# USO DE ALGORITMOS DE MACHINE LEARNING PARA LA DETECCIÓN DE ARCHIVOS MALWARE

SERGIO SANZ GARCÍA

MÁSTER UNIVERSITARIO EN CIBERSEGURIDAD
UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

Trabajo Fin Máster curso 2021/2022

23 de junio de 2022

Director:

Antonio Robles Gómez

# Autorización de difusión

Sergio Sanz García

23 de Junio del 2022

El abajo firmante, matriculado en el Máster Universitario en Ciberseguridad de la Escuela Técnica superior de Ingeniería informática, autoriza a la Universidad Nacional de educación a distancia (UNED) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Uso de algoritmos de machine learning para la detección de archivos malware", realizado durante el curso académico 2021-2022 bajo la dirección de Antonio Robles Gómez en el Departamento de Sistemas de Comunicación y Control , y a la Biblioteca de la UNED a depositarlo en el Archivo Institucional de la universidad con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

# Resumen en castellano

El siguiente proyecto de investigación aborda los problemas surgidos con el avance tecnológico y las consecuencias que ello provoca en la ciberseguridad. Con futuras amenazas cada vez más modernas, los actuales sistemas de detección como son los antivirus se encuentras obsoletos debido a que su tecnología en la detección de firmas no es eficaz a la hora de detectar estas nuevas amenazas. Para ello este proyecto de investigación abre la puerta al uso de algoritmos de machine learning para la detección de estas amenazas utilizando algoritmos que sean capaces de predecir la naturaleza de las archivos y así hacer frente a las nuevas y futuras amenazas.

En este proyecto se implementará varios desarrollos de algoritmos de machine learning, que sean capaces de predecir la naturaleza de un archivo dado, para ello se ha utilizado las características PE de los archivos ejecutables para generar una base de datos rica en features que nos servirá para el desarrollo de los modelos de los algoritmos.

En segundo lugar se utilizara las características almacenadas en la base de datos para generar modelos predictivos acorde con unos parámetros que se irán reajustando hasta obtener los resultados más óptimos en cada modelo.

En tercer lugar, se compararán los resultados obtenidos de los modelos para concluir cual de los algoritmos utilizados es el más efectivo para el problema propuesto en esta investigación.

Para finalizar se implementará un servicio web que permita realizar pruebas de manera online a los modelos generados por los diferentes algoritmos, unido a la implementación de funcionales extras como puede ser la impresión de los informes de clasificación de cada modelo.

## Palabras clave

Ciberseguridad, Machine learning, Árboles de decisión, Red Neuronal, Bosques aleatorios, Naive Bayes, K Nearest Neighbor

# Índice de Acrónimos

| Acrónimos | Significado | Comentario |
|---|---|---|
| Archivo PE | Portable Executable | Estructura de archivos ejecutables pertenecientes al sistema operativo Windows. |
| SAST | Static Application Security Testing | Herramienta de análisis estático que proporciona información de los problemas de seguridad y calidad del código fuente. |
| DAST | Dynamic Application Security Testing | Herramienta de análisis dinámico que ejecuta vulnerabilidades conocidas sobre una aplicación para detectar sus vulnerabilidades. |
| CI/CD | Continuous integration and continuous delivery | Método para distribuir las aplicaciones a los clientes con frecuencia mediante el uso de la automatización en las etapas del desarrollo de aplicaciones. |
| IDE | Integrated development environment | Entorno destinado al desarrollo y prueba de Código. |
| PSI | Printable String Information | Técnica de extracción de las cadenas imprimibles de muestras de malware para analizar sus ocurrencias. |
| Dword | Double word | Unidad de datos que es dos veces el tamaño de una palabra |

| Acrónimos | Significado | Comentario |
|---|---|---|
| Curva PR | Curva precision-recall | Métrica para medir la calidad de un modelo de aprendizaje automático |
| Curva ROC | Receiver Operating Characteristic | Característica Operativa del Receptor, Métrica para medir la calidad de un modelo de aprendizaje automático |
| FPR | False Positive Rate | Tasa de falsos positivos |
| FNR | False Negative Rate | Tasa de falsos negativos |
| TNR | True Negative Rate | Tasa de verdaderos negativos |
| TPR | True Positive Rate | Tasa de verdaderos positivos |
| GUI | Graphical User Interface | Interface gráfica de usuario |
| PaaS | Platform as a Service | Plataforma como servicio |
| IPS | Intrusion Detection System | Aplicación usada para detectar accesos no autorizados a un ordenador o una red. |
| IDS | Intrusion Prevention System | Software que detecta y elimina las posibles amenazas existentes de un sistema o equipo. |

# Abstract

The following research project addresses those problems that have arisen with technological progress and the consequences that this causes in cybersecurity. With modern future threats, current detection systems such as antiviruses are obsolete because their signature detection technology is not effective in detecting these new threats. Therefore, this research project opens the door to the use of machine learning algorithms to detect these threats using algorithms that are capable of predicting the nature of the files, facing new and future threats thus.

In this project, several developments of machine learning algorithms will be implemented. These are capable of predicting the nature of a given file, for which the PE characteristics of the executable files have been used to generate a database rich in features that will help us with the development of algorithm models.

Secondly, the characteristics stored in the database will be used to generate predictive models according to parameters that will be readjusted until the most optimal results are obtained in each model.

Third, the results obtained from the models will be compared to conclude which of the algorithms used is the most effective as a solution for the problem proposed in this research.

Finally, a web service that allows online testing of the models generated by the different algorithms, together with the implementation of extra functions such as the printing of the classification reports of each model, will be implemented.

## Keywords

Cybersecurity, Machine learning, Decision trees, Neural Network, Random Forests, Naive Bayes, K Nearest Neighbor

# Índice general


**Índice** I

**Agradecimientos** IV

**Dedicatoria** V

**Datos del Estudiante** VI

**1. Introducción** 1
- 1.1. Contexto y Justificación del trabajo . . . 1
- 1.2. Objetivos de la investigación . . . 3
- 1.3. Breve resumen del producto obtenido . . . 6
- 1.4. Breve descripción de los otros capítulos de la memoria . . . 7

**2. Planificación y costes** 9
- 2.1. Planificación del trabajo . . . 9
- 2.2. Inventario de hardware/software y costes . . . 11

**3. Estado del arte** 12
- 3.1. ¿Qué es un malware? . . . 12
- 3.2. Análisis de código . . . 14
  - 3.2.1. Análisis estadístico . . . 14
  - 3.2.2. Análisis Dinámico . . . 15
  - 3.2.3. Aprendizaje Automático . . . 16
- 3.3. Tipos de características . . . 19
- 3.4. Tipos de medición . . . 22
  - 3.4.1. Matriz de Confusión . . . 23
  - 3.4.2. Informe de clasificación . . . 24
  - 3.4.3. Curva PR y ROC . . . 26
- 3.5. Desequilibrio de clases . . . 28

**4. Metodología** 31
- 4.1. Construcción del Datasets . . . 31
- 4.2. Construcción de la Base de datos . . . 33
- 4.3. Herramientas de gestión . . . 36
- 4.4. Bibliotecas y Lenguajes de programación . . . 37
- 4.5. Algoritmos de Machine Learning . . . 38
  - 4.5.1. Redes neuronales . . . 39

4.5.2. Árboles de decisión . . . 41
4.5.3. Bosques Aleatorios . . . 43
4.5.4. K-Nearest Neighbors . . . 45
4.5.5. Naïve Bayes . . . 46

**5. Desarrollo del Proyecto** **49**
5.1. Preparación y tratamiento de datos . . . 49
5.2. Red Neuronal . . . 53
5.2.1. Matriz de Confusión . . . 57
5.2.2. Informe de clasificación . . . 59
5.3. Arboles de decisión . . . 60
5.3.1. Matriz de Confusión . . . 64
5.3.2. Informe de clasificación . . . 65
5.4. Bosque Aleatorio . . . 66
5.4.1. Matriz de Confusión . . . 70
5.4.2. Informe de clasificación . . . 73
5.5. K-Nearest Neighbor . . . 74
5.5.1. Matriz de Confusión . . . 77
5.5.2. Informe de clasificación . . . 79
5.6. Naive Bayes . . . 80
5.6.1. Matriz de Confusión . . . 82
5.6.2. Naive Bayes . . . 83

**6. Resultados y discusión** **84**
6.1. Resultado . . . 84
6.2. Discusión . . . 92

**7. Conclusión y trabajos futuros** **95**
7.1. Conclusión . . . 95
7.2. Trabajos Futuros . . . 97

**Bibliography** **101**

**A. Encabezados PE** **102**

**B. Guía de Usuario** **106**
B.1. Introducción . . . 106
B.2. Login de Usuario . . . 106
B.3. File Upload . . . 108
B.3.1. Descargar excel del dataset . . . 108
B.3.2. Subida de archivos al dataset . . . 109
B.4. Algoritmos de inteligencia artificial . . . 110
B.4.1. Generar un nuevo modelo . . . 111
B.4.2. Informe de Clasificación . . . 111

B.4.3. Análisis de archivos . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112

**C. Guía de Usuario, Ejecución en Local** **114**
C.1. Introducción . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
C.2. Requisitos e Instalación . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114

# Agradecimientos

Quiero agradecer a la Universidad Nacional de educación a distancia los recursos docentes y materiales proporcionados en el master de ciberseguridad, que han servido para la elaboración de esta investigación. También agradecer al equipo docente del master su gran labor en la enseñanza, en especial a Antonio Robles que ha sabido guiarme en la realización de esta investigación y cuyas directrices y mentoria han servido para obtener los resultados mostrados en esta memoria.

# Dedicatoria

Quiero dedicar esta investigación a mi familia en especial a mi tia Marisa, sin ellos esto no hubiera sido posible y a mi pareja, Sandra, por su apoyo y ayuda en todas las dificultades surgidas a lo largo del desarrollo de la investigación.

Sin vosotros este proyecto me hubiera costado mucho más.

Muchas gracia.

# Datos del estudiante


- Nombre: Sergio
- Apellidos: Sanz García
- DNI: 50994334E
- Email: ssanz227@alumno.uned.es

# Capítulo 1

# Introducción

## 1.1. Contexto y Justificación del trabajo

En este mundo cada vez más virtualizado, donde se está creando nuevos modelos económicos, personales y laborales, es indudable el gran impacto que los sistemas informáticos tienen en nuestra vida, ya sea de la manera de pasar nuestro tiempo, relacionarnos o trabajando.

No solo la tecnología está cambiando nuestra forma de vivir, si no también está abriendo nuevas puerta a nuestra vulnerabilidad. Los sistemas de protección utilizados anteriormente para protegernos de amenazas virtuales se están viendo ineficaces ante el gran avance tecnológico de las amenazas cibernéticas, poniendo en grave peligro nuestras vidas que tan inocentemente dejamos en las nuevas tecnologías.

Actualmente los sistemas de detección de malwares que disponemos para proteger el nuevo modelo económico y social virtualizado se basan en un sistema de detección por firmas donde se comparan los hash de los archivos analizados con los hash almacenados en una base de datos.

Este sistema es ineficaz para archivos de malware futuros, donde los hash de dichos archivos no se encuentran en estas base de datos, además de ellos, muchos autores de malware

ofuscan o alteran dichos hash para evitar la detección de estos sistemas de detección (*Sukwong et al. 2011*).

En el ultimo año se han detectado más de 1312 millones de malware (figura 1.1) de los cuales más de 173 millones son nuevas variantes, tal y como se puede apreciar en la siguiente figura 1.2. Con los datos proporcionados por AV Test, se puede pronosticar que este año 2022 la cifra de nuevas variantes de amenazas se incrementa respecto al año anterior.

