 25 % de las muestras no presentan ocurrencias para estos atributos.
- Basándose en la desviación **std**, puede considerarse que `HEADER` presenta sus valores compactados alrededor de la media, mediante que para los otros dos casos, éstos se encuentran un poco más dispersos.

A continuación se incluyen las imágenes correspondientes a los gráficos de las densidades de cada uno de estos tres atributos.

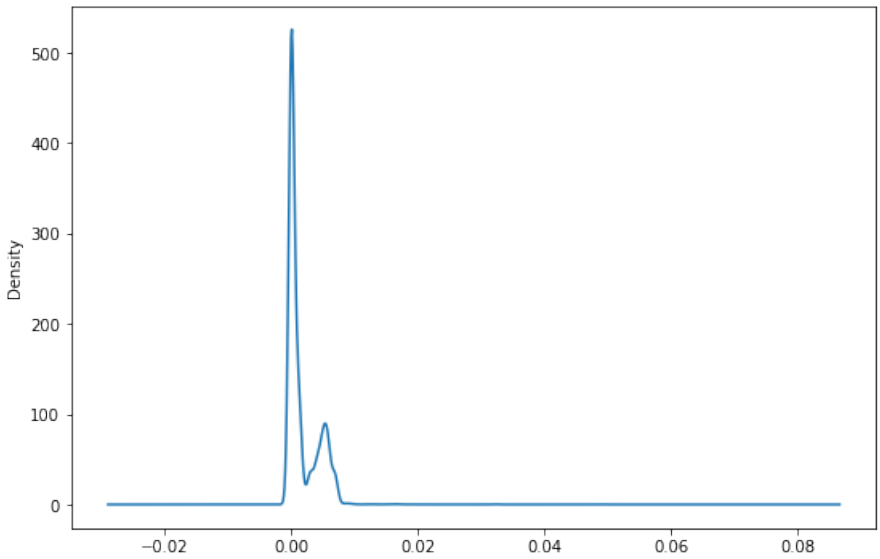

FIGURA 8.1: *KDE plot* para `HEADER`

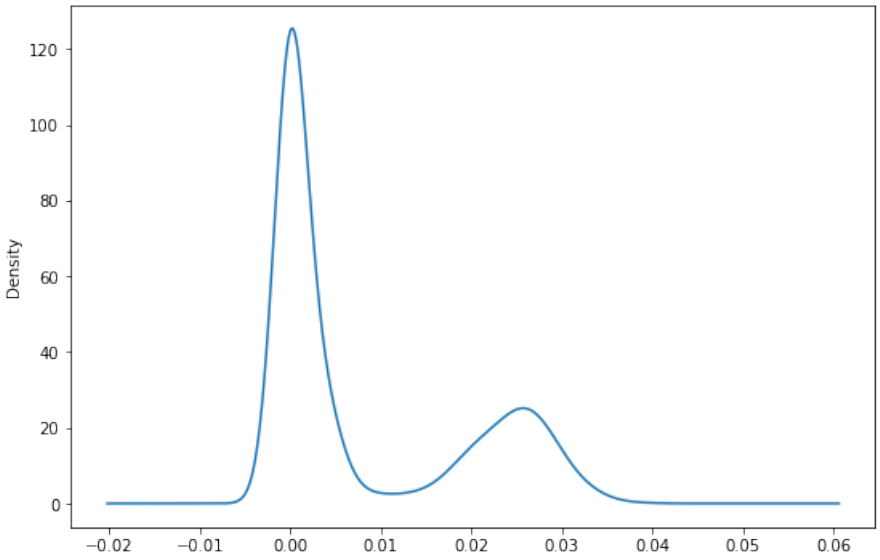

FIGURA 8.2: *KDE plot* para el *n-grama* `proc_mov_push_mov`

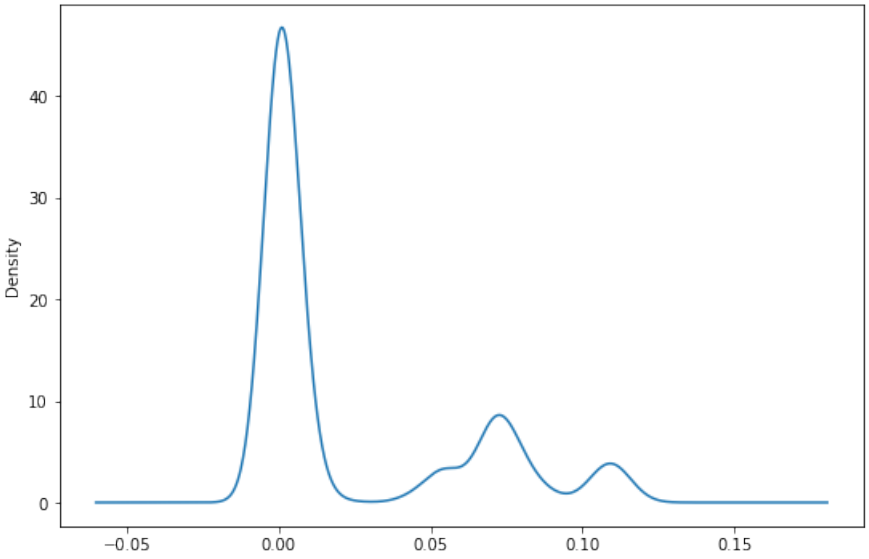

FIGURA 8.3: *KDE plot* para el *n-grama* `add_pop`

Por otro lado, también se encontraron archivos cumpliendo alguna de estas condiciones:

- Estar compuestos enteramente por una misma sección `.data` o `seg000`.
- Utilizar sólo una única librería `kernel32.dll` o `msvbvm60.dll`.
- Ejecutar una misma instrucción `dd` o `db`.
- Ejecutar la misma secuencia de instrucciones (*4-grama*), `push_push_push_call`.

Finalmente, se incluye el gráfico de densidad para los valores observados para la sección `seg000`, ya que resulta curioso que la amplia mayoría de las muestras no

utilizan esta sección en absoluto. Sin embargo, las que lo hacen, no hacen uso de ninguna otra.

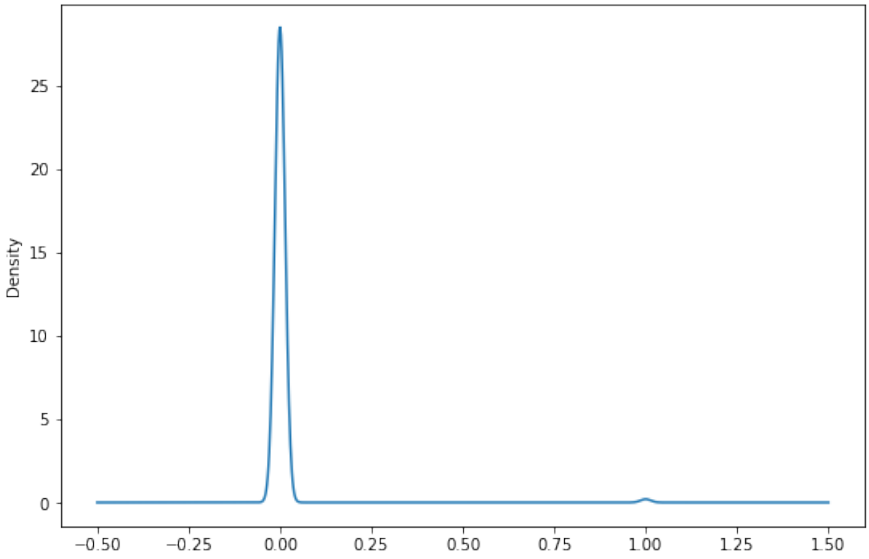

FIGURA 8.4: *KDE plot* para la sección `seg000`

8.4. Importancia de las variables

Con el *dataset* y sus valores nulos ya resueltos, se procedió a determinar la importancia de las variables. Hacer un estudio de la importancia de las variables es de gran utilidad, ya que permite comprender qué atributos son más relevantes para el modelo, lo que, a su vez, aporta beneficios adicionales tales como:

- Verificar la correctitud del modelo y evaluar posibilidades de mejoras al centrar el enfoque en aquellos atributos que son más importante para el modelo.
- Acortar los tiempos de entrenamiento sin sacrificar demasiada *performance* al seleccionar los atributos más relevantes y descartar aquellos cuya incidencia es despreciable.
- Mayor interpretabilidad del modelo sin tener que sacrificar, necesariamente, demasiada *performance*, si se hace una selección de atributos adecuada.

Para llevar adelante este proceso, se utilizó un algoritmo de clasificación *Random Forest*. Se debe recordar que un *Random Forest* es un ensamble de *Decision Trees* que utiliza una variación del método *bagging*, en donde muchos árboles independientes son entrenados utilizando el mismo conjunto de datos. Normalmente un *forest* puede contener varios cientos de árboles.

Utilizando la librería de *Scikit-learn*[4] es posible establecer la importancia de los atributos en un *Random Forest* a través de dos métodos distintos. El método por defecto para computar la importancia de las variables se basa en un mecanismo de decremento mínimo de impureza o impuridad Gini. La impuridad Gini es una medida que determina cuál es la probabilidad de que una nueva observación haya sido incorrectamente clasificada. Cuando un árbol es construido, la decisión acerca de qué variable utilizar para separar cada nodo emplea el cálculo de impuridad Gini.

Para cada variable, la suma de los decrementos Gini por cada árbol del bosque es acumulada cada vez que esa variable es elegida para separar un nodo. La suma es luego dividida por la cantidad de árboles en el bosque para determinar un promedio. Este método cuenta con la ventaja de ser fácil de implementar y el cálculo es rápido. Por el contrario, dado que las estadísticas computadas derivan del modelo de entrenamiento, éstas podrían no reflejar la habilidad del *feature* de ser útil para realizar predicciones que generalicen al conjunto de prueba.

El otro método para determinar la importancia de las variables, que es quizás el más utilizado, consiste básicamente en observar qué tanto se decrementa la precisión del modelo cuando esa variable es excluida. Cada árbol tiene sus propias observaciones *out-of-bag* (OOB), los cuales no serán utilizados durante su construcción. Estas observaciones serán utilizadas para calcular la importancia de una variable específica. Primero, la precisión de la predicción en las observaciones OOB es medida, luego los valores de las variables en las observaciones OOB son mezcladas aleatoriamente, mientras que las otras variables se mantienen igual. Por último, el decremento en la precisión de la predicción en las observaciones mezcladas es medido. El decremento mínimo en la precisión a través de todos árboles es reportado.

Las ventaja de este método es que puede ser aplicado a cualquier modelo, resulta bastante eficiente y es una técnica bastante confiable. Como desventaja resulta mucho más costoso computacionalmente que el método por defecto. También podría sobreestimar la importancia del predictor correlacionado.

En esta investigación se utilizó el método por defecto para el cálculo de la importancia de las variables por ser el que menos costo computacional tiene.

La imagen 8.5 ilustra el resultado que arrojó el proceso indicado anteriormente. En dicho gráfico puede observarse la totalidad de las variables, ordenadas de mayor a menor relevancia y cómo los valores al final de la curva son realmente muy pequeños.

[4] https://scikit-learn.org/

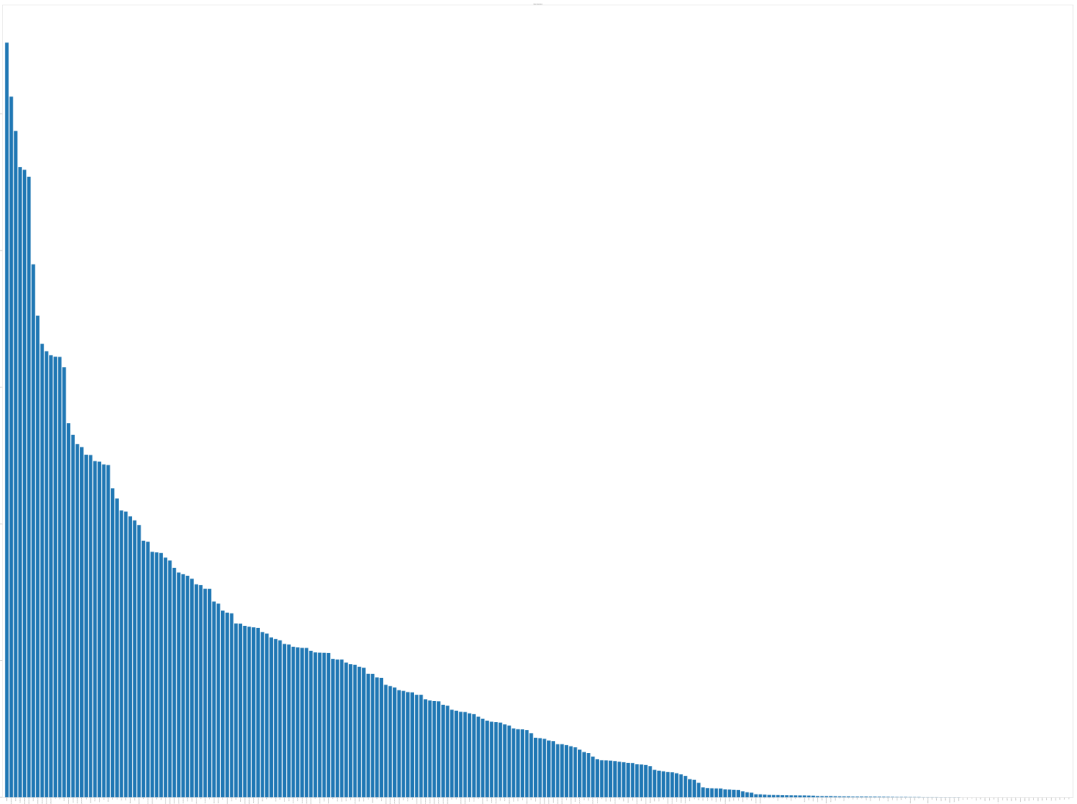

FIGURA 8.5: Importancia de las variables

8.5. Selección de variables

Determinar la importancia de las variables no sólo nos puede ayudar a lograr una mejor interpretación de los datos, sino que también nos permite establecer un *ranking* y seleccionar aquellos *features* que son realmente importantes para el modelo de predicción.