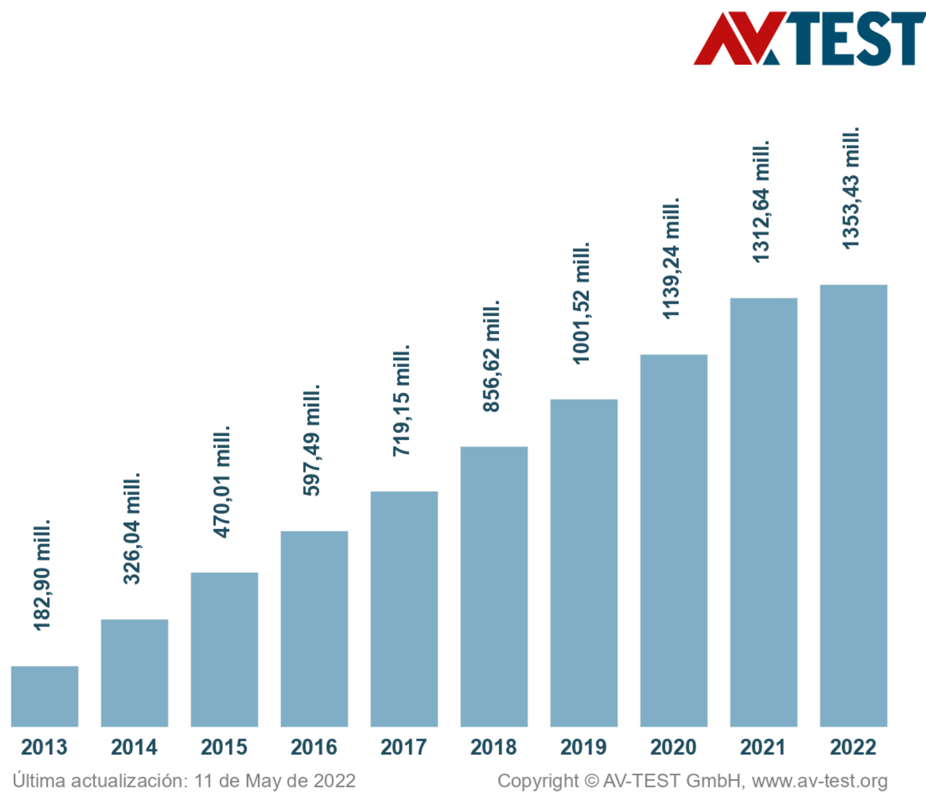



**Figura 1.1**: *Número total de archivos maliciosos detectados desde 2013. Fuente: https://www.av-test.org/es/estadisticas/software-malicioso/*

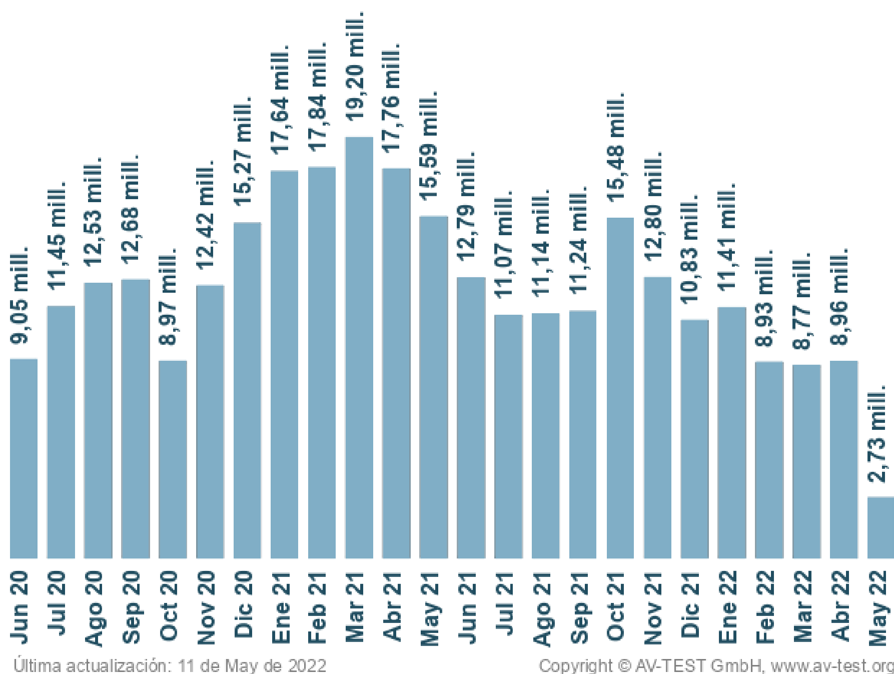



**Figura 1.2**: *Número total de nuevos archivos maliciosos detectados desde junio del 2020. Fuente: https://www.av-test.org/es/estadisticas/software-malicioso/*

## 1.2. Objetivos de la investigación

El siguiente proyecto de investigación aborda el estudio del uso de algoritmos automáticos en el campo de la ciberseguridad, contribuyendo a la implementación de estos para la detección de amenazas ejecutables en sistemas de ámbito local. Esta investigación no abordara la detección de amenazas en redes internas o privadas y se limitara a sistemas locales, como una mera actualización de las propias herramientas antivirus del mismo.

La siguiente investigación estará comprendida en varias fase. La primera fase estará compuesta por la recopilación de muestras de archivos benignos y malignos de los cuales se recopilara los datos de cabecera PE (*Portable Executable*) que serán los que se utilizaran

en las fases posteriores. Además de ello, esta fase incluye la virtualización de los datos de cabecera, trasformando estos datos hexadecimales a datos numéricos tratables por los algoritmos automáticos.

La segunda fase de la investigación comprende el estudio de la literatura publicada hasta el momento para determinar los algoritmos más eficaces a la resolución de los objetivos de este proyecto. Una vez obtenido un listado con los algoritmos, se procede a la implementación de estos utilizando los datos recopilados de la fase anterior. Esta fase incluye el estudio de las diferentes configuraciones que deberá tener cada algoritmo para obtener los resultados óptimos.

La tercera fase de la investigación compara los resultados obtenidos de los algoritmos automáticos, para determinar cuál es el modelo estadístico generado, más efectivo y óptimo para la resolución del problema planteado en esta investigación.

La última fase del proyecto engloba el desarrollo del sistema web, que usa los modelos generados por los algoritmos automáticos, pertenecientes a las fases anteriores de la investigación, para la detección de archivos maliciosos, tanto de manera individual como colectiva. Además de ello el sistema cuenta con un servicio que permite enriquecer la base de datos de manera automática permitiendo cargar archivos PE de manera masiva. El sistema también permite generar nuevos modelos predictivos de los diferentes algoritmos automáticos mencionados en esta investigación.

Los objetivos parciales del proyecto engloban:

- La implementación de seis modelos estadísticos de aprendizaje automático, de diferentes algoritmos de machine learning, con unos índices de precisión que superen el 80 %, para determinar que dichos modelos cuentan con unos niveles de calidad aceptables, para ser aplicados como solución al problema de clasificación.

- Estudio de la literatura publicada del uso de algoritmos de machine learning en el ámbito de la ciberseguridad, con el objetivo de extender dichas investigaciones a un ámbito más práctico.

- Análisis de las diversas características de los archivos recopilados, para determinar cuál de ellas son más efectivas a la hora de implementar modelos más eficaces.

- Análisis de los diversos algoritmos de machine learning, utilizados en otras investigaciones, para definir cuáles son los más óptimos para aplicar una solución al problema planteado de clasificación.

## 1.3. Breve resumen del producto obtenido

La investigación descrita en este documento proporciona un estudio de los diferentes algoritmos de clasificación que existen en la actualidad y su uso en el ámbito de la ciberseguridad, a la hora de clasificar archivos ejecutables del sistema operativo Windows en dos categorías, malware o benigno. Además del estudio de dichos algoritmos y su aplicación en la ciberseguridad se ha implementado diferentes modelos estadísticos de los siguientes algoritmos de machine learning de clasificación más comunes; Redes neuronales, Árboles de decisión, Bosques Aleatorios, K Nearest Neighbor y Naive Bayes. Estos modelos respaldan el estudio de esta tecnología en el campo, aportando valor a la investigación a través de los resultados obtenidos.

Esta investigación ha recorrido todo el flujo de aprendizaje necesario para llegar a implementar los modelos, desde la creación del dataset que utiliza los algoritmos, a través de la obtención de las características de cabecera PE de los archivos recopilados, de las diversas fuentes de información, y la virtualización de estos, hasta la implementación y ajuste de los parámetros de los diversos algoritmos.

Esto ha dado como resultado la creación de un sistema web al alcance del público en general*, que permite el uso de los modelos generados por los algoritmos mencionados, para determinar la naturaleza de un determinado archivo subido al sistema. Además de ello el sistema permite genera matrices de confusión con las precisiones de los modelos de cada algoritmo, incluso la generación de nuevos modelos y la incorporación de archivos PE para enriquecer la base de datos del sistema. Se adjunta dentro de la memoria el apéndice B, con la guía de usuario del sistema.

*https://machinelearninganalyzer.herokuapp.com/

## 1.4. Breve descripción de los otros capítulos de la memoria

En los próximos capítulos se hablara de los siguientes contenidos.

- Capitulo 2. Planificación y costes. Donde se muestra la planificación llevada a cabo para el desarrollo de esta investigación, al igual que los costes acarreados para su desarrollo y finalización.
- Capitulo 3. Estado del arte. En este capítulo se explica los conceptos básicos necesarios para comprender la investigación. Es un capítulo teórico destinado a mostrar los diferentes métodos de análisis existentes utilizados en el ámbito de la ciberseguridad, las diferentes características de los archivos utilizadas en la literatura y las métricas más comunes para evaluar modelos probabilísticos de los algoritmos de machine learning.
- Capitulo 4. Metodología utilizada. Se expone la metodología seguida para el desarrollo de la investigación. Este capítulo engloba aspectos técnicos del proyecto, algoritmos de machine learning, arquitectura de la base de datos, herramientas utilizadas, entre otros...
- Capitulo 5. Desarrollo del proyecto. Se muestra como se ha llevado a cabo la investigación, la creación del datasets, la implementación de los diferentes algoritmos y los modelos obtenidos.
- Capitulo 6. Resultados. En este capítulo se presenta los resultados obtenidos de cada modelo, realizando una discusión sobres las diversas métricas conseguidas y comparando cada resultados con el resto de los modelos elaborados.
- Capitulo 7. Conclusión y Trabajos Futuros. Se presenta la conclusión de la investigación y las diversas vías que este trabajo aporta para proyectos futuros.

- Capitulo 8. Repositorio y datos de la aplicación. Capitulo recopilatorio de la parte técnica de la investigación, donde se muestra el repositorio con el código, requisitos del sistema y datos del mismo.
- Bibliografía. Expone la literatura utilizada para elaborar la siguiente investigación.
- Apéndice A. Encabezados PE. Expone el listado de características, utilizadas en esta investigación, que compone la secciones del encabezado PE de los muestras utilizadas.
- Apéndice B. Guía de usuario del servidor. Guía de usuario del sistema web implementado.
- Apéndice C. Guía de usuario del servidor, ejecución en local

# Capítulo 2

# Planificación y costes

## 2.1. Planificación del trabajo

El siguiente capitulo muestra la planificación inicial de la investigación, indicando aproximadamente las fecha iniciales y finales de cada fase del proyecto. Para ello se ha utilizado un diagrama Gantt que muestre gráficamente las diferentes fases en las que se componen la investigación y en la que el proyecto ha ido pasando.

| Fases | Fecha de inicio | Fecha Final |
|---|---|---|
| Fase 1 | 01/06/2022 | 05/06/2022 |
| Definición de los objetivos | 01/06/2022 | 02/06/2022 |
| Organización del proyecto | 02/06/2022 | 03/06/2022 |
| Preparación del proyecto | 03/06/2022 | 04/06/2022 |
| Requisitos del proyecto | 04/06/2022 | 05/06/2022 |
| Fase 2 | 06/06/2022 | 19/06/2022 |
| Investigación de la literatura publicada | 06/06/2022 | 12/06/2022 |
| Análisis de los tipos de características | 13/06/2022 | 16/06/2022 |
| Análisis de los tipos de medición | 17/06/2022 | 19/06/2022 |

| Fases | Fecha de inicio | Fecha Final |
|---|---|---|
| Fase 3 | 20/06/2022 | 30/06/2022 |
| Planteamiento de la metodología | 20/06/2022 | 24/06/2022 |
| Análisis de las herramientas tecnológicas | 24/06/2022 | 25/06/2022 |
| Análisis de los algoritmos de machine learning | 26/06/2022 | 30/06/2022 |
| Fase 4 | 01/07/2022 | 31/08/2022 |
| Creación del dataset | 01/07/2022 | 10/07/2022 |
| Desarrollo de la Red Neuronal | 11/07/2022 | 18/07/2022 |
| Desarrollo de la Árboles de decisión | 19/07/2022 | 26/07/2022 |
| Desarrollo de la Bosques Aleatorios | 27/07/2022 | 03/08/2022 |
| Desarrollo de la K-Nearest Neighbors | 04/08/2022 | 11/08/2022 |
| Desarrollo de la Naive Bayes | 12/08/2022 | 21/08/2022 |
| Refactorización y pruebas | 22/08/2022 | 31/08/2022 |
| Fase 5 | 01/09/2022 | 25/09/2022 |
| Comparación de resultados | 01/09/2022 | 14/09/2022 |
| Conclusión y lineas futuras | 15/09/2022 | 25/09/2022 |

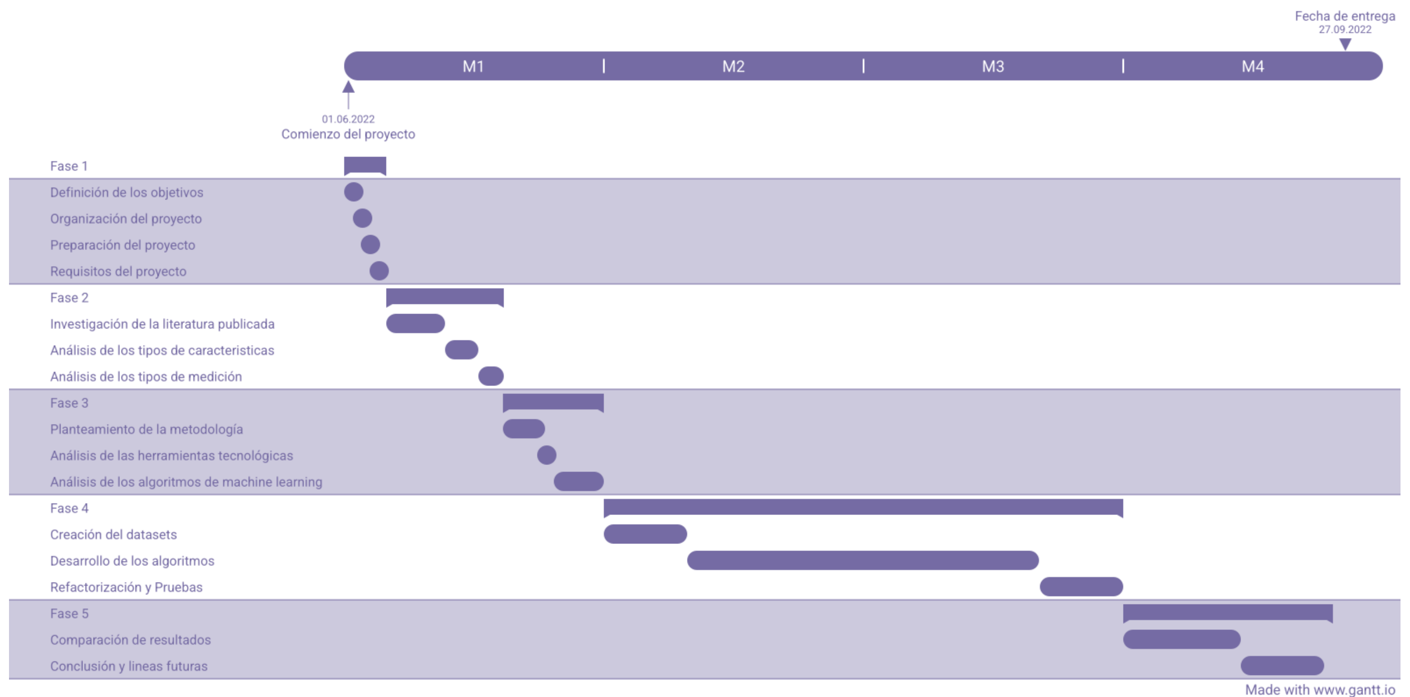



**Figura 2.1**: *Diagrama Gantt del proyecto*

## 2.2. Inventario de hardware/software y costes

Esta investigación no hace uso de ningún elemento de hardware para su desarrollo. La siguiente tabla muestra el inventario del Software utilizado.

| Nombre | Características | Versión |
|---|---|---|
| Python | Lenguaje de programación utilizado para la implementación de la investigación | 3.8.8 |
| Postgresql | Herramienta de base de datos | 5.0 |
| Visual studio code | Software de edición de código | 1.71.2 |
| Fork | Herramienta de control de versiones | 1.77.0.0 |

La siguiente tabla muestra las horas empleadas en el desarrollo de esta investigación. La fase 4 y fase 5 correspondientes a la creación de los modelos de aprendizaje automático y comparación de los resultados obtenidos, fueron las que más tiempo han llevado en esta investigación, debido a que corresponde a toda la parte esencial de esta.

| Descripción Tarea | Horas | Total servicio |
|---|---|---|
| Fase 1 | 30 horas | 600 € |
| Fase 2 | 41 horas | 820 € |
| Fase 3 | 45 horas | 900 € |
| Fase 4 | 160 horas | 3200 € |
| Fase 5 | 24 horas | 480 € |
| Total | 366 | 6000 € |
| Precio x hora | 20 € | |

El coste total de la implementación de la investigación asciende a 6.000€.

# Capítulo 3

# Estado del arte

## 3.1. ¿Qué es un malware?

Un malware es un software malicioso que infecta la máquina host sin el conocimiento del propietario, llevando a cabo actividades no autorizadas, por ejemplo el robo de datos o el daño al equipo portador [9].

La siguiente tabla muestra los diferentes tipos de malware clasificados actualmente.

| Tipos de malware | Comportamiento |
|---|---|
| Virus | Software que incrusta código malicioso en el sistema |
| Gusano | Software capaz de replicarse afectando al rendimiento del sistema y el colapso de la red |
| Troyano | Permite el acceso de manera ilícita al sistema |
| BackDoor | Abre una puerta trasera en el sistema permitiendo el acceso del mismo |
| Rootkits | Esconde los procesos y archivos que permiten al intruso mantener el acceso al sistema |

| | |
|---|---|
| KeyLoggers | Roba información a través de la captura de los datos de entrada y salida de los periféricos |
| Botnets | Infección de sistemas operativos para su uso en ataques masivos tanto tecnológicos (Ataques DDoS) como fraudulentos (Phishing) |
| Stealers | Software de tipo Troyano que infecta el sistema con el objetivo de obtener información confidencial del sistema. |
| Hijackers | Secuestra programas para manejarlos a voluntad del ciberdelincuente. |
| Adware | muestra publicidad en el sistema. Redirecciona las búsquedas del navegador a sitios web de publicidad y recopila datos comerciales acerca de ti. |
| Spyware | recopila información del usuario, uso del sistema y hábitos de navegación. |
| Ransonware | Encripta los contenidos almacenados en el sistema solicitando una cantidad monetaria para su desencriptado. |
| Rogueware | Envía notificaciones falsas al usuario, informando de una supuesta vulnerabilidad del sistema, ofreciendo a cambio de un importe herramientas de protección. Estas herramientas suelen ser otro tipo de malwares. |

| | |
|---|---|
| ScareWare | Muestra ventanas emergentes en el navegador haciéndose pasar por empresas licitas para conducir al usuarios a sitios web infectados con otros malwares. |

Estos software van evolucionando según se va avanzando tecnológicamente, con el objetivo de vulnerar los sistemas de seguridad actuales y con el agravante de ser capaces de producir un mayor grado de daño a los equipos portadores, además de ser capaces de replicarse y extender su alcance de infección.

## 3.2. Análisis de código

Los nuevos sistemas antivirus emplean controles heurísticos estáticos y análisis dinámicos, basados en reglas para detectar los malware en función de la estructura y el comportamiento del archivo a analizar.

Los expertos en ciberseguridad utilizan estos dos enfoques para analizar el malware; el análisis estático y el análisis dinámico.

### 3.2.1. Análisis estadístico

El análisis de código estático es una serie de pruebas automatizadas capaces de detectar errores y vulnerabilidades, comparando la información recibida en el escaneo con una serie de reglas previamente establecidas[26], por este motivo los análisis estadísticos solo pueden identificar los casos en los que se rompen las reglas programadas. También existe el riesgo de falsos positivos, por lo que es necesario interpretar los resultados.

Existe gran variedad de herramientas especializadas que utiliza esta clase de análisis para detectar las vulnerabilidad más comunes. Las pruebas estadísticas de seguridad de aplicaciones (SAST) realizan pruebas y análisis automatizados de código fuente, sin necesidad de

ejecutarlo, para detectar posibles vulnerabilidades. Dado que el análisis estático no requiere de la ejecución de los programas analizados, se puede analizar el código fuente al principio del proceso de CI/CD o directamente desde un IDE que pueda ofrecer informes detallados e inmediatos para la implementación de soluciones a los problemas localizados al finalizar dicho análisis.

Para mejorar la detección de vulnerabilidades, es más común el uso de herramientas de análisis estadístico al comienzo de la fase de detección, para localizar de manera automática todas las posibles amenazas y vulnerabilidades más conocidas, y complementar dichos resultados con un análisis dinámico del software.

### 3.2.2. Análisis Dinámico

El análisis de código dinámico es una serie de pruebas que se realizan en una aplicación cuando está en ejecución en busca de vulnerabilidades potencialmente explotables. Las herramientas DAST se utilizan para identificar vulnerabilidades tanto en tiempo de compilación como en tiempo de ejecución. [26]

Este análisis se realiza durante la fase de pruebas del ciclo de vida de desarrollo del software, una vez que el código de la aplicación se puede compilar e implementar en un entorno de prueba o ensayo, se usan las herramientas de pruebas de seguridad de aplicaciones dinámicas (DAST) con los flujos de trabajo CI/CD.

Las herramientas DAST utilizan un diccionario de vulnerabilidades conocidas y entradas maliciosas para testear una aplicación. Ejemplos de los casos que se incluyen, son los siguientes:

- Consultas SQL para identificar vulnerabilidades de inyección SQL
- Cadenas de entradas largas para explotar vulnerabilidades de desbordamiento de búfer.