En este contexto, utilizando la librería *scikit learn* para la selección de variables, junto al resultado obtenido en el paso anterior, se utilizó el objeto `SelectFromModel`[5], provisto por la ya antes mencionada librería *scikit learn*. Este objeto seleccionará todos aquellos *features* cuyo valor de importancia sea mayor a un umbral, en nuestro caso a la media de importancia de todos los *features*.

El resultado de aplicar este proceso nos permitió reducir notablemente la complejidad del *dataset*, el cuál pasó de tener 242 columnas a tan sólo 89, cuyas columnas se componen de:

- 6 códigos de sección.
- 4 librerías `DLL`.

[5] https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectFromModel.html

- 15 códigos de operaciones.
- 19 *2-gramas*.
- 21 *3-gramas*.
- 20 *4-gramas*
- 4 columnas correspondientes a los distintos tamaños de archivo.

8.6. Estandarización de atributos

Como ya se mencionó en el marco teórico de esta investigación, muchas veces se trabaja con datos con magnitudes que resultan muy diferentes entre sí. Esto puede ser un problema para muchos estimadores empleados por los algoritmos de *machine learning* que son sensibles al estandarizado de atributos, ellos podrían tener un comportamiento errado si sus valores no fueran más o menos similares. Por ejemplo, muchos elementos utilizados en la función objetivo de un algoritmo de aprendizaje (tales como *RBF Kernel*, *Support Vector Machine* o las regularizaciones L1 y L2 de los modelos lineales) asumen que todos los *features* están centrados alrededor del 0 y tienen su varianza en el mismo orden. Si un *feature* posee una varianza cuyo orden de magnitud es superior a otros, éste probablemente domine la función objetivo y haga que al estimador no le sea posible aprender de otros *features* correctamente.

Dependiendo del problema que se quiere abordar, se pueden utilizar diferentes técnicas, tanto de estandarización, como de normalización de los datos, para que estos resulten útiles a la hora de poner en marcha los algoritmos de *machine learning* que se desean. Para la presente investigación se realizó el escalado estándar, ya que para realizar la extración de atributos utilizando *Kernel PCA*, el escalado estándar funciona muy bien.

8.7. Correlación

A través del cálculo de la correlación de *Pearson*, la cual calcula un coeficiente que establece la medida en la que dos variables se correlacionan, se pudo construir la gráfica 8.6. Los puntos más claros son indicadores de una alta correlación entre los pares, mientras que colores oscuros son indicadores de una correlación leve o de la ausencia de la misma.

Del gráfico se desprenden algunas correlaciones obvias tales como las de los *n-gramas* con otros *n-gramas* similares, como `dd` y `dd_dd_dd_dd` con valor de 0.98. También se percibieron otras correlaciones que hablan de la estructura de los archivos. Por ejemplo, la sección `HEADER` presenta un **0.7** de correlación con la sección `.idata`.

Por otro lado, algunas correlaciones son esperables dado el funcionamiento del lenguaje *assembler*, por ejemplo, la operación `mov`, para el llamado a subrutinas tiene una correlación de **.85** y **.82** con las operaciones relacionadas al pasaje de parámetros, `push` y `pop`.

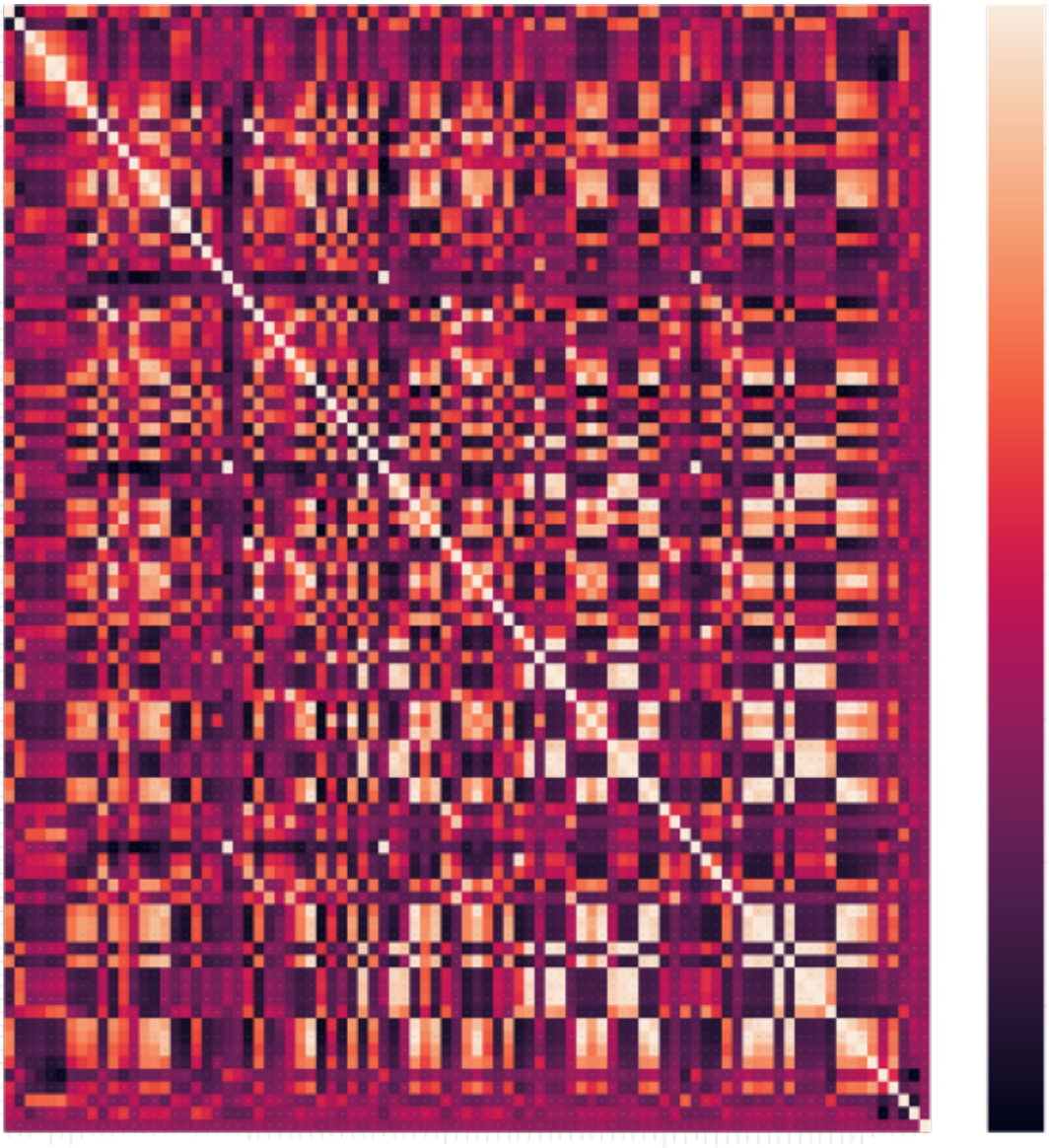

FIGURA 8.6: Correlación de *Pearson*

8.8. Extracción de atributos con *Kernel PCA*

Principal Component Analysis (PCA), es una herramienta que se utiliza para reducir la dimensionalidad de los datos sin perder información. PCA reduce la dimensión hallando algunas combinaciones lineales ortogonales (componentes principales) de las variables originales con la varianza más alta. El primer componente principal captura la mayor parte de la varianza en los datos. El segundo componente principal es ortogonal al primer componente y captura la varianza restante, la cual es dejada

por el primer componente principal y así sucesivamente. Existen tantos componentes principales como el número de variables original. Estos componentes principales no tienen correlación y se encuentran ordenados de manera que los primeros componentes principales expliquen la mayoría de la varianza en los datos originales.

Dado que PCA es un método lineal, sólo puede ser aplicado a conjuntos de datos que sean linealmente separables. Resultará muy buena herramienta siempre y cuando los datos cumplan esta premisa. *Kernel PCA*, en cambio, utiliza una función de *kernel* para proyectar el *dataset* a un espacio dimensional más alto que sea linealmente separable, de manera similar a como lo hace el *Support Vector Machine*.

Para la presente investigación, el análisis de los componentes principales se llevó a cabo utilizando la herramienta *Kernel PCA*. En el gráfico 8.7 se puede observar el resultado de aplicar dicha herramienta.

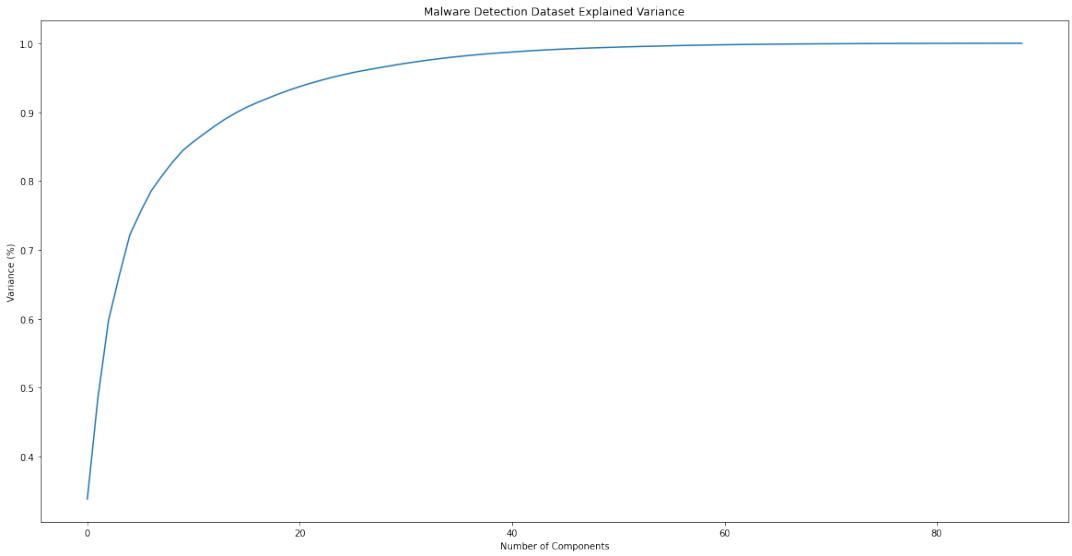

FIGURA 8.7: Análisis de componentes con *Kernel PCA*

Al observar el gráfico, es posible determinar cómo, con un número de componentes superior a 40 se logra explicar un gran porcentaje de la varianza del modelo. En la práctica el número de componentes a seleccionar depende de qué tanta varianza se desea que el KPCA explique. Para esta investigación se fijó el valor en 99.5 %, por lo que tuvieron que tomarse los primeros 53 componentes principales.

8.9. Resumen

La realización de todos los pasos previamente descriptos nos permitió, no sólo reducir drásticamente la dimensionalidad del conjunto de datos, sino también eliminar muestras con las que hubiera sido posible trabajar, dado que los archivos de

origen se encontraban dañados. También se logró un mejor entendimiento del problema que se quiere modelar con los algoritmos de *machine learning*. Además, al rellenar los datos faltantes o datos nulos con ceros, como así también su estandarización se logró llevar al *dataset* a un estado más consistente y completo.

CAPÍTULO 9

Clasificación de *Malware*

En el capítulo anterior se describieron las tareas realizadas para la depuración, seleción y tranformación de atributos. Así se obtuvo un conjunto de datos más consistente, en donde cada una de las variables involucradas son realmente importantes para el modelo. De este modo, se puede dar comienzo a la construcción y elaboración de diversos algoritmos de *machine learning*. Estos algoritmos permitirán realizar las clasificaciones de las distintas familias de *malware*, y a partir de los resultados que estos arrojen, se podrán realizar comparaciones y establecer conclusiones.

9.1. Algoritmos de clasificación

Como ya se mencionó en el marco teórico de esta investigación, existen varios modelos que pueden ser utilizados para problemas de clasificación, de todos ellos se seleccionaron los siguientes:

- *K-Nearest Neighbors*
- *Random Forest*
- *XGboost*
- *Red Neuronal*

Todos ellos serán ejecutados tomando como datos de entrada el *dataset* ya preprocesado. Los resultados serán luego evaluados mediante el valor correspondiente a la precisión gracias a la libreria `scikit-learn`, la matriz de confusión y la gráfica ROC (*Receiver Operating Characteristic*).

9.1.1. *K-Nearest Neighbors*

En una primera instancia se realizó una implementación del algoritmo *K-Nearest Neighbors*. Como ya fue explicado en el marco teórico, el *K-Nearest Neighbors* o KNN, clasifica cada punto mediante el análisis de sus vecinos más cercanos dentro del

conjunto de entrenamiento. El punto es asignado a la clase más común que es encontrada entre dichos vecinos. Es un algoritmo no paramétrico, por lo que no realiza asunciones acerca de cómo los datos están distribuidos.