- Números negativos y positivos grandes para detectar vulnerabilidades de desbordamiento y subdesbordamiento de enteros.
- Datos de entrada inesperados para explotar suposiciones no validas de los desarrolladores.

Tras ejecutar los casos de prueba contra la aplicación las herramientas DAST analizan su comportamiento. Si la aplicación es vulnerada la herramienta registra esa vulnerabilidad.

### 3.2.3. Aprendizaje Automático

El aprendizaje automático es un disciplina que ayuda a descubrir patrones automáticamente a partir de una gran cantidad de datos para predecir el resultado de observaciones desconocidas en función de un conjunto de patrones previamente identificados [18].

Existen varios tipos de algoritmos de aprendizaje automáticos clasificados en dos grupos:

- Algoritmos supervisados.
- Algoritmos No supervisados.
- Algoritmos semi supervisados.

#### Algoritmos Supervisados

Los algoritmos supervisados son aquello que el conjunto de entrenamiento contiene la solución dada [4]. Se pueden dividir en dos tipos diferentes, dependiendo de la finalidad con la que se utilicen, no son excluyentes ya que un mismo algoritmo puede utilizarse para ambas situaciones.

- Algoritmo de Regresión: Son los algoritmos que utilizan la solución incorporada en el conjunto de entrenamiento para predecir la solución de una nueva muestra dada.
- Algoritmo de Clasificación: Son algoritmos que utilizan la solución dada en el conjunto de enteramiento para clasificar la nueva muestra dada.

Ejemplos de los principales algoritmos supervisados:

- Árboles de decisión
- Clasificación de Naïve Bayes
- Regresión por mínimos cuadrados
- Regresión Logística

**Algoritmos No Supervisados**

Los algoritmos no supervisados son aquellos que aprenden sin necesidad de tener la solución dentro de su conjunto de entrenamiento [4].

Un ejemplo claro de algoritmos no supervisados son los algoritmos de visualización que se les proporciona una gran cantidad de datos complejos y sin etiquetar para generar una representación en 2D y 3D de los datos. Estos algoritmos intentan preservar la mayor cantidad de estructura posible para que pueda comprender cómo se organizan los datos y tal vez identificar patrones inesperados.

Las principales aplicaciones de los algoritmos de aprendizaje no supervisado son:

- Reducción de dimensionalidad: Cuyo objetivo es simplificar los datos sin perder la información. Para lograrlo se fusiona varias características correlacionadas en una sola.

- Detección de anomalías: El algoritmo obtiene instancias normales durante el entrenamiento, aprendiendo a reconocerlas, al incorporar una nueva instancia puede detectar si es normal o una anomalía.
- Aprendizaje de reglas de asociación: El algoritmo profundiza en grandes cantidades de datos y descubre relaciones interesantes entre atributos.

Ejemplos de los principales algoritmos no supervisados:

- K-Medias
- Clusterización Jerárquica
- Density Based Scan Clustering (DBSCAN)
- Modelo de Agrupamiento Gaussiano

**Algoritmos Semi Supervisados**

En algunas ocasiones se cuenta con un conjunto de datos parcialmente etiquetado. Los algoritmos semi supervisados es una combinación entre un algoritmo no supervisado y un algoritmo supervisado para tratar todo el conjunto de datos. Se puede utilizar algoritmos de agrupación para agrupar instancias similares y luego cada instancia sin etiquetar se puede etiquetar con las etiqueta más común en su clúster. Tras etiquetar todo el conjunto de datos se utiliza cualquier algoritmo de aprendizaje supervisado.

**Aprendizaje automático en la ciberseguridad**

Los actuales sistemas de detección de archivos maliciosos, que utilizan algoritmos de aprendizaje automático, aplican algoritmos de clasificación.

Los algoritmos de clasificación facilitan la construcción de clasificadores que aprenden automáticamente las características de cada archivo, malware o benigno, aprendiendo de datos etiquetados.

Esta clase de algoritmos suelen ser más eficaces ante las técnicas de ofuscación de los archivos maliciosos en comparación con los sistemas actuales de detección de firmas.

Los pasos a seguir para el uso de estos algoritmos de aprendizaje automático son los siguientes:

- Se debe crear un conjunto de datos etiquetados para el entrenamiento(registros).
- Cada registros contiene un conjunto de características (características específicas del archivo) y una etiqueta (malware o benigno).
- Se debe virtualizar las características de los registros (las etiquetas tendrán un valor binario, siendo 0 el archivo benigno y 1 el archivo malware).
- El clasificador genera un modelo matemático que mapea la relación de las características y las etiquetas del archivo prediciendo la clase de cada registro en el datos de prueba.
- Se evalúa la precisión del clasificador mediante el uso de métricas definidas.

## 3.3. Tipos de características

Las principales fuentes de datos que proporciona los archivos ejecutables y que se pueden utilizar en los algoritmos de aprendizaje automático para la detección de archivos maliciosos, son los siguientes:

- String n-grams: Es una técnica de minería de texto donde $n$ representa una cantidad de técnicas o caracteres consecutivos en una frase[13]. Estas cadenas se pueden encontrar

en ejecutables en forma de comentarios, URL, mensajes de impresión, nombres de funciones, comandos de importación de bibliotecas... Estas cadenas en ocasiones son exclusivas de archivos malwares que se pueden utilizar como identificadores de los mismos[16].

Otra de las técnicas que se utiliza de análisis de cadenas, para detectar archivos malwares, es la de información de cadenas imprimibles(PSI) que consiste en extraer las llamadas a funciones imprimibles de los archivos PE y analizar el número de ocurrencias para cada función[8].

- Encabezado PE: Encabezados PE: Es la estructura de archivos estándar, para el sistema operativo Windows, que encapsula toda la información que necesita el sistema, para que pueda cargar ese ejecutable en la memoria y ejecutarlo.

  Los archivos PE contiene secciones que cuentan con referencias a las bibliotecas DLL, necesarias para importar y exportar el código y los datos necesarios para ejecutar el archivo y los recursos necesarios.

  Algunas investigaciones han utilizado los datos extraídos de la cabecera NT(NT_HEADERS) para detectar la naturaleza del archivo analizado [11]

  La cabecera NT es la cabecera principal del encabezado PE que contiene los siguientes datos;

  1. La firma Dword y firma PE del archivo.
  2. El archivo de cabecera (FILE_HEADER) que contiene la información más básica sobre el diseño del archivo.
  3. Cabecera opcional: que contiene la información más critica del archivo.

Otros estudios como los de Michael Sikorski en *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*[16] especifican el uso de las bibliote-

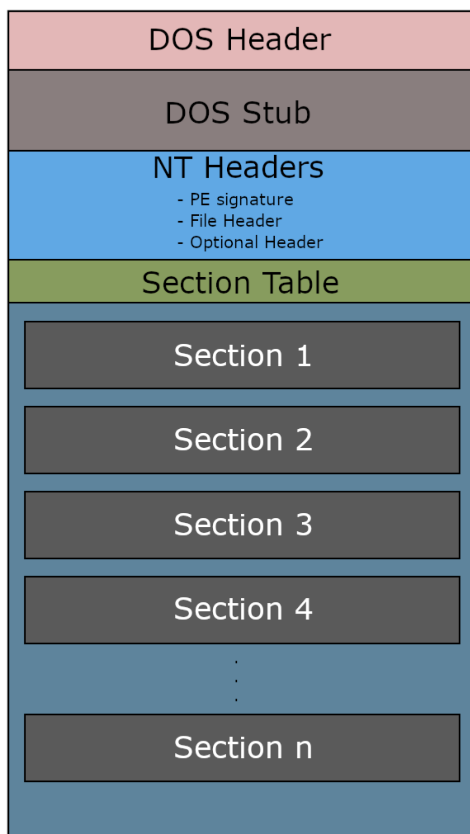



**Figura 3.1**: *Fuente: https://0xrick.github.io/win-internals/pe2/*

cas DLL, incorporados en la sección rdata del encabezado PE, y de las llamadas a las funciones DLL como datos significativos a la hora de detectar archivos malwares.

La utilización de la String n-grams, para determinar la naturaleza del archivos a analizar, ha logrado un éxito modesto con esta técnica [23] .

Sin embargo, también se ha argumentado que, tanto desde unas perspectiva teórica como empirica, las frecuencias de los n-gramas no difieren de los lo suficiente entre el software malicioso y el benigno como para ser útiles para la detección práctica de malware [25]. En otras palabras, el aprendizaje automático no diferencia de manera efectiva entre el malware y el software benigno al observar fragmentos aislados de código de máquina.

La utilización de cabeceras de PE para la detección de archivos de tipo Malware o benignos han resultado más eficaces que las características de n-grama. Estudios como el de 2012 de Joel Yonts[29] demuestran su eficacia, tras comparar las diferencias en ciertos componentes de cabecera entre un programa malicioso de uno benigno.

Yonts diseño unas reglas de detección que individualmente discrimino razonablemente bien entre los software malware y los benignos en su base de datos[29] pero no usa, las características en conjunto como lo haría un clasificador de aprendizaje automático. Otros estudios, como el Guanhua Yan[5], avalan la importancia de los datos de cabecera PE debido a la información que aportan y la facilidad de su extracción.

No hay que olvidar que el autor del archivo de malware podría ofuscar ciertos parámetros del encabezado PE, que permitan convertir en eficaces el uso de estas características. La mejor forma de evitar este escenario seria detectar la entropía de las secciones y las características del encabezado. Una alta entropía en el encabezado del archivo mostraría un encriptado en sus datos, la presencia de una sección con características de lectura, escritura y ejecución significaría un posible empaquetado de los datos.

Cuanto más agresivo sea la heurística de los algoritmos, más probable será que detecte malware, pero también de obtener muchos falsos positivos.

## 3.4. Tipos de medición

Para poder evaluar los modelos de clasificación creado por los algoritmos de aprendizaje automático, existen varias medidas de desempeño disponibles que se pueden utilizar para medir su calidad.

### 3.4.1. Matriz de Confusión

El objetivo de la matriz de confusión es contar las veces que las instancias de la clase A se clasifican como clase B, para todos los pares A/B. Para poder diseñar una matriz de confusión se debe tener un conjunto de predicciones con las que comparar con los objetivos reales del modelo. Como se muestra en la siguiente tabla, cada fila representa una clase real, mientras que cada columna representa una clase predicha.

**Figura 3.2**: *Fuente: https://www.juanbarrios.com/la-matriz-de-confusion-y-sus-metricas/*

Un modelo de clasificación perfecto tendría solo verdadero positivos y verdaderos negativos siendo su diagonal; falsos negativos y falsos positivos con valores 0.

Una matriz de confusión proporciona mucha información del modelo de clasificación evaluado, entre ellos la métricas de precisión y sensibilidad (Recall).

La métrica de precisión del modelo de clasificación es la precisión de las predicciones positivas. La precisión se obtiene con la división de los números de verdaderos positivos (TP) entre la suma de los mismos con los números totales de falsos positivos (FP).

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

Para obtener una precisión perfecta se debe de crear un clasificador que siempre haga predicciones negativas, excepto una sola predicción positiva en la instancia en la que tiene más confianza. Si la predicción es correcta, el modelo de clasificador contiene una precisión del 100 %. Pero este modelo de clasificador no sería efectivo ya que el modelo ignora todas las instancias positivas excepto una. Por lo tanto, la precisión generalmente se usa junto con la sensibilidad.

La métrica de sensibilidad muestra la proporción de instancias positivas que el modelo clasificador detecta correctamente. Esta métrica se obtiene a partir de la siguiente formula, siendo $FN$ el número total de falsos negativos :

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

### 3.4.2. Informe de clasificación

Otras de las métricas utilizadas para evaluar modelos de clasificación de algoritmos automáticos, son los informes de clasificación. Los Informes de clasificación muestran las métricas de precisión, sensibilidad, puntuación $F_1$ y soporte del modelo.

Como hemos mencionado en el apartado anterior, la métrica de precisión del modelo es la precisión de las predicciones positivas dentro del modelo y la sensibilidad muestra la proporción de instancias positivas que el modelo detecta correctamente. En ocasiones es conveniente combinar ambas métricas en una sola métrica, esta combinación es llamada puntuación $F_1$ y es utilizada para comparar modelos de clasificación.

La puntuación $F_1$ es la medida armónica de precisión y sensibilidad. Mientras que la

medida regular trata todos los valores por igual, la media armónica da mucho más peso a los valores bajos. Como resultado, el clasificador solo obtendrá una puntuación $F_1$ alta si tanto la recuperación como la precisión son altas.

$$F_1 = \frac{2}{\frac{1}{precision} + \frac{1}{sensibilidad}} = 2 \times \frac{precision \times sensibilidad}{precision \times sensibilidad} = \frac{TP}{TP + \frac{FN+FP}{2}}$$

La puntuación favorece a los clasificadores que tiene una precisión y sensibilidad similares. Ambas métricas están en un constante equilibrio, ya que, si se desea un modelo de clasificación con una alta precisión, el modelo tendrá una baja recuperación y de manera inversa, si se desea un modelo con una alta recuperación, el modelo tendrá una baja precisión de clasificación. Este equilibrio se conoce como equilibrio entre precisión/recuperación.

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 1.00 | 0.99 | 0.99 | 1005 |
| 1 | 0.99 | 1.00 | 1.00 | 2918 |
| accuracy | | | 1.00 | 3923 |
| macro avg | 1.00 | 0.99 | 0.99 | 3923 |
| weighted avg | 1.00 | 1.00 | 1.00 | 3923 |

**Figura 3.3**: *Informe de clasificación*

El informe de clasificación, mostrado en la figura 3.3, expone las métricas de precisión, sensibilidad, puntuación $F_1$ y soporte del modelo por cada clase, pero también de manera promedia para obtener un solo número capaz de describir el rendimiento general.

- El promedio micro (Micro avg) calcula una métrica de media global contando las sumas de los verdaderos positivos (TP), los falsos negativos (FN) y los falsos positivos (FP). En ocasiones a la hora de obtener el informe de clasificación esta métrica no se

representa debido a que el micro promedio esencialmente calcula la proporción de observaciones clasificadas correctamente de todas las observaciones, lo que comúnmente se utiliza para calcular la precisión general [10].

- El promedio macro (macro avg) se calcula utilizando la media aritmética(media no ponderada) de todas las métricas de cada clase, tratando a todas por igual independientemente del número de soporte que tenga cada clase.
- El promedio ponderado (weighted avg) se calcula tomando la media de todas las métricas por clase, tomando en cuenta el peso que tiene cada clase en el relación con la suma de todos los valores.

La columna de soporte (support) representa el número de ocurrencias reales de la clase en el conjunto de datos. Un ejemplo con la Figura 3.3 seria que para la clase negativa(0) solo hay 12500 observaciones con esa etiqueta.

### 3.4.3. Curva PR y ROC

Como se ha mencionado anteriormente, el equilibrio entre precisión y recuperación lleva al modelo generado a ser más precisión o con mayor de recuperación. La curva de precisión/recuperación(Curva PR) muestra gráficamente las puntuaciones calculadas basándose en una función de decisión. Si esta puntuación es mayor que el umbral de decisión o decision threshold, el modelo está aumentando su precisión a la hora de clasificar las muestras, en cambio sí está por debajo está disminuyendo su precisión y aumentando su recuperación [22].

La curva ROC es una herramienta utilizada para la medición de la calidad de los modelos de los algoritmos de clasificación, representando la tasa de verdaderos positivos(sensibilidad) junto con la tasa de falsos positivos(FPR).

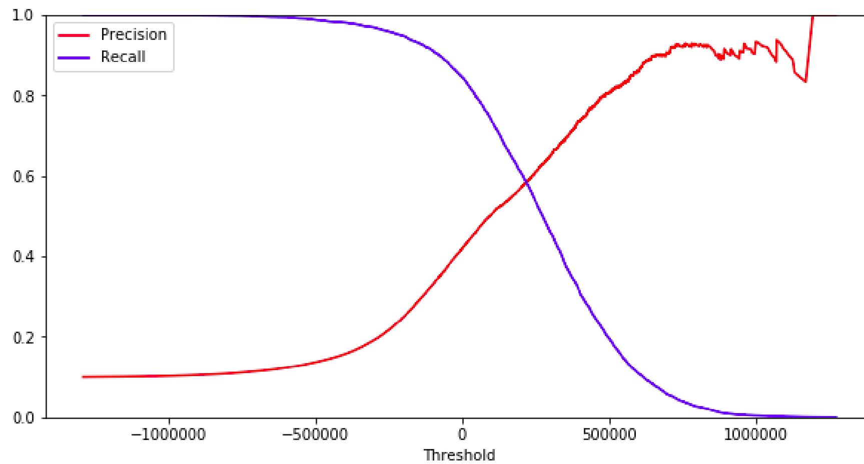



**Figura 3.4**: *Fuente: https://medium.com/bluekiri/curvas-pr-y-roc-1489fbd9a527*

- La tasa de falsos positivos (FPR) es la proporción de instancias negativas que se clasifican incorrectamente como positivas.
- la tasa de negativas verdaderas (TNR) es la proporción de instancias negativas que se clasifican correctamente como negativas(especialidad).

Cuanto mayor sea el sensibilidad (TPR) más falsos positivos (FPR) produce el clasificador. La línea puntera representa la curva ROC de un clasificador puramente aleatorio; un buen clasificador se mantiene lo más lejos posible de esa línea (hacía la esquina superior izquierda).

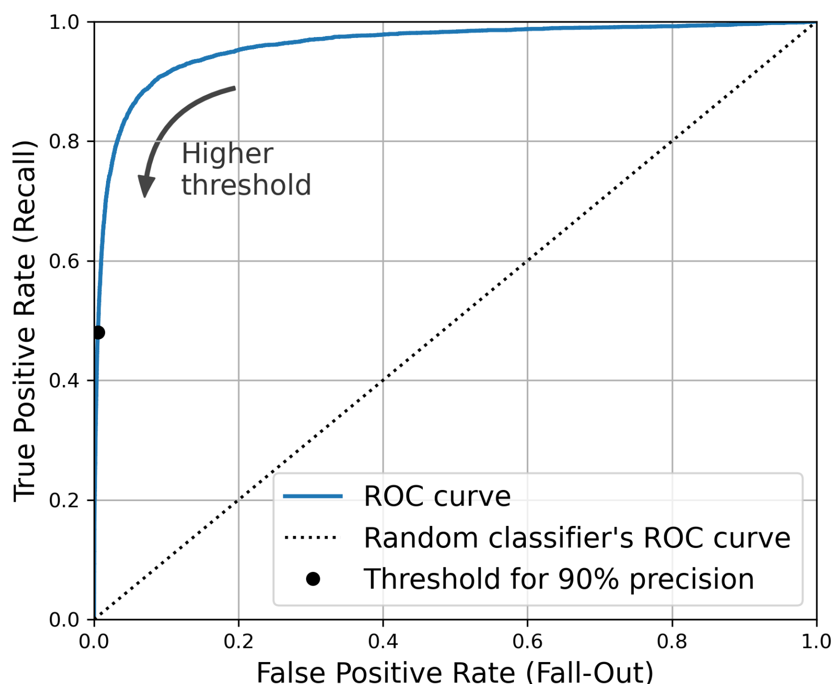



**Figura 3.5**: *Fuente: Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*

La forma de comparar modelos de clasificador utilizando la curva ROC, es medir el área bajo la curva (AUC). Un clasificador perfecto tendrá un AUC igual a 1, mientras que un clasificador puramente aleatorio tendrá un AUC igual a 0,5.

Por lo general se usa la curva PR en modelos con muestras no balanceadas. Con esta clase de modelos la curva ROC puede representar un modelo con métricas con valores altos y la curva PR con margen de mejora, indicando la baja probabilidad en una de las clases.

## 3.5. Desequilibrio de clases

Muchos estudios han construido clasificadores de aprendizaje automático que han logrado un alto rendimiento en experimentos en entornos de laboratorio, pero tras ejecutarlos en entornos reales han sido ineficaces. Esto es debido a que los experimentos utilizados han usado grandes cantidades de archivos malware y benignos en los datos de entrenamiento de los clasificadores, llegando a ser equitativos. En un entorno real nos encontramos con un

desequilibrios en los datos, donde hay más datos benignos que malignos, este desequilibrio de clases disminuye el rendimiento del clasificador.

Los estudios iniciales sobre los desequilibrios de clases como el de Guanhua Yan en 2008 [5] especifican las técnicas Boosting y Bagging para el éxito de algoritmos de clasificación con desequilibrios de clases.

Actualmente para enfrentarse a este problema existen dos posibles aproximaciones. El primero será integrar soluciones que se centren en el algoritmos, ya sea modificando su arquitectura como a través del ajuste de hiperparametros para inducir mejores ajustes en datos desbalanceados. El segundo sería modificar la estructura de datos originales modificando el número de instancias que se incluyan en el estudio (técnicas de balanceo de datos).

Las siguientes soluciones que integran las técnicas mencionadas son:

- Refuerzo de aprendizaje en las clase minoritaria o mayoritaria: Una solución bastante eficaz para problemas en los que realmente es difícil recoger o incluir instancias de una de las clases, como son los algoritmos de clasificación binaria.

  Estos algoritmos están enfocados como soluciones adaptativas sin abordar ninguna modificación de los datos originales. Lo que hacen es introducir un sesgo para balancear las clases, lo que implica una ponderación para otorgar más importancia a la clasificación correcta en la clase minoritaria, como desarrollaremos en este trabajo, dando más peso a los archivos malware dentro de los datos de entrenamiento.

  El siguiente ejemplo es un fragmento de código del proyecto donde se le ha dado un peso de 2.2 a la etiqueta 1, representando a los archivos malware. Este peso se da tras dividir la totalidad de las muestras de archivos benignos con la totalidad de los archivos malignos.

```
tree_model = tree.DecisionTreeClassifier(criterion='entropy',
                                         min_samples_split=20,
                                         min_samples_leaf=5,
                                         max_depth = depth,
                                         class_weight={1:2.2})
```

- Técnicas de balanceo de datos o Técnicas de resampling: Son soluciones que inciden directamente sobre los datos, y modificando las distribuciones de alguna de las clases. Estas técnicas consisten en eliminar parte de las instancias de la clase mayoritaria (undersampling) para igualar el número de instancias a utilizar o replicar las instancias de la clase minoritaria (oversampling).