9.1.1.1. Implementación

Para la implementación del *K-Nearest Neighbors*, se llevaron a cabo los siguientes pasos:

1. Se leen los archivos que contienen los datos preprocesados explicados anteriormente, y el archivo que contiene las etiquetas.

2. Utilizando la librería `scikit-learn` se separa el conjunto de datos en *training* y *testing*, con una proporción 80 % y 20 % respectivamente.

3. Se crean dos objetos, una para el clasificador `KNeighborsClassifier()` y otra correspondiente al análisis de componentes de vecindario, denominada `NeighborhoodComponent` ambas pertenecientes a la librería *scikit-learn*. El *Neighborhood Component Analysis*[1] tiene como objetivo encontrar una transormación lineal que maximice la precisión en la clasificación del vecino más cercano (estocástico) en el conjunto de entrenamiento.

4. Tanto el `KNeighborsClassifier()` con un valor `k=5` establecido como valor por defecto, como el `NeighborhoodComponentsAnalysis()`, también con sus valores por defecto, son colocados dentro de una estructura llamada `Pipeline`. El propósito de un `Pipeline`[2] es ensamblar varios pasos que puedan ser validados juntos utilizando *cross-validation*, permitiendo así la configuración de diferentes parámetros.

5. Ya finalizado el paso anterior, se procede a realizar el `fit()`, ajustando el modelo de acuerdo a los datos de entrenamiento que hayan sido provistos.

6. El paso final será el de la predicción, la cual será realizada sobre el conjunto de prueba. Esto nos permitirá luego proyectar los resultados de la matriz de confusión y la precisión que hemos logrado.

9.1.1.2. Métricas y Evaluación

Para el caso del *K-Nearest Neighbors*, gracias a la utilización del análisis de componentes del vecindario, hemos logrado una precisión en la predicción del 98.98 %, contra 95.34 % de no haberla utilizado. Se debe recordar que la **precisión** es una de las métricas más comúnes utilizadas para medir la performance del modelo de clasificación. Ésta nos permitirá identificar rápidamente la proporción de aciertos que obtuvo nuestro modelo.

[1] https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.NeighborhoodComponentsAnalysis.html

[2] https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

Como ya hemos mencionado en el marco teórico, la matriz de confusión en sí no es una métrica, pero sí nos permite realizar una evaluación acerca de cómo desempeñó nuestro modelo para cada clase. En la matriz de confusión del *K-Nearest Neighbors* en el gráfico 9.1 pueden observarse en la diagonal principal dos valores, un valor entero, el cual representa el número de aciertos que obtuvo el modelo para esa clase, y un valor proporcional. Mientras que por debajo y por encima de esa diagonal se encuentran los valores de las observaciones que han sido clasificadas erróneamente.

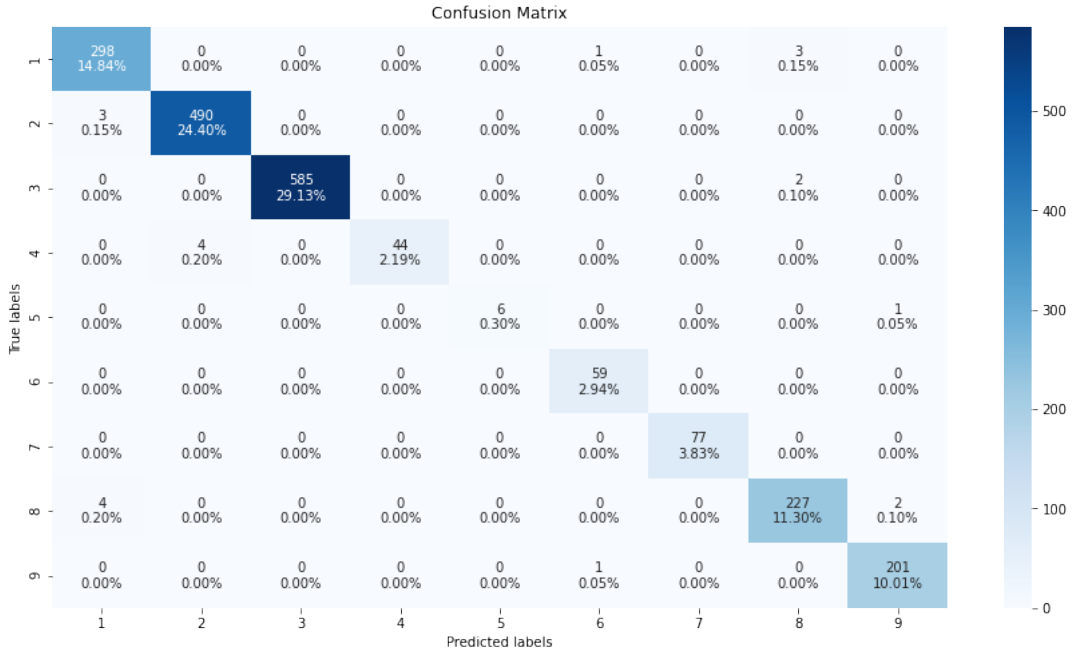

FIGURA 9.1: Matriz de confusión para *K Nearest Neighbors*

Por otro lado, un gráfico que resulta interesante de analizar es la curva ROC (*Receiver Operating Characteristic*). Una curva ROC es un gráfico que representa la performance del modelo en todos los umbrales de clasificación. Esta curva se dibuja en base a en dos parámetros:

- El ratio de los Verdaderos Positivos (*True Positive Rate*, *TPR*)
- El ratio de los Falsos Positivos (*False Positive Rate*, *FPR*)

En donde, el **TPR** es un sinónimo para el *Recall* y está dado por:

$$TPR = \frac{TP}{TP + FN} \tag{9.1}$$

Y el **FPR** se encuentra definido por:

$$FPR = \frac{FP}{FP + TN} \tag{9.2}$$

Esta curva típicamente presenta a los *ratios* de los verdaderos positivos en el eje Y, y el *ratio* de los falsos positivos en el eje X. Esto significa que en la esquina izquierda la gráfica se encuentra en el punto ïdeal", un *ratio* de falsos positivos igual a cero y un *ratio* de verdaderos positivos igual a 1. Por lo tanto, cuanto más grande es el área bajo la curva (*Area Under the Curve*, AUC), mejor.

Si bien estas curvas son normalmente utilizadas para clasificación binaria, es posible extender la curva ROC y el área ROC a clasificación multi-clases, utilizando la herramienta *scikit-learn*. Para realizar esto es necesario *binarizar* la salida, para que luego el clasificador aprenda a predecir cada clase contra otra.

El gráfico antes descripto para el *K-Nearest Neighbours* se puede encontrar en la figura 9.2.

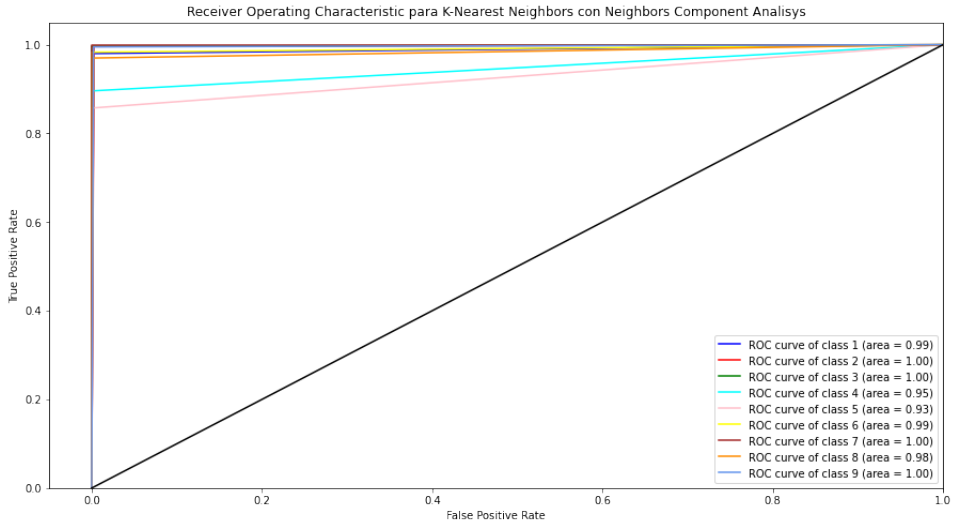

FIGURA 9.2: ROC *K Nearest Neighbors*

9.1.2. *Random Forest*

A continuación se realizó la implementación de un algoritmo *Random Forest*. *Random Forest* se compone de un ensamble de varios árboles de decisión que utiliza *bootstrapping* (selección aleatoria de un conjunto de observaciones con reemplazo), varios subconjuntos aleatorios del *dataset* cuando considera la separación de cada nodo en un árbol de decisión, y el voto promedio para mejorar la precisión de la predicción y controlar el sobreajuste (*overfitting*).

9.1.2.1. Implementación

Para la implementación del *Random Forest*, se llevaron cabo los siguientes pasos:

1. Se leen los archivos que contienen los datos preprocesados explicados anteriormente, y el archivo que contiene las etiquetas.

2. Utilizando la librería `scikit-learn` se separa el conjunto de datos en *training* y *testing*, con una proporción 80 % y 20 % respectivamente.

3. Se crea una instancia del clasificador `RandomForestClassifier()`[3] con todos sus parámetros con valores por defecto.

4. Ya finalizado el paso anterior, se procede a realizar el `fit()`, ajustando el modelo de acuerdo a los datos de entrenamiento que hayan sido provistos.

5. El paso final será el de la predicción, la cual será realizada sobre el conjunto de prueba. Esto nos permitirá luego proyectar los resultados de la matriz de confusión y la precisión que hemos alcanzado.

9.1.2.2. Métricas y Evaluación

Para el *Random Forest* la precisión alcanzada fue del 98.80 %. La matriz de confusión puede observarse en el gráfico 9.3. Mientras que la curva ROC se encuentra en el gráfico 9.4.

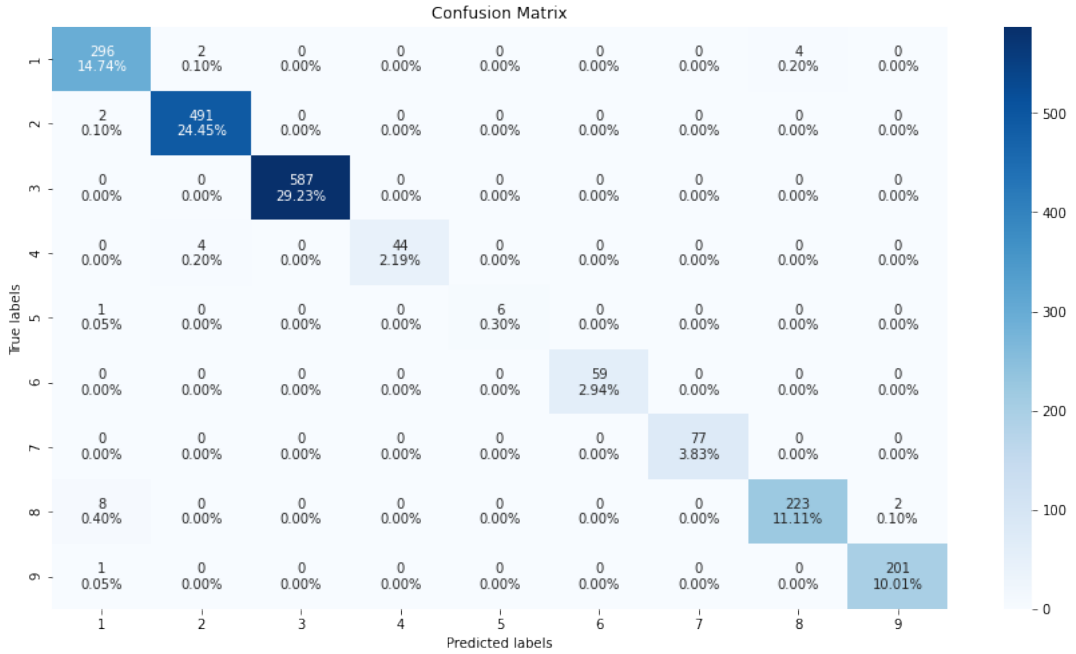

FIGURA 9.3: Matriz de confusión para *Random Forest*

[3] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

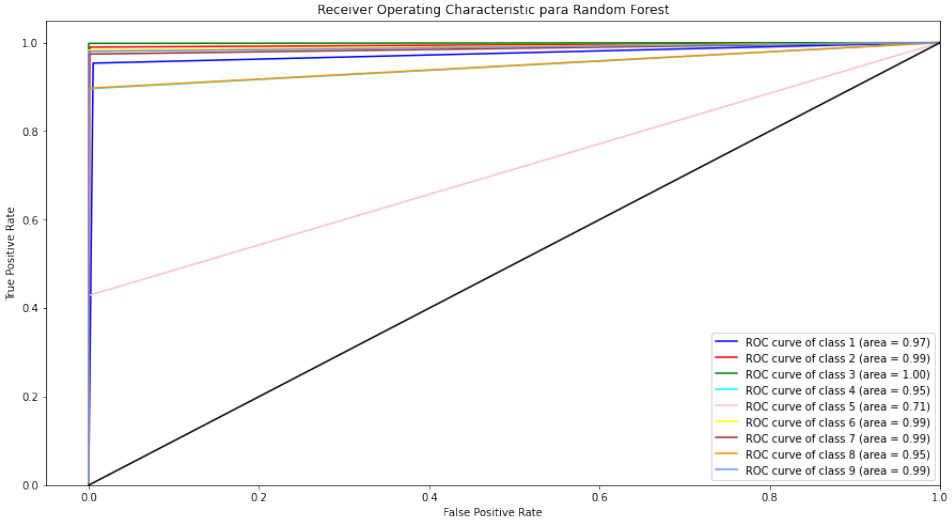

FIGURA 9.4: ROC *Random Forest*

9.1.3. *XGBoost*

XGBoost[4] o *Extreme Gradient Boosting* es una de las implementaciones de algoritmos predictivos supervisados más utilizados en la actualidad.