  Un ejemplo de la técnica en la clase mayoritaria, undersampling, seria este, donde utilizamos el algoritmo NearMiss, un algoritmo muy similar a K-nearest, permite equilibrar el conjunto de datos desequilibrado, reduciendo la clase mayoritaria:

  ```
  model = NearMiss(ratio=0.5, n_neighbors=3, version=2, random_state=1),
      X_train_res, y_train_res = model.fit_sample(X_train, y_train)
  ```

  Un ejemplo con la técnica oversampling, seria utilizando la clase RandomOverSampler que permite duplicar las instancias minoritarias de un datasets. En este ejemplo se declara por porcentaje para asegurar que la clase minoritaria sea sobremuestreada para tener la mitad del número de instancias que la clase mayoritaria:

  ```
  model =  RandomOverSampler(ratio=0.5)
      X_train_res, y_train_res = model.fit_sample(X_train, y_train)
  ```

# Capítulo 4

# Metodología

## 4.1. Construcción del Datasets

Se genera un dataset con diversas muestras, recopiladas a través de diferentes repositorios de internet cuyos objetivos son similares a los expuestos en este mismo trabajo [2] [1]. Se logra alcanzar el siguiente tamaño de muestras con las que se trabajara en esta investigación:

- Benignos: 5012
- Malignos: 14599

Como se ha explicado anteriormente, en el capítulo 3 apartado 3.3, hay diversas fuentes de información que un archivo nos puede ofrecer para poder identificar si es malware o benigno. Nos centraremos en utilizar la información del encabezado PE, basándonos en los resultados de la tesis doctoral de Samuel Kim,*PE Header Analysis for Malware Detection*[11].

Con el objetivo de aumentar la precisión y la exactitud de los resultados obtenidos y con el fin de recopilar datos de mayor calidad y orientar el desarrollo a las características que podrían ser útiles para el modelo [6]. Se opta por un desarrollo de conocimiento completo utilizando las primeras cabeceras que forman el encabezado PE; la cabecera DOS, la cabecera PE, la cabecera opcional y la cabecera de sección. El valor extraído se usa como una característica, formando un dataset de 77 características diferentes (ver Apéndice A).

Hay una serie de programas y bibliotecas que analizan las cabeceras de los archivos ejecutables, como PortEx, profile y objdump. En esta investigación se utiliza la biblioteca pefile de python para extraer los primeros encabezados que engloban la cabecera PE.

Tras recopilar archivos malware y benignos a través de diversas fuentes de internet, se desarrolla una serie de métodos que permitan cargar todos los archivos de manera máxima dentro de la base de datos, el método *uploadFile* permite coger los archivos insertados y cargarlos en el servidor de manera individual para su tratamiento.

Una vez cargado el archivo se procede a usar la biblioteca pefile de python para obtener las cabeceras expuestas en el apéndice A de esta misma investigación, para posteriormente virtualizar dichos elementos y utilizar un procedimiento de almacenado para su inserción en la base de datos. Una vez insertado se procede a eliminar el archivo cargado en el servidor y cargar el siguiente.

Esta funcionalidad se ha implementado en el servicio para facilitar la subida de archivos de manera máxima (ver apartado B.3.2, del apéndice B).

Se realiza una división del total de instancias almacenadas en la base de datos, el 20 % de las instancias almacenadas se utiliza como datos de pruebas, para determinar la precisión del modelo generado por los diferentes algoritmos de machine learning, el 80 % restante de las instancias, como datos de entrenamiento para entrenar los modelos generados.

El siguiente ejemplo es un método perteneciente al código de esta investigación, donde se utiliza la biblioteca *train_ test_ split* de sklearn para generar los conjuntos de entrenamiento y los de test.

```
def data_Preparation(datasets):
    y = datasets['type']
    x = datasets.drop(['type'], axis=1).values
    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2,
     random_state=0)
```

## 4.2. Construcción de la Base de datos

Se crea una base de datos que permita almacenar todo el dataset en diversas tablas. Se escoge el sistema de gestión de base de datos Postgresql, debido a su compatibilidad con el resto de las herramientas y servicios en esta investigación, principalmente por su compatibilidad con el lenguaje de programación Python y el servidor Heroku, también utilizado en este proyecto.

El tipo de base de datos que se utiliza es un modelo de base de datos relacional bajo la tercera forma normal, donde se cumple los siguientes principios:

- Principio de atomicidad: cada columna contiene un solo dato.
- Individualidad de cada tuplas: Cada tabla está diseñada para evitar grupos repetidos en su contenido.
- Eliminación de las dependencias parciales: Cada columna solo depende de la clave primaria (PK)
- Eliminación de las dependencias transitiva: para evitar problemas en el futuro y eliminar la dependencia que hay con el tipo de la muestra (malware o benigno), se procede a diseñar la tabla Type_file que contenga estos datos.

La base de datos se compone por seis tablas relacionales donde cada columna que conforman dichas tablas, a excepción de la columna ïs_admin"que es de tipo Bool, son de tipo Çharacter"para incorporar los datos decimales de cada parámetro del encabezado.

- Login_users: Esta tabla almacena los datos de registro de cada usuario (correo, contraseña, nombre y si es administrador). Es la tabla principal del servicio, ya que controla los acceso al mismo.
- Type_file: Es una de las cinco tablas que componen el datasets(Type_file, Dos_header, Pe_header, Optional_header, Section_header) y que utiliza los algoritmos de machine learning, Esta tabla almacena el tipo de archivo que es cada muestra, siendo maligno o benigno, junto con el nombre de la muestra como PK.
- Dos_header: Esta tabla almacena todos los datos pertenecientes a los valores de encabezado DOS de cada muestra, la PK utilizada es el nombre de la muestra dentro del encabezado PE.
- Pe_header: Esta tabla almacena los valores pertenecientes al encabezado principal PE de cada muestra, la PK que se utilizada, al igual que con el resto de las tablas que compone el datasets, es el nombre de la muestra obtenido del encabezado PE.
- Optional_header: En esta tabla se almacena los valores del encabezado opcional de cada muestra, al igual que el resto de las tablas la PK se compone del nombre obtenido del encabezado PE de la muestra.
- Section_header : En la siguiente tabla se almacena los valores pertenecientes al encabezado de sección de cada muestra, estos valores, al igual que los valores de la cabecera opcional, se obtienen directamente de la cabecera PE, la PK de la tabla la compone el nombre de la muestra obtenido del encabezado de la misma.

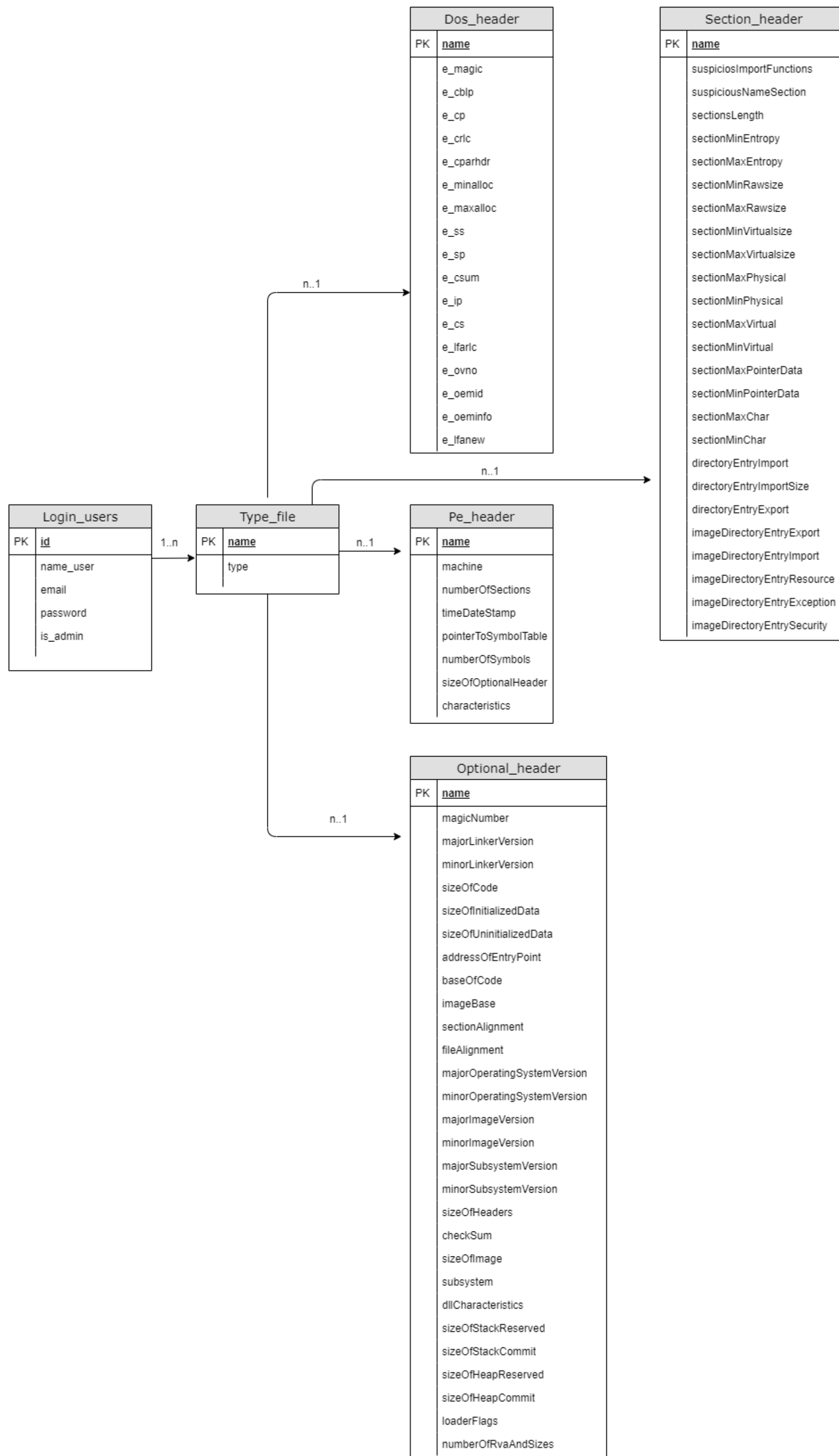



**Figura 4.1**: *Modelo relacional de BBDD*

## 4.3. Herramientas de gestión

Para desarrollar la siguiente investigación, se ha requerido del uso de múltiples herramientas de terceros que han permitido crear de manera más cómoda toda la parte de programación.

El siguiente listado muestra y detalla el uso de las múltiples herramientas utilizadas:

- Visual Studio Code (https://code.visualstudio.com/): Es un editor de código abierto multiplataforma y con intellisence que permite el desarrollo y la depuración de cualquier código de lenguaje de programación de manera más rápida y productiva. Además, Visual Studio Code cuenta con una compatibilidad con el sistema de gestión de versiones Git, pudiendo realizar commits, push y pull de manera más cómoda.

- Github (https://github.com/): es una plataforma colaborativa, con un sistema de control de versiones, llamado Git, y basado en la nube para desarrollos de software. El sistema de gestión de versiones integrado en el servicio almacena el código fuente de un proyecto y rastrea el historial completo de todos los cambios realizados en ese código. Además, GitHub permite a los desarrolladores cambiar, adaptar y mejorar el software de sus repositorios de forma gratuita. Cada repositorio contiene todos los archivos de un proyecto, así como el historial de revisiones de cada archivo.

- Fork (https://fork.dev/): Es una interfaz gráfica de usuario (GUI) multiplataforma que permite sustituir la línea de comandos que se realiza para la subida de versiones y cambios del código del sistema de control de versiones, por una interfaz amigable. Además de esto Fork permite ver de manera gráfica las bifurcaciones de versiones, teniendo más control sobre el avance del proyecto.

- Heroku (https://machinelearninganalyzer.herokuapp.com/): Es un entorno de desarrollo e implementación completo en la nube (PaaS), con recursos que permiten implementar, administrar y escalar aplicaciones. Este entorno permite implementar apli-

caciones simples de manera gratuita con recursos limitados, y es totalmente compatible para aplicaciones desarrolladas con el lenguaje de programación Python.

## 4.4. Bibliotecas y Lenguajes de programación

El resultado de esta investigación finaliza con la implementación de un servicio web que permita la detección de posibles archivos malware y la creación de modelos predictivos capaces de poder descubrirlos.

Se utiliza el lenguaje de programación HTML5, junto con la biblioteca Bootstrap, un lenguaje de programación de tipo frontend, para implementar la parte de interacción con el usuario, siendo el elemento visual del servicio, capaz de interactuar entre el usuario y el servidor.

Para el desarrollo de la parte backend del proyecto, se opta por el lenguaje de programación Python, debido a que este lenguaje cuenta con bibliotecas enfocadas a los algoritmos de machine learning, data science y Big data, necesarios para el desarrollo de esta investigación.

las principales bibliotecas utilizadas en esta investigación son las siguientes:

- Pefile: Es un módulo de Python que permite analizar y trabajar con las cabeceras de los archivos PE. Este módulo se utiliza en esta investigación para recopilar los datos de cabecera de los archivos que componen el dataset y que se utiliza para el entrenamiento y prueba de los algoritmos de machine learning.

- Scikit-learn: Es un módulo de Python basado en la tecnología de bibliotecas como pandas y NumPy, que proporciona los algoritmos de machine learning que se van a utilizar a lo largo de esta investigación.

- Panda: Es un módulo de Python basado en Numpy, que permite el tratamiento, limpieza y visualización de los datases enfocados a tareas de data science y machine learning.
- TensorFlow: es un módulo matemático de python que permite hacer cálculos numéricos enfocados a algoritmos machine learning. Utiliza un flujo de datos y programación diferenciable para realizar diversas tareas enfocadas al entrenamiento e inferencias de redes neuronales profundas.

## 4.5. Algoritmos de Machine Learning

La literatura publicada hasta el momento sobre el uso de algoritmos de machine learning para la detección de archivos malware, concluyen en el uso de determinados algoritmos más efectivos que otros para dicha tarea. Basándonos en los resultados obtenidos por Zane A. Markel en *Machine Learning based Malware Detection* [15] podemos determinar que los algoritmos de árboles de decisión son más efectivos en la detección de malwares que los algoritmo de Naive Bayes. Otros de los proyectos en lo que se basa esta investigación es el de Samuel Kim, *PE Header Analysis for Malware Detection* [12], en el cual los resultados de su investigación con los Algoritmos de Bosques aleatorios y K-Nearest Neighbors obtuvieron los mejores resultados en la detección de esta tarea en comparación con el algoritmo de máquinas de vectores de soporte.

Por otro lado la investigación de Edward Raff, *Learning the PE Header, Malware Detection With Minimal Domain Knowledge* [20], con los algoritmos de redes neuronales, en un conjunto de datos con conocimiento del dominio mínimo, expone que dicho algoritmo utilizado como un algoritmo de clasificación, puede mostrar unos resultados bastante prometedores a la hora de detectar archivos malware.

Con dicha literatura, esta investigación trabaja sobre los siguientes algoritmos.

- Redes Neuronales: Son algoritmo de aprendizaje profundo cuyo funcionamiento y estructura es similar a un cerebro biológico. Los datos de entrada de la red neuronal activan los nodos o neuronales de la capa inicial y estas a su vez activan los de la capa siguiente, llegando a procesar los datos de una capa a las capas ocultas de la red y mostrando la solución en las neuronas de la capa de salida.
- Árboles de decisión: Son algoritmos de aprendizaje supervisado no paramétrico, donde el uso de reglas de decisión simples permite clasificar las muestras de entrada a través de sus características. Esta clasificación se realiza a través de una estructura donde la primera regla simple es el nodo raíz y dependiendo de su aplicación con las características, la muestra va saltando a un nodo hijo o a otro, hasta llegar al nodo hoja que muestra la clasificación del dato.
- Bosques Aleatorios: Son algoritmos de aprendizaje supervisado, compuestos por múltiples algoritmos de árboles de decisión para combinar las salidas de estos algoritmos y llegar a una única solución.
- K Nearest Neighbor: Es un algoritmo supervisado no parametrizado que utiliza una serie de muestras de diferentes clases para declararlas vectores iniciales y utilizarlas para determinar el tipo de clase del resto de muestras, a través de la proximidad de estas con los vectores iniciales.
- Naive Bayes: Es un algoritmo de clasificación que utiliza el teorema de Bayes para clasificar las muestras de entrada.

### 4.5.1. Redes neuronales

El algoritmo de redes neuronales pertenece a la categoría de algoritmos de aprendizaje profundo, son estructuras inspiradas en el cerebro humano, imitando la forma en que las neuronas biológicas se envían señales entre sí. Las redes neuronales artificiales se componen

de nodos que simulan el efecto de las neuronas de un cerebro biológico, estas neuronas están interconectadas formando capas neuronales. Al igual que un cerebro biológico envía señales eléctricas al resto de neuronas, los nodos que forman la capa inicial de la red neuronal, envía la información de entrada al resto de capas ocultas a modo de estímulo, activando el resto de neuronas hasta alcanzar la señal de salida[7].

Por extensión, la capa de salida necesita las activaciones de la capa oculta para mostrar un resultado. Las neuronas de una capa determinada se conectan con todas las neuronas de la capa anterior. Las situadas en la capa de entrada, se incian manualmente y harán que se disparen las que se encuentran en la capa oculta, que serán las que detecten patrones. Finalmente, la capa de salida se activa en base a estos patrones.

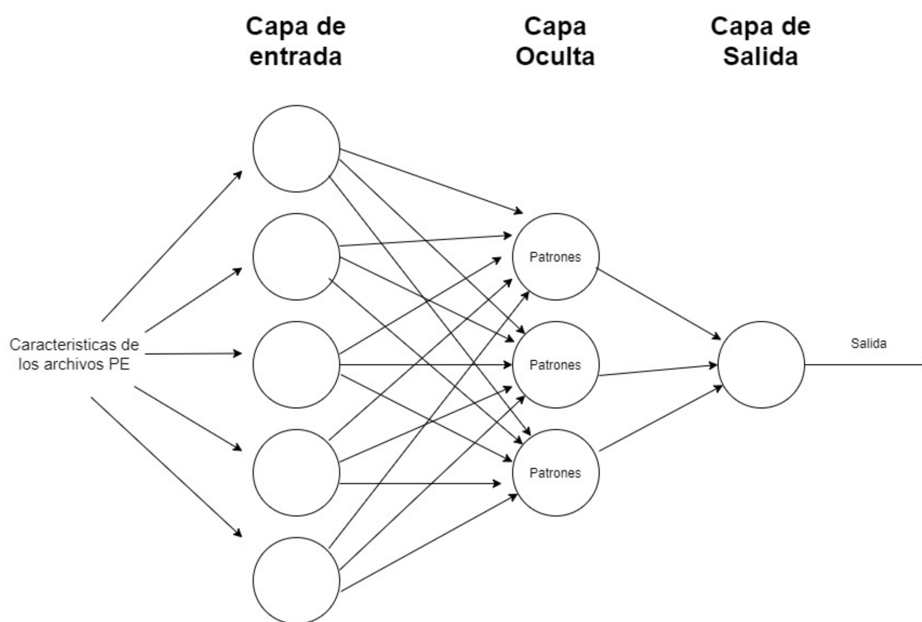



**Figura 4.2**: *Arquitectura de una red neuronal*

La propagación directa es la motivación básica de avance de una red neuronal. La activación de una neurona en una capa dada, L, es la suma de todas las activaciones de la neuronas en la capa anterior.

$$a_n^L = \sum_{k=0} a_k^{L-1}$$

Si sumamos todas las activaciones de las neuronas anteriores, posiblemente la suma sea superior a uno. Sin embargo, se supone que las activaciones de las neuronas están entre cero y uno, por lo que se usa una función especial, llamada Sigmoid, para acercar el valor entre cero y uno. Los valores positivos tienden a uno y los valores negativos tienden a cero[24].

$$\alpha(x) = \frac{1}{1 + e^{-x}}$$

Incorporando la función Sigmoid dentro de la fórmula de activación, obtiene la siguiente ecuación.

$$a_n^L = \alpha(\sum_{k=0} a_k^{L-1})$$

### 4.5.2. Árboles de decisión

Los algoritmos de Árboles de decisión cuentan con una estructura en la que puede tener los siguientes tipos de nodos:

- Nodo de decisión: Consiste en un test relativo al valor de un atributo. De cada nodo interno parten tantas ramas como respuestas haya al test, que normalmente equivale al número de posibles valores que puede tener el atributo en cuestión.
- Nodo hoja: En cada nodo hoja sólo puede haber instancias con un único valor de clase.

El algoritmo busca obtener un árbol lo más simple posible y predictivo. Esto puede resultar un problema debido a que este tipo de algoritmos no garantizan su optimización.

Frente a este problema el algoritmo cuenta con los métodos de clasificación ID3 y C4.5 que son capaces de obtener una buena solución, aunque no garanticen que sea óptima[19].

El número de muestras tiene que ser superior al de posibles clases. Esto se debe a que el proceso de generalización está basado en un análisis estadístico que no podría distinguir patrones de interés a partir de coincidencias aleatorias.

El algoritmo de árboles de decisión utiliza formulas heurísticas para elegir en cada nodo de decisión aquel atributo que tenga mayor capacidad de discriminación sobre las muestras asociadas al nodo[14].

Esta heurística tiene la característica también de intentar encontrar el árbol de decisión más pequeño (con menos nodos). Para ello se utiliza la entropía, una función básica de la teoría de la información que mide el grado de desorden o impureza, de cada una de las particiones generadas.

La entropía realiza una suma ponderada de la información transmitida por cada clase por la proporción de los elementos de la misma. En esta medida se refleja la variabilidad existente, o el grado de desorden, en el conjunto de muestras presentados frente a las diferentes particiones o clases posibles. Donde $n_k$ es el número de muestras de la clase k y n es el número total de muestras.

$$Entropia(E, A_i, v_j) = -\sum_{k=1}^{C} \frac{n_{ijk}}{n_{ij}} log_2(\frac{n_{ijk}}{n_{ij}})$$

Donde $Entropia(E, A_i, v_j)$ es la entropía de los muestras $E$ cuando el atributo $A_i$ tiene el valor $v_j$ , $n_{ijk}$ es el número de muestras que tienen el valor $v_j$ del atributo $A_i$ y pertenecen a la clase $c_k$ , $n_{ij}$ es el número de muestras que tienen el valor $v_j$ del atributo $A_i$, y $C$ es el número de clases.

En todo el proceso recursivo se elige, en cada nodo de decisión, el atributo que mayor ganancia de información aporte. Si se toma como referencia la anterior medida de entropía, se calcula, para cada atributo, la información esperada requerida si se eligiera dicho atributo como nodo de decisión[27].

La ponderación considera la proporción de ejemplos de cada rama.

$$Ganancia(E, A_i) = Entropa(E) - \sum_{v_j \epsilon Valores(A_i)} \frac{n_{ij}}{n} x Entropa(E, A_i, v_j)$$

Dado que se desea elegir el atributo para que la ganancia sea máxima, y, en esta fórmula la Entropía (E) es una constante con respecto al atributo $A_i$ es suficiente con elegir aquel atributo que minimice la fórmula.

$$\sum_{v_j \epsilon Valores(A_i)} \frac{n_{ij}}{n} x Entropa(E, A_i, v_j)$$

### 4.5.3. Bosques Aleatorios

Los Bosques Aleatorios (Random Forest) es un algoritmo que combina la salida de múltiples árboles de decisión para llegar a un único resultado. Existen diferentes árboles de decisión que forman el bosque y difieren unos de otros en dos elementos claves:

- Se utilizan distintas submuestras aleatorias (con reemplazamiento) de entre los datos de entrenamiento.
- Se utilizan distintos grupos de variables predictivas en cada árbol.