Como ya se mencionó en el marco teórico de esta investigación, *XGBoost* utiliza el principio de *boosting*. La idea del *booting* es generar varios modelos de predicción ”débiles”secuencialmente, con el fin de generar un modelo más ”fuerte”, con mayor poder predictivo y mayor estabilidad en sus resultados. Para lograr esto, el modelo emplea un algoritmo de optimización denominado *Gradient Descent* (descenso del gradiente).

Cada uno de estos modelos tomará los resultados del modelo anterior y los comparaá. Si el nuevo modelo tiene mejores resultados, entonces se utilizará como base para realizar modificaciones. Si, en cambio, tiene peores resultados, se regresa al mejor modelo anterior y el mismo será modificado de una manera diferente. Este proceso es iterativo y se repetirá hasta un punto en el que la diferencia entre los modelos consecutivos sea insignificante, lo que indicaría que se llegó al mejor modelo posible, o cuando se llega al número de iteraciones máximas definidas por el usuario.

9.1.3.1. Implementación

Para la implementación del *XGBoost*, se llevaron cabo los siguientes pasos:

1. Se leen los archivos que contienen los datos preprocesados explicados anteriormente, y el archivo que contiene las etiquetas.

[4] https://xgboost.ai/

2. Utilizando la librería `scikit-learn` se separa el conjunto de datos en *training* y *testing*, con una proporción 80 % y 20 % respectivamente.

3. Se crea una instancia del clasificador `XGBClassifier()`[5] con todos sus parámetros con valores por defecto. `XGBClassifier()` es una implementación de la API *Scikit-learn* para la clasificación *XGBoost*.

4. Ya finalizado el paso anterior, se procede a realizar el `fit()`, ajustando el modelo de acuerdo a los datos de entrenamiento que hayan sido provistos.

5. El paso final será el de la predicción, la cual será realizada sobre el conjunto de prueba. Esto nos permitirá luego proyectar los resultados de la matriz de confusión y la precisión que hemos logrado.

9.1.3.2. Métricas y Evaluación

Con la utilización de *XGBoost* se logró una precisión de 98.65 % para la clasificación. Su matriz de confusión puede verse en la figura 9.5, mientras que su curva ROC se observa en la figura 9.6.

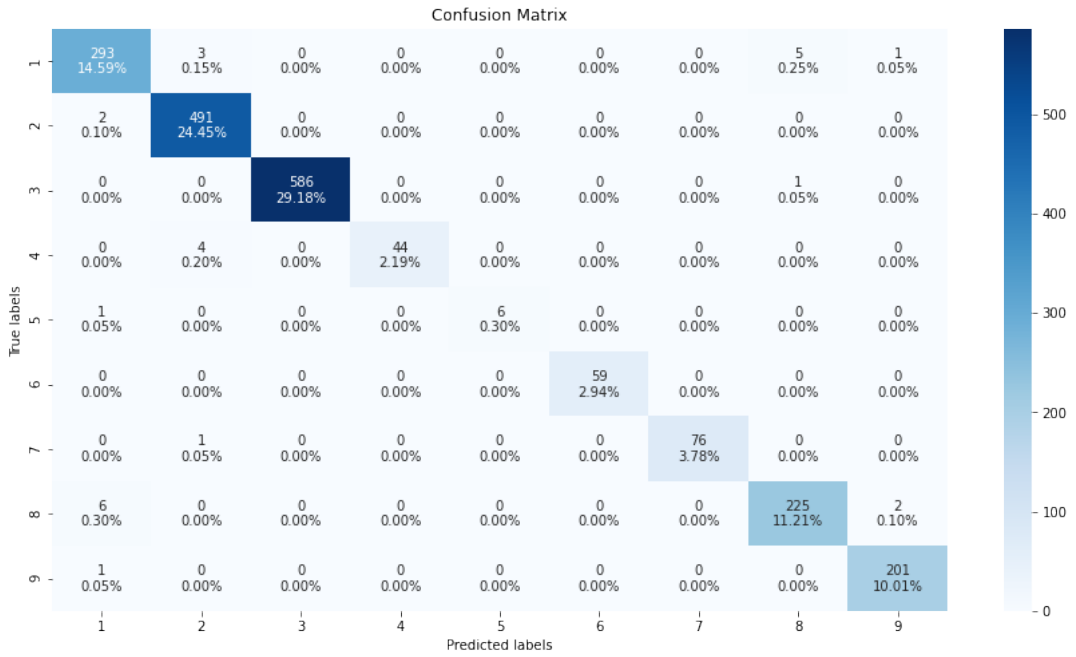

FIGURA 9.5: Matriz de confusión para *XGBoost*

[5] https://xgboost.readthedocs.io/en/latest/python/python_api.html?highlight=xgbclassifier#xgboost.XGBClassifier

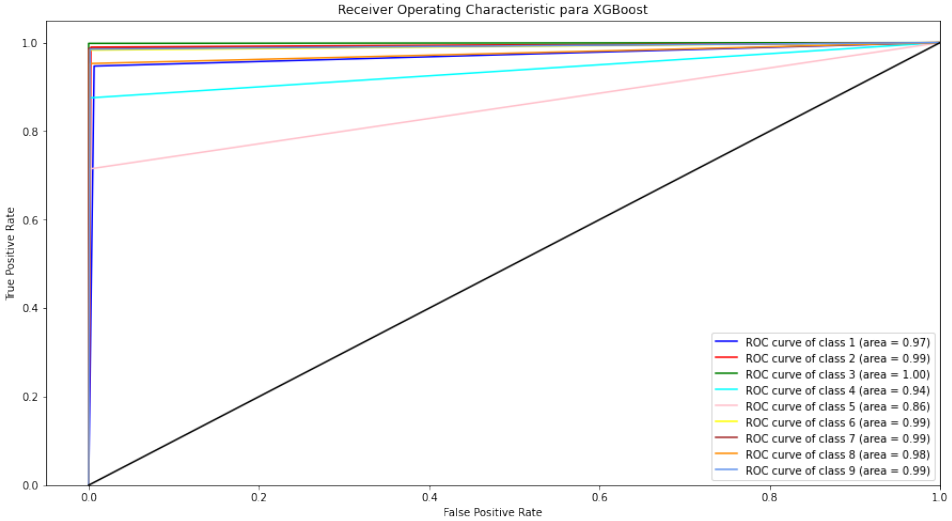

FIGURA 9.6: ROC *XGBoost*

9.1.4. *Artificial Neural Networks*

Las *Artificial Neural Networks* o Redes Neuronales Artificiales, se construyen de simples elementos llamados neuronas, las cuales toman un valor real, lo multiplican por un peso, y lo ejecutan a través de funciones de activación no lineales. Mediante la construcción de múltiples capas de neuronas, cada una de las cuales recibe parte de las variables de entrada y cuyos resultados luego serán pasados a la siguientes capas, la red puede ser capaz de aprender funciones realmente complejas. Teóricamente, una red neuronal será capaz de aprender la forma de cualquier función, con simplemente darle el suficiente poder computacional.

9.1.4.1. Implementación

La implementación de la *Artificial Neural Networks* se llevó a cabo utilizando la herramienta *Tensorflow*[6] integrado con la API embebida *Keras*[7]. *Tensorflow* es una biblioteca de *software* de código abierto para computación numérica, que utiliza grafos de flujo de datos. Los nodos en los grafos representan operaciones matemáticas, mientras que las aristas representan las matrices de datos multidimensionales (tensores) comunicadas entre ellos. Fue diseñado originalmente como una interfaz para expresar e implementar algoritmos de *machine learning*, entre los cuales se destacan las *Deep Neural Networks* o Redes Neuronales Profundas. Mientras que *Keras* es una API de alto nivel de *Tensorflow* para construir y entrenar modelos de aprendizaje profundo. Tiene una interfaz sencilla y resulta fácil de usar e implementar.

Los pasos para la implementación han sido los siguientes:

[6] https://www.tensorflow.org/overview
[7] https://www.tensorflow.org/guide/keras?hl=es

1. Se lee el archivo que contiene los datos preprocesados explicados anteriormente, y el archivo que contiene las etiquetas o *labels*.

2. En el caso de la red neuronal, una vez leído el archivo de *labels* se debió realizar sobre el mismo el *encoding* de los datos, ya que la red así lo requiere. Utilizando la función `OneHotEncoder()`, provista por *scikit-learn*, es posible codificar valores categóricos como un arreglo numérico *one-hot* (también conocido como *one-of-k* o *dummy variable*). Este esquema de codificado o *encoding*, crea una columna binaria por cada categoría y es capaz de devolver una matriz dispersa o un arreglo denso según se requiera. En nuestro caso se optó por un arreglo denso.

3. Una vez realizado el *encoding* de los *labels*, utilizando la librería `scikit-learn` se separa el conjunto de datos en *training* y *testing*, con una proporción 80 % y 20 % respectivamente.

4. A continuación se creó el modelo con las capas que lo componen. Además se deberá también definir cuántas salidas tendrá la red, en nuestro caso 9, ya que se tienen 9 clases distintas. La red se contruyó con dos capas de 128 neronas cada una, ambas con una función de activación *ReLU*, *Rectified Linear Unit*. Y la capa de salida con una función de activación *Softmax*.

5. El siguiente paso consistió en compilar el modelo. Los parámetros que fueron ajustados fueron el optimizador, para el cual se utilizó *adam*[8] (la optimización *Adam* es un método de descenso del gradiente estocástico que se basa en la estimación adaptativa de momentos de primer y segundo orden), la función de pérdida (*loss*), para la cual se definió `CategoricalCrossentropy()`[9] (se utiliza la función *crossentropy loss* cuando se tienen más de dos clases), y la métrica, para la cual le indicamos *accuracy*.

6. Finalizada la etapa anterior, se procedió a realizar el *fit*, ajustando el modelo a lo antes descripto.

7. Como paso final será el de la predicción, la cual será realizada sobre el conjunto de prueba. Esto nos permitirá luego proyectar los resultados de la matriz de confusión y la precisión que hemos logrado.

9.1.4.2. Métricas y Evaluación

La precisión alcanzada por el modelo propuesto de la *Artificial Neural Network* fue de un 99.06 %. Los gáficos 9.7 y 9.8 muestran cómo se comportó el modelo en cuanto a su precisión, o *accuracy*, y su pérdida o *loss*, respectivamente. Se debe recordar que

[8] https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam

[9] https://www.tensorflow.org/api_docs/python/tf/keras/losses/CategoricalCrossentropy

la precisión es una métrica que indicará que tan acertada fue la predicción del modelo comparado con los valores reales de los datos (el valor que representa se debe leer como un porcentaje). Mientras que la función de pérdida retornará un valor real, el cual se calcula tanto durante la fase de entrenamiento como la de validación, e indicará que tan bien, o que tan pobre, se comportó el modelo luego de cada iteración. Cuanto más pequeño sea dicho valor mejor resultará el modelo.

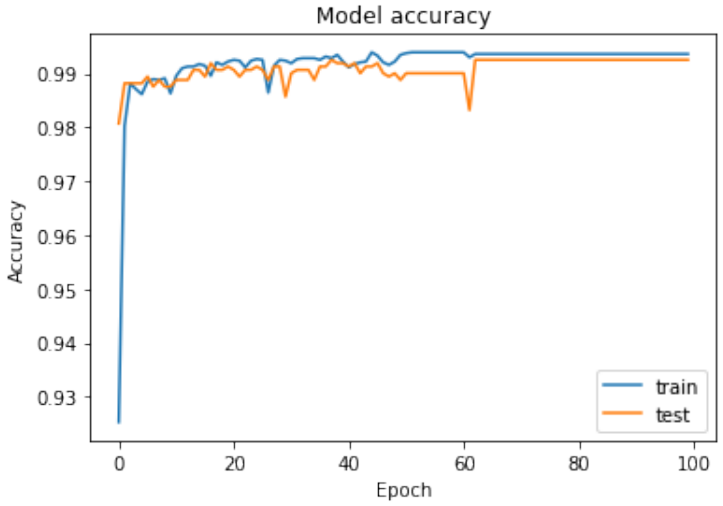

FIGURA 9.7: Precisión (*Accuracy*) del modelo en la clasificación de familias

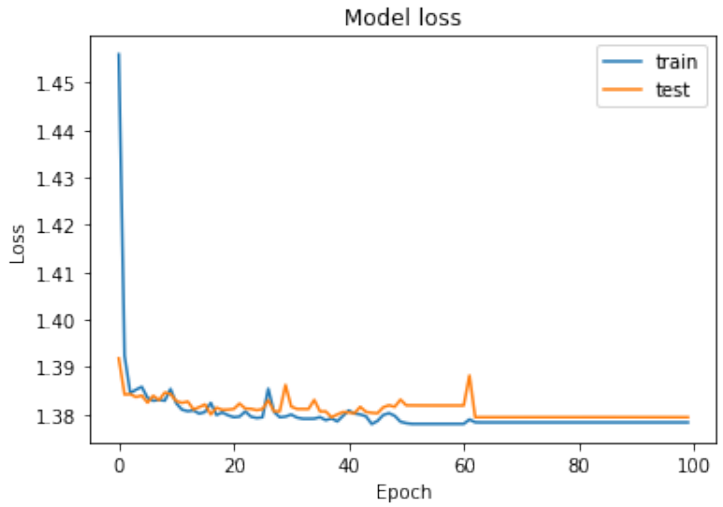

FIGURA 9.8: Pérdida (*Loss*) del modelo en la clasificación de familias

En el gráfico 9.9 se encuentra su matriz de confusión, en la cual se representan todas las predicciones que realizó el modelo sobre los datos de prueba, y qué tan acertadas fueron.

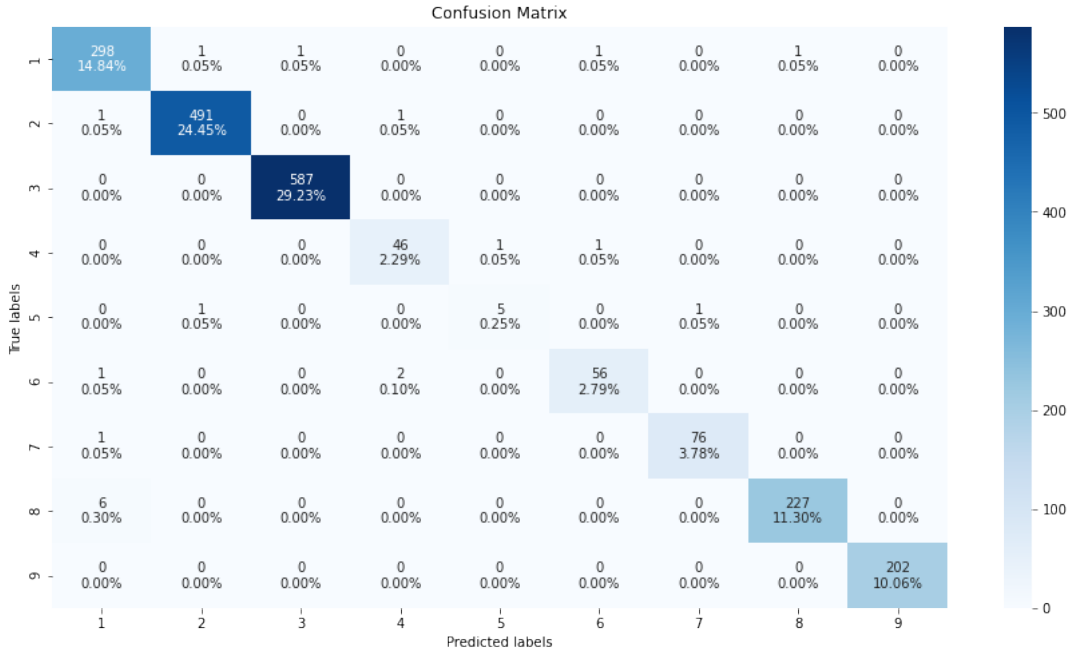

FIGURA 9.9: Matriz de confusión para *Artificial Neural Networks*

Por último, el gráfico 9.10 representa la curva ROC de la red.

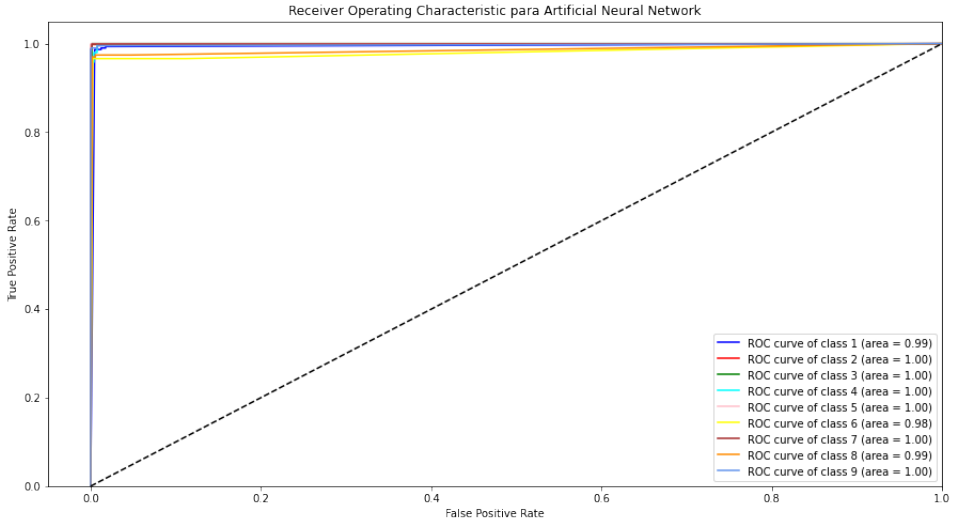

FIGURA 9.10: ROC *Artificial Neural Networks*

9.2. Comparaciones y Conclusiones

En una primera instancia se implementó un modelo *K-Nearest Neighbors*, para dar luego lugar a otros más complejos, como por ejemplo una *Artificial Neural Network*.

El *K-Nearest Neighbors* junto con el *Neighborhood Component Analysis*, han demostrado ser una muy buena opción, alcanzando valores óptimos casi a la par de cualquier otra implementación mucho más compleja y/o costosa. El *Neighborhood Component Analysis* logró darle al *K-Nearest Neighbors* cierta mejora a un algoritmo que ya por sí mismo había alcanzado muy buenos resultados, agregándole un costo computacional prácticamente imperceptible.

A continuación, a modo comparativo, se decidió implementar un método de ensamble, en nuestro caso un *Random Forest*. Estos poderosos algoritmos pueden resultar una muy buena alternativa ya que tienden a realizar una muy buena generalización de los datos y por lo tanto evitar el *overfitting*. Implementarlo resultó bastante simple, logrando una precisión en un rango cercano al 99 % en un tiempo de ejecución muy bueno.

Una implementación del algoritmo de gradiente de árboles reforzados que se ha vuelto muy popular y ampliamente utilizada por la comunidad de científicos de datos en los últimos años ha sido *XGBoost*. Esta robusta y eficiente implementación de código abierto, ha demostrado estar realmente estar a la altura de aquello para lo que fue creado. Su velocidad y rendimiento forman parte de sus carácterísticas principales, ya que internamente los datos los estructura en una matriz llamada `DMatrix`, la cual se encuentra optimizada tanto en el uso de la memoria como en la velocidad de entrenamiento. Esta popularidad y buena fama nos motivó a utilizarlo para la clasificación de nuestras familias de *malware*. Para esta investigación se logró una precisión cercana el 99 % en un tiempo de ejecución más que aceptable.

Por último, se optó por desarrollar una red neuronal, tanto por su popularidad y porque, básicamente son capaces de resolver casi cualquier problema. Para simplificar y hacer más rápida y sencilla su implementación, se utilizó el *framework Tensorflow*. Este *framework*, integrado con la API *Keras*, son capaces de proveer una interfaz sencilla y fácil de usar para la implementación de *Artifical Neural Networks* y *Deep Neural Networks*. La red neuronal fue, en esta investigación, la que logró la precisión más alta y una excelente performace.

Por lo tanto, podría decirse que todas las implementaciones han resultados ser opciones más que válidas y que pueden ser tenidas en cuenta para la clasificación de las diferentes familias de *malware*, haciendo que no sea necesario llevar a cabo una costosa implementación de una red neuronal para clasificar las muestras, ya que es posible obtener prácticamente los mismos resultados con cualquiera de los otros algoritmos de clasificación mencionados en esta investigación.

CAPÍTULO 10

Detección de *Malware*

En el capítulo 9, se definió cuales fueron los pasos realizados para llevar a cabo la clasificación de las distintas familias de *malware*. A continuación, se buscará implementar un sistema de clasificación binario, el cual sea capaz de determinar si un archivo es maligno o benigno.

En el presente capítulo se explicará el proceso de clasificación que se realizó para determinar si un archivo puede ser considerado maligno o no. Para ello, se comenzará describiendo en qué consistieron las tareas de obtención y desensamblado de los archivos, los pasos para generar el *dataset* y las etapas de preprocesamiento del mismo. Finalmente, se abordarán y describirán las soluciones propuestas.

10.1. Obtención y desensamblado de los archivos benignos

El problema de la detección de *malware* puede ser visto como uno de clasificación binaria, en donde lo que se debe determinar es si dado un archivo no conocido por el algoritmo, éste es capaz de determinar si el mismo es detectado como *malware*. Por lo tanto, será necesario contar con muestras de aplicaciones consideradas benignas.

De los sitios *CNET*[1] y *SourceForge*[2], se descargaron un total de 215 aplicaciones livianas. Una vez obtenidas todas estas aplicaciones, se procedió a realizarles el desensamblado. Para ello se utilizó la aplicación *Interactive Disassembler*[3], más conocida por su acrónimo IDA. Este desensamblador es generalmente utilizado para realizar ingeniería inversa sobre los ejecutables y, de este modo, poder convertir una aplicación en un archivo `ASM`, y de ser necesario uno `bytes`. Estos formatos son los que se requerirán para comenzar el proceso de extracción de la información, como fue descripto en el capítulo 7.

[1] https://download.cnet.com/
[2] https://sourceforge.net/
[3] https://www.hex-rays.com/products/ida/support/download_freeware/

Junto a las muestras correspondientes a archivos benignos, se seleccionaron un número igual de *malwares* provenientes del *dataset* con las nueve familias.

Por lo tanto, el total de muestras con los que se realizó el proceso de minería de datos se encontró conformado por un total de 430 archivos, de los cuales 215 se corresponden con los archivos benignos y 215 con los archivos malignos.

10.2. Generación del nuevo *dataset*

De forma análoga al trabajo realizado para la clasificación de familias, todos lo pasos correspondientes a la extracción de la información mencionados en el capítulo 7 debieron ser realizados nuevamente, para generar un nuevo *dataset*.

10.3. Análisis Exploratorio y Preprocesamiento

Con el *dataset* ya creado, se procedió a realizar el análisis exploratorio y el preprocesamiento de los datos, del mismo modo como se hizo para el *dataset* de *malwares*.

En cuanto al análisis exploratorio, se estudió nuevamente la correlación *Pearson*, como se puede ver en la gráfica 10.1, y, análogo a lo sucedido en la clasificación de familias, se pueden observar las correlaciones más altas para los *n-gramas* similares, como por ejemplo `mov_sub_mov` y `sub_mov`.

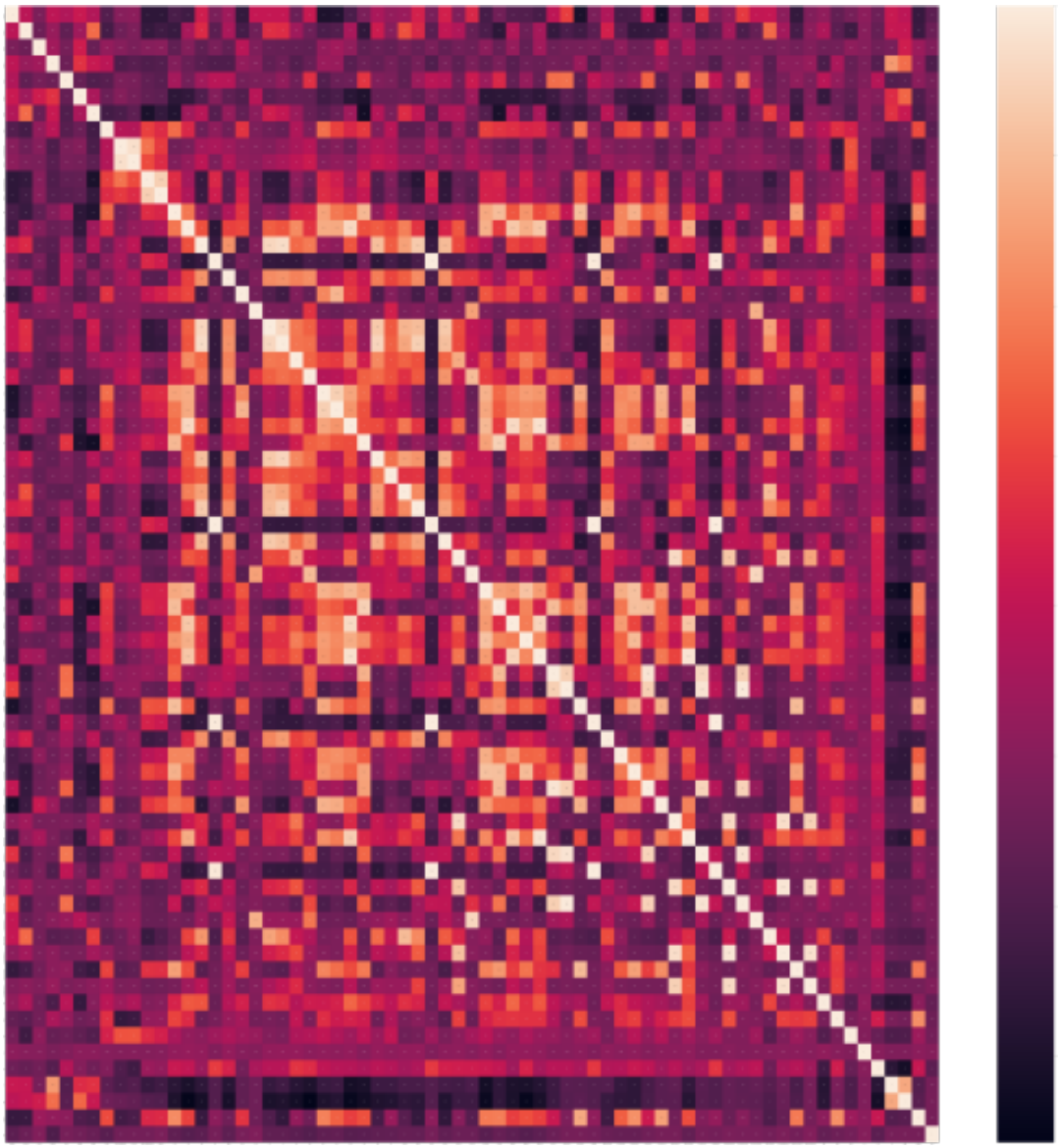

FIGURA 10.1: Correlación de Pearson para la clasificación *malware / no malware*

Por otro lado, el preprocesamiento, como ya se explicó en el capítulo 9, consistió en el tratamiento de los datos nulos o faltantes, determinar la importancia de las variables, estandarizar los datos y por último la selección y extracción de variables.

Para el tratamiento de los valores nulos o faltantes se tomaron las mismas decisiones que para el caso del *dataset* de *malware*, completando los valores faltantes con ceros.

En cuanto al cálculo de la importancia de las variables, el mismo se realizó utilizando un *Random Forest*. En el gráfico 10.2 se puede observar el resultado de aplicar dicho proceso. Si lo comparamos con el gráfico 8.5, podemos ver que son diferentes.

Las variables que allí eran consideradas importantes aquí no lo son e incluso probablemente ni se encuentren presentes, a excepción de la variable `HEADER` que lidera la lista en ambos casos. Estas diferencias ocurren porque cuando se extrajo la información del conjunto de datos inicial (los *samples* de *malware*), el proceso que realizó la minería determinó ciertas características que eran comunes entre ellas, mientras que cuando se tuvo que realizar el proceso nuevamente pero considerando en su lugar los archivos benignos, estas características naturalmente cambiaron.

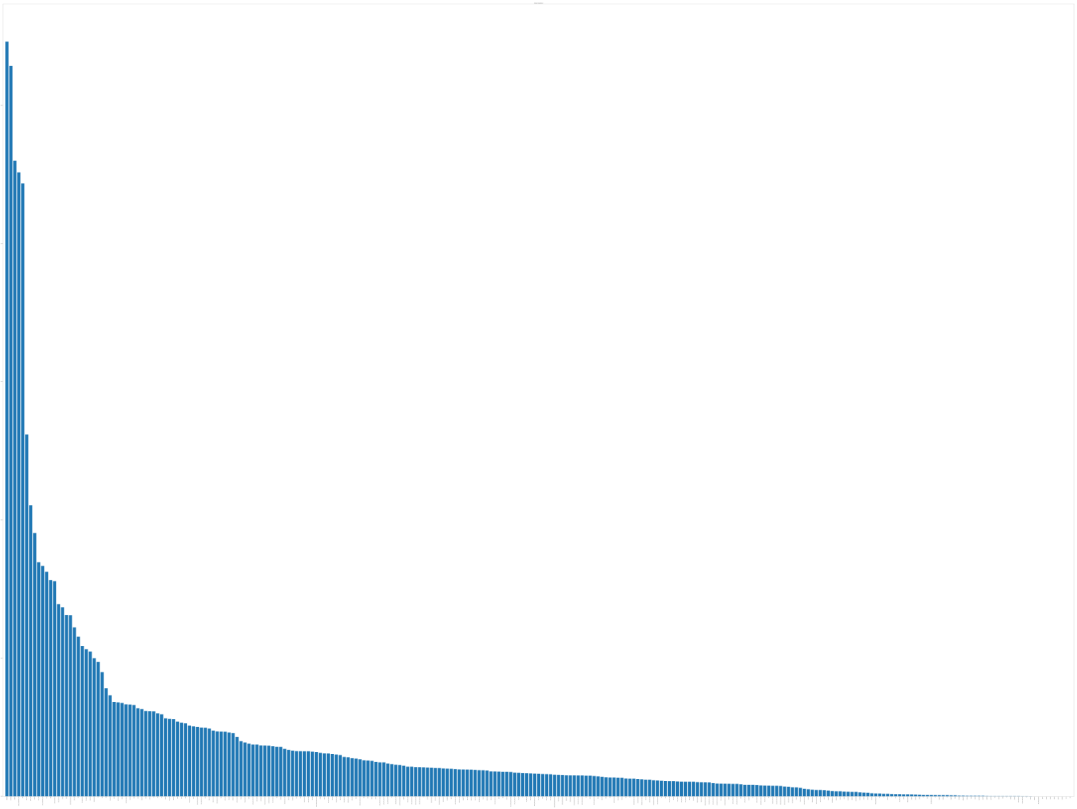

FIGURA 10.2: Importancia de las variables para la clasificación *malware/no malware*

Con respecto a la selección de atributos, se procedió de la misma manera que para el caso de la clasificación de familias. Al aplicar este proceso de selección se redujo el *dataset* compuesto de 269 columnas a 66, lo cual puede ser considerado como una reducción muy significativa.

Al igual que en el caso del *dataset* de *malwares*, se realizó el escalado de los datos utilizando la escala estándar.

Por último, se aplicó el proceso de extracción de variables utilizando *Kernel Principal Component Analysis*, más conocido por su acrónimo KPCA. En el gráfico 10.3 puede observarse como varía la varianza a medida que aumenta el número de componentes. En este caso, con 55 componentes fue posible explicar el 99.5 % de la varianza.

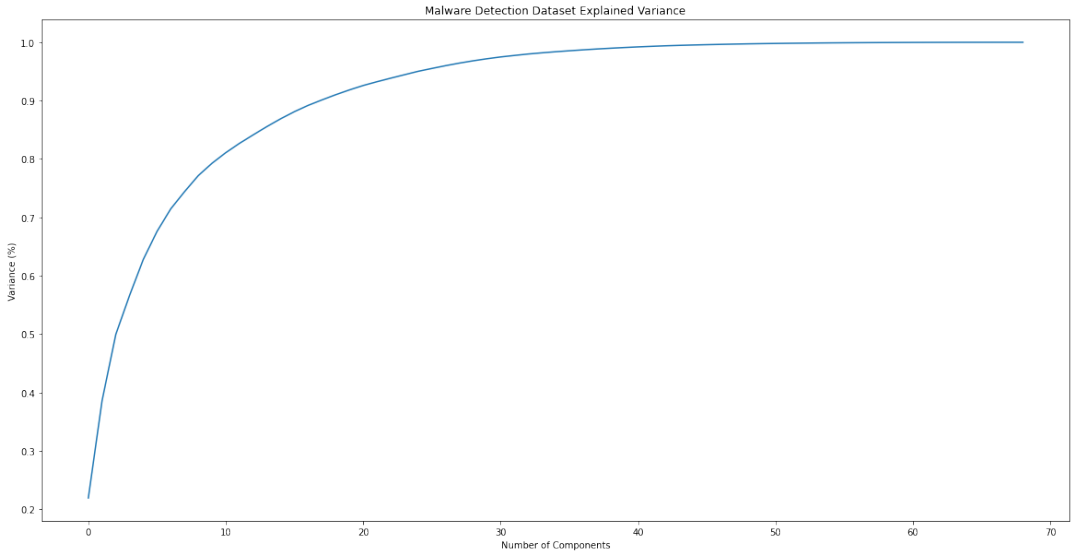

FIGURA 10.3: Análisis de componentes con *Kernel PCA* para la clasificación *malware/no malware*

10.4. Implementación de una solución

El objetivo de esta implementación consiste en poder determinar si un archivo puede o no ser clasificado como *malware* y con qué grado de precisión. Para ello se optó por llevar a cabo la implementación de una *Artifical Neural Network*, ya que la ésta fue la que mejor desempeño obtuvo para la clasificación de las familias de *malware*.

10.4.1. Modelo base

Para ello, se comenzó implementando la red neuronal respetando la misma configuración que se utilizó para la clasificación de familias, pero esta vez la clasificación sería binaria (*malware/no malware*). Esta red no logró un buen resultado, ya que su precisión fue apenas superior al 50 %. Sin embargo, si observamos los gráficos 10.4 y 10.5, se puede ver cómo el modelo está teniendo una precisión mayor al 90 % durante el entrenamiento, mientras que con los datos de prueba obtiene valores inferiores al 60 %, por lo que podría ser un claro indicio de que el modelo está sobreajustando (*overfitting*) los valores a los datos de entrenamiento.

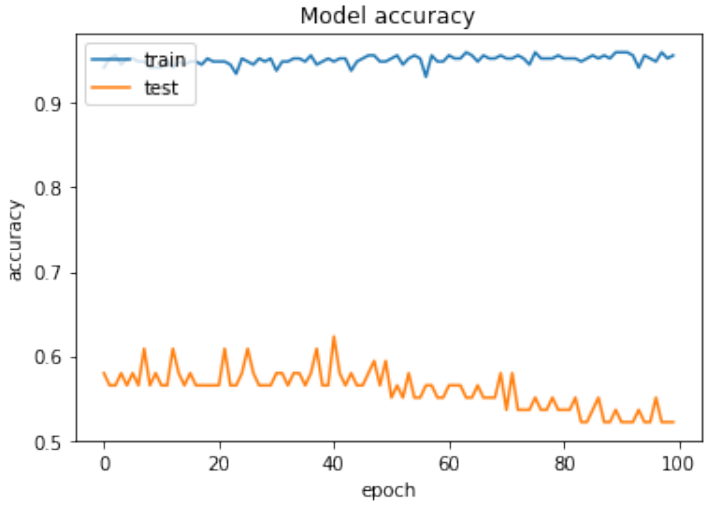

FIGURA 10.4: Precisión (*Accuracy*) del modelo en la clasificación *malware/no malware*

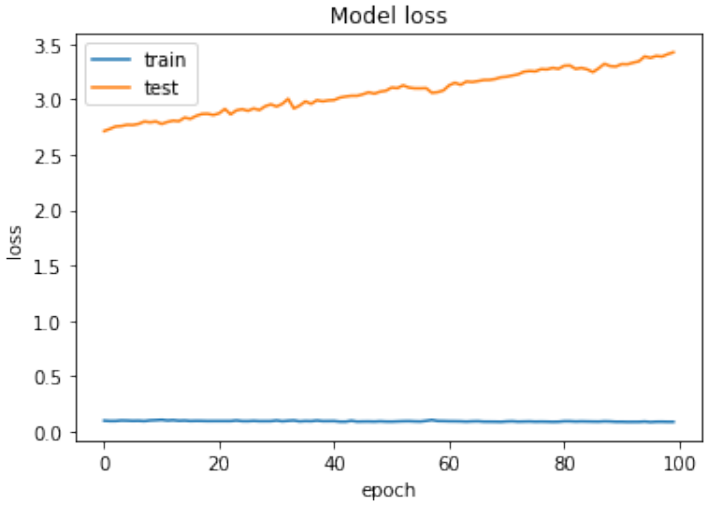

FIGURA 10.5: Pérdida (*Loss*) del modelo en la clasificación *malware/no malware*

10.4.2. Tratamiento del sobreajuste (*overfitting*)

Como se mencionó previamente, el modelo está sobreajustando los valores a los datos de entrenamiento. Existen varias alternativas que se pueden llevar a cabo para minimizar, e incluso eliminar, este problema. A continuación se irán mencionando y explicando en qué consisten cada una de ellas y cómo fueron aplicadas.

10.4.2.1. Complejidad del modelo

Una de las formas más simples de evitar del *overfitting* es simplificando la complejidad del modelo reduciendo el número de capas que lo componen.

Una forma de resolver este problema puede ser comenzando por la creación de un modelo simple, con unas pocas capas y luego ir agregando gradualmente más capas hasta llegar al modelo que mejor desempeño tenga. *Tensorflow*, junto con la herramienta interactiva de visualización *TensorBoard*[4] provee varias herramientas que ayudan al proceso de identificar el mejor experimento, o el conjunto de hiperparámetros más prometedor.

10.4.2.2. Tasa de aprendizaje (*learning rate*)

Muchas veces lo modelos responden mejor si la tasa de aprendizaje (*learning rate*) para los optimizadores es más baja durante el entrenamiento, por lo que es posible, o bien establecer una tasa fija (inferior) previamente definida, o ir decrementándola gradualmente a medida que avanza el entrenamiento. En nuestro experimento, la última solución fue la que mejor resultó.

10.4.2.3. Parada temprana (*early stop*)

Otra manera de prevenir el *overfitting* es deteniendo de manera temprana el proceso de entrenamiento, en lugar de entrenar por un cierto número de *epochs*, el proceso se detiene tan pronto como la pérdida (*loss*), o la función que se esté monitoreando, se eleve, ya que, de seguir entrenando, el modelo sólo empeoraría. Para ello se utilizó una utilidad provista por la API de *Keras*, `callbacks.EarlyStopping`, la cual permite parametrizar el monitoreo de la métrica que se desea y la paciencia que se le tendrá a la misma antes de detener el entrenamiento.

10.4.2.4. Regularización de pesos (*weight regularization*)

Una manera muy común de mitigar el *overfitting* es imponiendo restricciones en cuanto a la complejidad de la red, forzando que sus pesos sólo tomen valores pequeños, lo que hace que la distribución de los valores de pesos sea más regular". Esto se denomina *weight regualrization*, y se logra agregando a la función de pérdida de la red una penalización asociada con tener grandes pesos. Y pueden ser divididas en:

- ***L1 regularization***: en donde, el costo adicionado es proporcional al valor absoluto de los coeficientes de los pesos.
- ***L2 regularization***: en donde, el costo adicionado es es proporcional al cuadrado del valor de los coeficientes de los pesos.

En nuestro modelo hemos probado configurado el modelo para que tomara distintos valores para la regularización L2.

[4] https://www.tensorflow.org/tensorboard?hl=es-419

10.4.2.5. Agregado de *dropout*

Dropout es una de las técnicas más efectivas y más comúnmente utilizados de regularización. La justificación detrás de esta técnica indica que, los nodos individuales en una red no pueden confiar en la salida de los otros nodos, cada nodo debe generar sus propios *features* de salida que sean útiles por sí mismos. Por lo tanto, el *dropout* aplicado a una capa, consiste en quitar (llevarlo a cero) de manera aleatoria cierto número de *features* de salida de la capa durante el entrenamiento.

Con el uso de la API antes mencionada, el usuario puede establecer la proporción (*dropout rate*) de valores que serán quitados.

Todas estas estrategias fueron combinadas entre ellas por la red para realizar el ajuste de hiperparámetros. El resultado de aplicar dicho proceso puede verse en el gráfico 10.6. Allí se observa el listado de todas las pruebas realizadas, y la precisión alcanzada. A partir de estos resultados, se tomaron decisiones de diseño para la *Artificial Neural Network* que ayudaran a mejorar el funcionamiento de la misma.

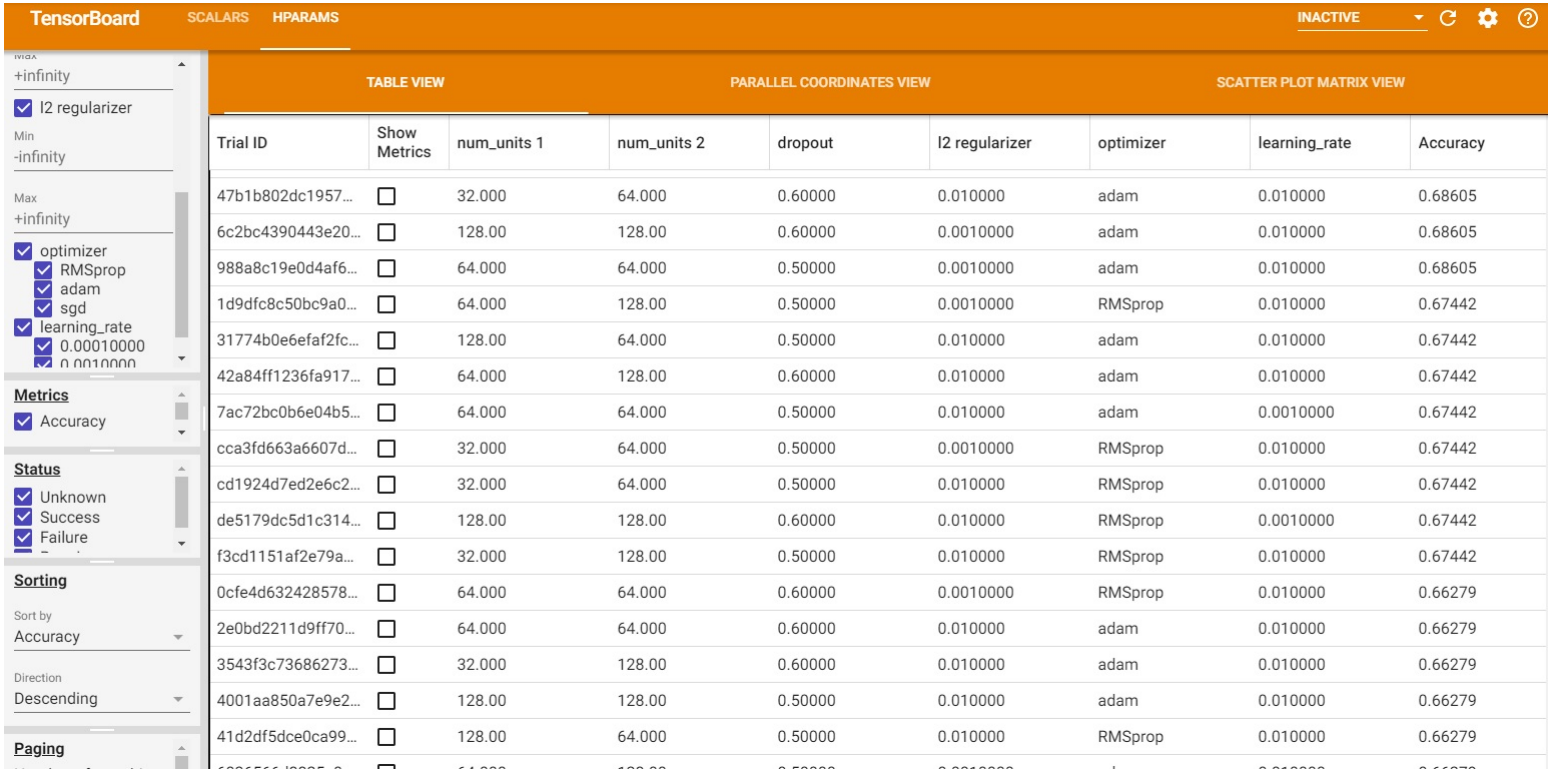

FIGURA 10.6: Configuración de hiper parámetros para la clasificación *malware/no malware*

10.4.3. Resultados obtenidos

Una vez obtenido un modelo con los parámetros optimizados, se realizaron nuevamente las pruebas para la clasificación binaria. Esta vez, el modelo obtuvo una precisión apenas superior al **68.6 %**. Si bien se logró una mejora sustancial respecto al modelo base (alrededor del 40 %), los resultados siguen sin ser buenos. Si se observa la gráfica 10.7 y 10.8 se puede apreciar que, aunque se ha logrado reducir ligeramente el sobreajuste, éste todavía se encuentra presente.

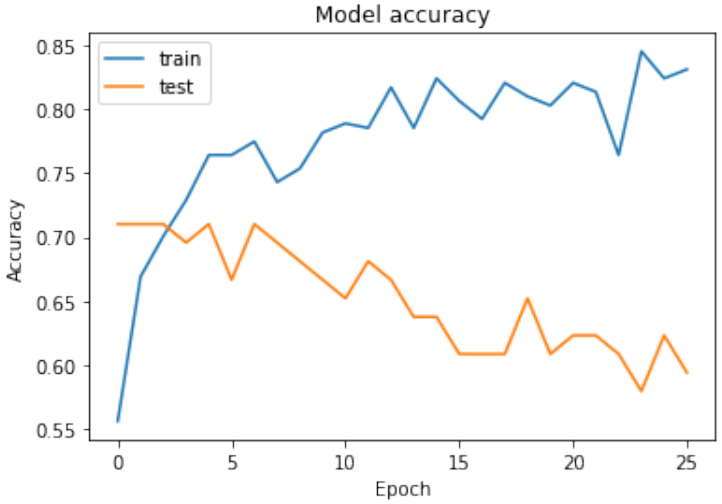

FIGURA 10.7: Precisión (*Accuracy*) del modelo en la clasificación *malware/no malware*

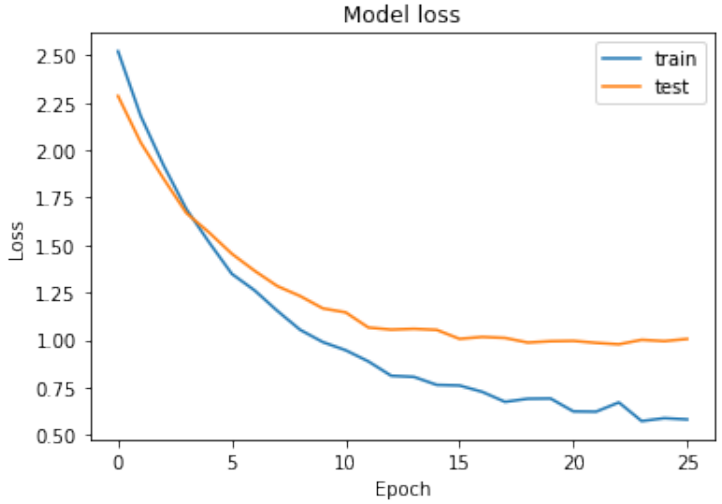

FIGURA 10.8: Pérdida (*Accuracy*) del modelo en la clasificación *malware/no malware*

A continuación se encuentra la gráfica 10.9, la cual muestra la matriz de confusión que dió como resultado la red.

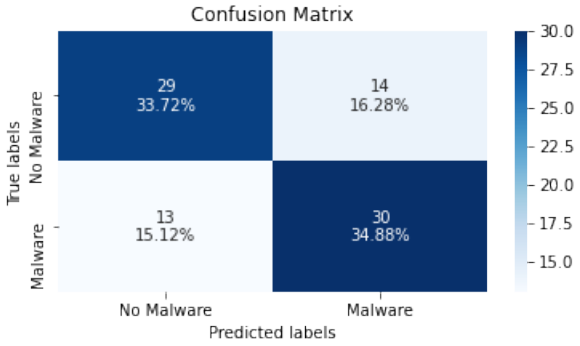

FIGURA 10.9: Matriz de consuficón para el modelo en la clasificación *malware/no malware*

Mientras que su curva ROC se encuentra graficada en la figura 10.10.

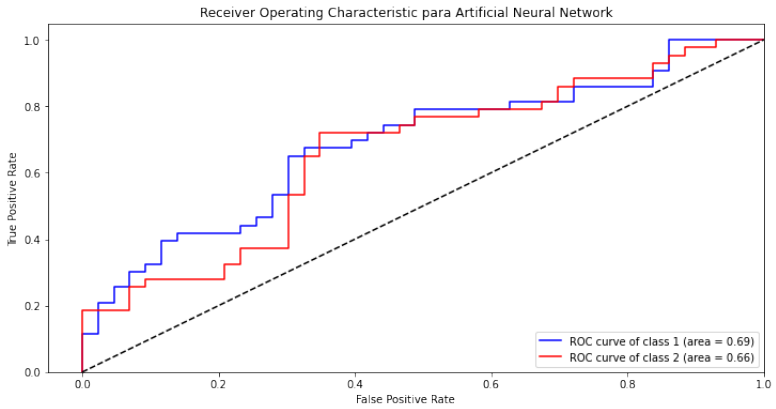

FIGURA 10.10: ROC el modelo en la clasificación *malware/no malware*

10.4.4. *XGBoost* como solución alternativa

En la sección anterior se vio como la implementación de la red neuronal no tuvo un buen desempeño realizando la clasificación binaria, dado que no fue posible reducir completamente el *overfitting*. A continuación, se utilizará un algoritmo de ensamble *XGBoost*, ya que estos, al igual que los *Random Forest* suelen realizar muy buenas generalizaciones y de este modo evitar el problema del *overfitting*.

10.4.4.1. Modelado

Al igual que en caso de la red neuronal, el modelo base del *XGBoost* tuvo que ser ajustado en sus hiperparámetros, ya que tampoco logró alcanzar una buena precisión. Este modelo se implementó utilizando la librería *scikit learn*, la cual cuenta con un *wrapper interface* para *XGBoost*. Esta interfaz permite crear el modelo y parametrizarlo según se desee. A continuación, se mencionarán los parámetros que han sido modificados para esta investigación.

- **Learning rate**: Indica la velocidad a la que aprenderá el modelo, logrando uno más robusto al ir encogiendo los pesos en cada iteración.

- **Max depth**: Este parámetro determinará la profundidad que puede tomar un árbol.
- **Subsample**: Este valor denota la fracción de *samples* u observaciones, que serán tomadas de manera aleatoria por cada árbol.
- **N-Estimators**: Este valor determina el número de árboles (o rondas) que tendrá el modelo.

Además, se utilizó también parada temprana (`early_stopping_rounds`) durante el entrenamiento para disminuir el *overfitting*.

10.4.4.2. Resultados obtenidos

La implementación de este modelo ha logrado cierta mejora respecto a la red neuronal artificial, alcanzando una precisión del **72.86 %**. A continuación se encuentran los gráficos correspondientes a la precisión lograda tanto en entrenamiento como en pruebas, al igual que la gráfica del error de clasificación.

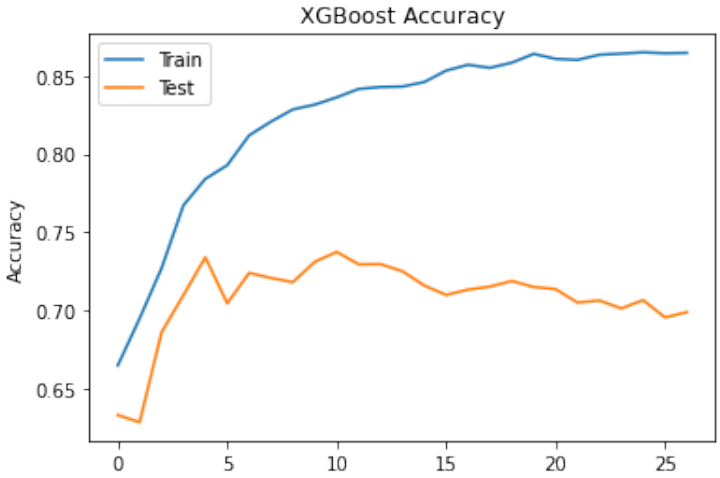

FIGURA 10.11: Precisión (*Accuracy*) del modelo en la clasificación *malware*/*no malware* utilizando *XGBoost*

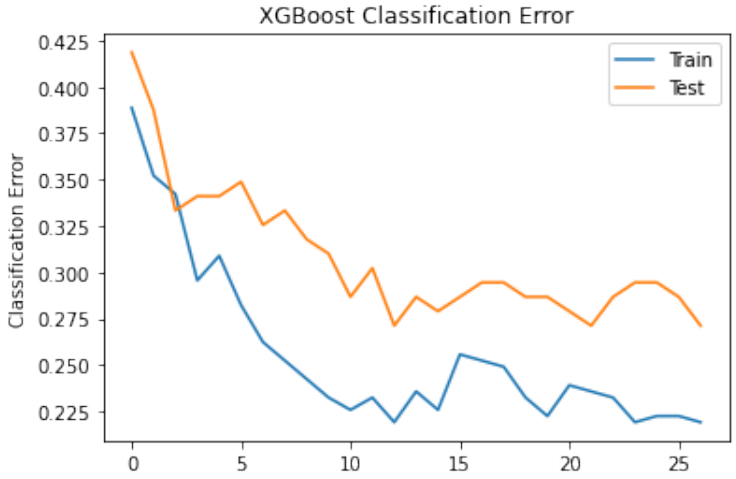

FIGURA 10.12: Error del modelo en la clasificación *malware/no malware* utilizando *XGBoost*

Su matriz de confusión puede observarse en la figura 10.13, y su gráfica ROC en 10.14.

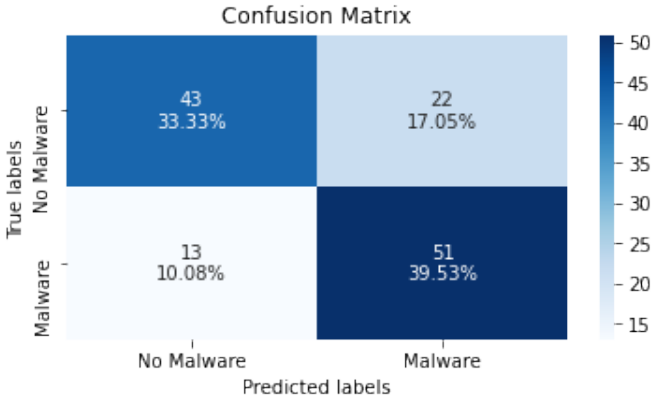

FIGURA 10.13: Matriz de confusión la clasificación *malware/no malware* utilizando *XGBoost*

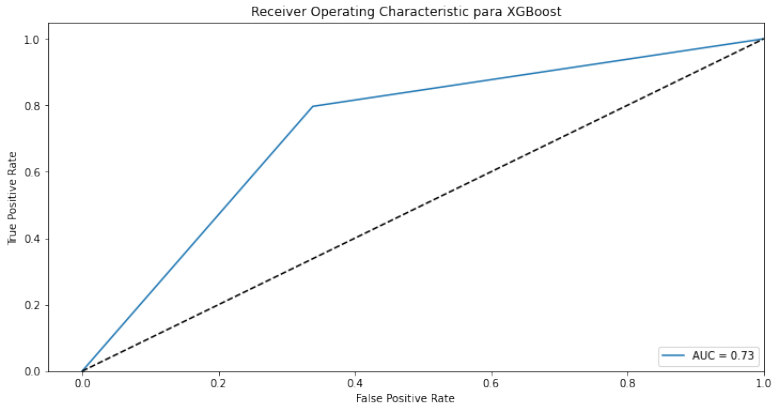

FIGURA 10.14: ROC de la clasificación *malware/no malware* utilizando *XGBoost*

10.5. Comparaciones y Conclusiones

Se comenzó implementando una red neuronal muy similar a la utilizada para la clasificación de familias de *malware*. Esta red, en su modelo base, no arrojó buenos resultados, lo que nos motivó a intentar ajustar sus hiperparámetros y determinar si de esta manera es posible lograr una mejora del modelo en sus predicciones. Para ello, se implementaron varias estrategias que ayudarían a dicho modelo a lograr un mejor desempeño. Este proceso se realizó combinando de diversas maneras los distintos parámetros del modelo, para luego determinar, mediante el uso de alguna métrica, la opción que produce mejores resultados.

Habiendo hecho esto, se modificó el modelo inicial incorporando estos parámetros y se procedió a correr nuevamente las pruebas de clasificación. Si bien se logró cierta mejora en la precisión y reducir el *overfitting*, su resultado no alcanzó valores aceptables.

Por otro lado, como una alternativa a la red neuronal, se llevó a cabo la implementación de un *XGBoost*, y de este modo evaluar si es posible mejorar la precisión de la predicción alcanzada por la red neuronal. Este modelo logró cierta mejora respecto a la red, con una precisión superior al 72 %, sin embargo su valor aún sigue siendo bajo.

Se identificaron al menos tres problemas que pueden impactar negativamente en la performance de ambos modelos: por un lado, no es posible garantizar que los *samples* benignos sean realmente benignos. Esto se debe a que los archivos considerados benignos fueron descargados de sitios que no pueden dar fe de su legitimidad. Por otro lado se encuentra la heterogeneidad de los archivos, siendo estos muy dispares entre sí. Los programas pueden variar desde un *plugin*, hasta un editor de texto. Por último, cabe mencionar el tamaño del *dataset* utilizado, ya que sólo se dispuso de un total de 430 *samples*, mientras que la red de clasificación de familias contaba con un total superior a 10000 *samples*.

CAPÍTULO 11

Conclusiones

Las tecnologías de la información facilitan la vida de las personas en un sinnúmero de actividades, incluyendo comunicaciones, comercio, viajes, estudio, trabajo, y muchas otras. Los *hackers* también utilizan la tecnología para lograr sus propósitos, aprovechando las vulnerabilidades de los sistemas para robar información, obtener acceso, o cualquier otra actividad con un fin malicioso. Los métodos tradicionales de detección utilizados por los antivirus ya no resultan efectivos ante los avances y la evolución de estos ataques.

Expertos en la ciencia de datos, junto a la comunidad anti *malware* han aunado sus esfuerzos para encontrar una solución que pueda hacer frente a esta problemática. Esto alentó a las empresas dedicadas a la industria del software antivirus a comenzar a utilizar técnicas de *machine learning* y *deep learning* en el desarrollo de sus productos.

Motivados por los avances logrados en este campo se decidió dar inicio a esta investigación e intentar poner a prueba dichas afirmaciones.

De este modo tuvo lugar un extenso número de tareas de *data mining* dedicadas a procesar un conjunto de datos comprendido por cerca de once mil archivos *malware* en formato `ASM` y `bytes`, con el objetivo de extraer de ellos los atributos que se estipularon que podrían llegar a ser representativos del comportamiento o de la fisonomía de estos programas.

Esta etapa de la investigación fue la más demandante, requiriendo no sólo la escritura de procesos para extraer los datos necesarios, sino también la construcción de herramientas que permitieran visualizar la información obtenida en cada etapa, con el fin de validar los resultados obtenidos, detectar errores de manera temprana y apoyar la toma de decisiones para cuestiones como el establecimiento de punto de corte para la selección de atributos relevantes.

Una vez consolidados los datos en un *dataset* único, la fase de preprocesamiento y depurado nos permitió evaluar distintos aspectos de los datos recabados, reafirmando así suposiciones hechas al comienzo del proceso de *data mining*, como la importancia de los tamaños de los archivos. También permitió descubrir, mediante el estudio de la relevancia y correlación de las variables, relaciones subyacentes y descartar *features* que habíamos supuesto que podrían ser de utilidad, como los *snapshots* de los primeros 1024 *bytes* de los archivos `ASM`. Así mismo, la utilización de distintas técnicas permitió una considerable reducción de la dimensionalidad del problema, con un sacrificio prácticamente insignificante de performance.

Finalmente, la elaboración de distintos modelos de *machine learning* para la clasificación de familias de *malware* permitió poner a prueba la eficacia de los datos extraídos, obteniendo resultados superiores al 98 % de precisión para todos los casos, e incluso superiores a 99 % para la red neuronal, sin siquiera tener que realizar un *tuning* de los parámetros.

Por otro lado, el sobreajuste observado en los modelos de clasificación binaria nos llevó a un estudio más profundos de las distintas alternativas para resolver este problema, aunque, en definitiva, el tamaño pequeño del conjunto de datos resultó ser un inconveniente imposible de eludir.

En conclusión, teniendo en cuenta los resultados obtenidos, es posible afirmar que la utilización de técnicas de *data mining* y *machine learning* para clasificar familias de *malware* han resultado muy efectivas. En cuanto a la clasificación binaria, estamos seguros que, de contar con un conjunto de datos considerablemente más grande (en el orden de los miles) para realizar el entrenamiento, los algoritmos podrían lograr resultados ampliamente superiores a los alcanzados en este trabajo.

CAPÍTULO 12

Trabajos Futuros

A continuación se enunciarán algunas características que podrían resultar interesantes para abordar, o al menos ser tenidas en cuenta, en una investigación futura.

- Seguir la línea de ejecución de los *jumps* al momento de extraer los códigos de operación, ya que las instrucciones que se encuentren dentro de estos *loops* se ejecutarán múltiples veces y no sólo una.
- Explorar distintos valores de corte al momento de seleccionar los *features* mas relevantes del conjunto de datos.
- Promediar la longitud de las secciones en los archivos `ASM`, en lugar de sólo contar la cantidad de ocurrencias.
- Formar *n-gramas* con el contenido de los archivos `bytes`.
- Extender la clasificación incluyendo más familias de *malwares*.
- Entrenar los algoritmos de clasificación binaria con un *dataset* considerablemente mayor al utilizado.
- Poder garantizar que los archivos benignos sean realmente benignos.
- Contar con un *dataset* más equilibrados, en donde las cantidades de archivos que componen cada clase sen más parejas.
- Buscar nuevas técnicas que puedan reducir el *overfitting* para la clasificación binaria.
- Establecer un mejor ajuste de hiperparámetros para los algoritmos de clasificación binaria.
- Implementar una red neuronal profunda más compleja para la clasificación de las imágenes en escala de grises.

Bibliografía

[1] Tushar Sharma Dipanjan Sarkar Raghav Bali. *Practical Machine Learning with Python*. Bangalore, Karnataka, India: Apress, 2018.

[2] Christopher C. Elisan. *Advanced Malware Analysis*. Estados Unidos: Mc Graw Hill Education, 2015.

[3] Mark A. Hall y Christopher j. Pal Ian H.Witten Eibe Frank. *Data Mining. Practical Machine Learning Tools and Techniques*. Cambridge, Miami, Estados Unidos: Morgan Kaufmann, 2017.

[4] T. Hastie J. Gareth D. Witten y R. Tibshirani. *An Introduction to Statistical Learning*. New York: Springer, 2014.

[5] Matthew Kirk. *Thoughtful Machine Learning with Python*. Sebastopol, California, Estados Unidos: O'Reilly, 2016.

[6] H. Liu y M. Cocea. *Granular Computing based Machine Learning*. Suiza: Springer, 2018.

[7] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[8] Stuart j. Russell y Peter Norvig. *Artificial Intelligent: A Modern Approach, Thrid Edition*. Inglaterra: Pearson, 2010.

[9] Scikit-learn. *scikit-learn Tutorials*. https://scikit-learn.org/stable/tutorial/index.html. 2020.

[10] M. Sikorski y A. Honig. *Practical Malware Analysis*. San Francisco: No Starch Press, 2012.

[11] Mark Stamp. *Introduction to Machine Learning with Applications in Information Security*. Boca Raton, Florida: CRC Press, 2018.

[12] Tensorflow. *Tensorflow Tutorials*. https://www.tensorflow.org/tutorials. 2020.

[13] Yehezkel S. Resheff e Itay Lieder Tom Hope. *Learning Tensorflow*. Sebastopol, California, Estados Unidos: O'Reilly, 2017.

[14] José Unpingco. *Python for Probability, Statistics, and Machine Learning*. San Diego, California, Estados Unidos: Springer, 2016.

[15] Jake VanderPlas. *Python Data Science Handbook*. Sebastopol, California, Estados Unidos: O'Reilly, 2016.

[16] University of Virginia Computer Science. *x86 Assembly Guide*. http://www.cs.virginia.edu/~evans/cs216/guides/x86.html. 2018.

[17] Summet Dua y Xian Du. *Data Mining and Machine Learning in Cybersecurity*. Boca Raton, Florida, Estados Unidos: CRC Press, 2011.