Este algoritmo se basa en la evaluación de los distintos árboles de decisión construidos, combinados para la definición de un clasificador fuerte, que representaría la predicción final del modelo. En concreto esta es una técnica que permite el trabajo con árboles de decisión en condiciones de mayor estabilidad. La combinación de árboles distintos corrige la volatilidad de los árboles de decisión simple cuando hay cambios en los datos[21].

Este algoritmo parte de un conjunto de datos de los que se tiene valores, para un conjunto de variables independientes $X = (X_1, X_2 ... X_p)$ y valores para una variable dependiente Y, en este caso de tipo binario que se quiere predecir con una función $f(X)$. En la predición resulta una función de pérdida del tipo $L(Y, f(X))$ donde el error de clasificación será.

$$L(Y, f(X)) = \begin{cases} 0 \ si \ Y = f(X) \\ 1 \ si \ Y \neq f(X) \end{cases}$$

El objetivo del clasificador será minimizar el valor esperado de la función de pérdida minimizando los errores en las predicciones.

$$E_{XY}(L(Y, f(X)))$$

Minimizar la anterior expresión permite ofrecer la siguiente variante condicional.

$$f(x) = E(Y|X = x)$$

Con el objetivo de ofrecer como valor de la predicción final, tomaremos el que más veces se haya predicho para cada uno de los casos del conjunto de datos para todos los árboles.

Este método se llama voto a la clase de la mayoría. Estas probabilidades representan la proporción de veces que un archivo se ha clasificado en una clase. La clase que alcanza el

máximo de esas probabilidades es la clase predicha para el archivo.

Este algoritmo ofrece mejores rendimientos y predicciones que los árboles de decisión, ofreciendo la posibilidad de buenos ajustes con parámetros teóricos.

### 4.5.4. K-Nearest Neighbors

El algoritmo K-Nearest Neighbors es un algoritmo de clasificación de datos que clasifica un objeto desconocido asignándole una clase que represente a la mayoría de sus vecinos más cercanos[30]. Sean los objetos el conjunto de grafos $D$, conjunto de clases $C$ y un conjunto de entrenamiento $TrS = (G_j, c_j)_{j=1}^{M}$ donde $G_j \in D$ es un grafo y $c_j \in C$ es la clase del grafo. El clasificador KNN induce a la $TrS$ a una función de mapeo $f : G \to C$ que asigna una clase a un gráfico desconocido del conjunto de prueba TeS. El problema de K-Nearest Neighbors se puede definir con la siguiente expresión:

$$(G_*, c_*) = arg_{(G_j,c_j) \in TrS} \ \ min \ \ d(G_i, G_j) \forall G_i \in TeS$$

Donde $d(G_i, G_j)$ es la métrica utilizada para calcular una disimilitud entre $G_i$ y $G_j$. En definición, el número de llamadas a la función de disimilitud es igual a $M$, donde $M$ es el tamaño del conjunto de entrenamiento. Para extender la expresión mencionada anteriormente de K-Nearest Neighbors, se introduce $K$ en el conjunto de KNN dentro de la consulta del gráfico $G_i \in TeS$. Sea $K = (G_1, c_1), ..., (G_j, c_j), ..., (G_k, c_k)$ un conjunto de gráficos junto con sus etiquetas de clase $(G_j, c_j) \in TrS$ la expresión anterior quedaría :

$$K = arg_{(G_j,c_j) \in TrS} \ \ sort \ \ d(G_i, G_j) \forall G_i \in TeS$$

Donde sort es una función que realiza una ordenación ascendente de valores $d(G_i, G_j)$, $k$ es el número de valores retenidos para elegir el número de vecinos más cercanos de $G_i$. Para

utilizar la expresión formulada anteriormente en un contexto de clasificación se debe definir un operador de votación.

El operador de votación máxima es una función $\tau : K \rightarrow C$ definida por:

$$c_j^* = arg_{c_j \in C} max \ \ m_{c_j}(K)$$

Donde $m_{c_j}$ es una función que cuenta el número de observaciones que caen en cada clase $c_j$. El clasificador KNN ha sido muy utilizado. El uso de este clasificador es muy sencillo ya que es una técnica no paramétrica y por tanto no necesita conocimientos sobre la distribución de clases. Además, cuando se define la métrica, el clasificador KNN puede proporcionar una explicación del resultado de la clasificación y, por lo tanto, el clasificador KNN tiene una ventaja sobre los otros clasificadores que se consideran modelos de caja negra.

Se ha demostrado que cuando hay suficientes patrones de entrenamiento, el error de clasificación es menor que el error de Bayes.

### 4.5.5. Naïve Bayes

El algoritmo Naïve Bayes es un algoritmo de clasificación basado en la regla de Bayes, que asume los atributos $X_1 ... X_n$ como condicionales independientes unos de otros. El valor de esta suposición es que simplifica dramáticamente la representación de $P(X|Y)$, y el problema de estimación a partir de los datos de entrenamiento[3].

Por ejemplo si $X = X_1, X_2$, podemos simplificarlo derivándolo con la propiedad general de las probabilidades.

$$P(X_1|X_2, Y)P(X_2|Y)$$

En cambio, con los condiciones independientes se puede definir la siguiente expresión.

$$P(X|Y) = P(X_1, X_2|Y)$$

Si derivamos directamente con esta expresión obtenemos la siguiente simplificación.

$$P(X_1|Y)P(X_2|Y)$$

En el caso de que $X$ contenga $n$ atributos, la derivación quedaría así.

$$P(X_1...X_n|Y) = \Pi_{i=1}^{n} P(X_i|Y)$$

Simplificando ahora con el algoritmo Naïve Bayes, asumiendo en general que $Y$ es cualquier variable de valor discreto, y los atributos $X_1...X_n$ son atributos discretos. El objetivo del algoritmo es entrenar un clasificador que genere la distribución de probabilidad sobre los posibles valores de $Y$, para cada nueva instancia $X$ que le pedimos que clasifique. La expresión de la probabilidad de que $Y$ asuma $k$ según la regla de Baye es el siguiente.

$$P(Y = y_k|X_1...X_n) = \frac{P(Y = y_k)P(X_1...X_n|Y = y_k)}{\sum_j P(Y = y_j)P(X_1...X_n|Y = y_j)}$$

Donde la suma se toma sobre todos los valores posibles $y_j$ de $Y$. Ahora, suponiendo que $X_i$ son condiciones independientes se puede utilizar la formula derivada de las condiciones independientes, vistas anteriormente, para expresar la siguiente formula.

$$P(Y = y_k|X_1...X_n) = \frac{P(Y = y_k)\Pi_i P(X_i|Y = y_k)}{\sum_j P(Y = y_j)\Pi_i P(X_i|Y = y_j)}$$

Esta ecuación es la ecuación fundamental para el clasificador Naïve Bayes. Muestra como calcular la probabilidad de que $Y$ tomara cualquier valor dado de una nueva instancia

$X = X_1 ... X_n$ y las distribuciones $P(Y)$ y $P(X_i|Y)$ estimados a partir de los datos de entrenamiento[17].

Interesados por el valor más probable de $Y$ la regla de clasificación de Bayes se formularia como:

$$Y \leftarrow arg\ \ max_{y_k} \frac{P(Y = y_k)\Pi_i P(X_i|Y = y_k)}{\sum_j P(Y = y_j)\Pi_i P(X_i|Y = y_j)}$$

Simplificando la formula:

$$Y \leftarrow arg\ \ max_{y_k} P(Y = y_k)\Pi_i P(X_i|Y = y_k)$$

# Capítulo 5

# Desarrollo del Proyecto

## 5.1. Preparación y tratamiento de datos

Como se ha mencionado en los apartados anteriores, se realizada una búsqueda a través de múltiples repositorios de internet, enfocados al estudio y tratamientos de archivos malware, para recopilar muestras de archivos benignos y malignos. Llegando a contar con un tamaño de muestra, para esta investigación, de 5.012 archivos benignos y 14.599 de archivos malignos.

Se usa la biblioteca de Python *Pefile* para obtener los datos de las cuatros primeras secciones de la cabecera PE; Cabecera DOS, Cabecera PE, Cabecera Opcional y Cabecera de Sección, llegando a crear un datasets de 77 características diferentes pertenecientes a cada dato extraído de cada sección (ver Apéndice A).

Uno de los objetivos de esta investigación es el uso de la función completa, o conocimiento del dominio, para entrenar y probar modelos probabilísticos de algoritmos de machine learning, utilizando la totalidad de las características obtenidas. Para ello se implementa una serie de métodos que permiten seleccionar todas las muestras recopiladas, y obtener los datos de las cabeceras PE. Dicha información es gestionada por un método de virtualización (Figura 5.1) que realiza la conversión de los datos hexadecimal, obtenidos de las cabeceras, en datos numéricos preparados para su procesamiento.

```python
def vectorization_of_data(data1, data2):

    try:
        data = int(data2,16)
        return data

    except ValueError:
        return int (binascii.hexlify(data2.encode('utf-8')),16)
    except TypeError:
        return int(hex(0), 16)
```

**Figura 5.1**: *Método de virtualización*

Una vez creado el datasets se almacenan las características de cada muestra en la base de datos de Postgresql para su posterior uso en los diferentes algoritmos de machine learning.

| | name character varying | e_magic character varying | e_cblp character varying | e_cp character varying | e_crlc character varying | e_cparhdr character varying | e_minalloc character varying | e_maxalloc character varying | e_ss character varying | e_sp character varying | e_csum character varying | e_ip character varying | e_cs character varying | e_lfarlc character varying | e_ovno character va |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | VirusShare_a878ba260... | 23117 | 144 | 3 | 0 | 4 | 0 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 0 |
| 2 | VirusShare_ef9130570... | 23117 | 144 | 3 | 0 | 4 | 0 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 0 |
| 3 | VirusShare_ef84cdeba... | 23117 | 144 | 3 | 0 | 4 | 0 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 0 |
| 4 | VirusShare_6bf3608e6... | 23117 | 144 | 3 | 0 | 4 | 0 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 0 |
| 5 | VirusShare_2cc94d952... | 23117 | 144 | 3 | 0 | 4 | 0 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 0 |
| 6 | VirusShare_eff7676f69... | 23117 | 80 | 2 | 0 | 4 | 15 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 26 |
| 7 | VirusShare_e76cac211... | 23117 | 144 | 3 | 0 | 4 | 0 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 0 |
| 8 | VirusShare_cef6cdf0e... | 23117 | 80 | 2 | 0 | 4 | 15 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 26 |
| 9 | VirusShare_59af5dfb0... | 23117 | 144 | 3 | 0 | 4 | 0 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 0 |
| 10 | VirusShare_fda0add9d... | 23117 | 144 | 3 | 0 | 4 | 0 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 0 |
| 11 | VirusShare_4ea0b18e8... | 23117 | 144 | 3 | 0 | 4 | 0 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 0 |
| 12 | VirusShare_c30d49720... | 23117 | 144 | 3 | 0 | 4 | 0 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 0 |
| 13 | VirusShare_efcd8cbc4... | 23117 | 144 | 3 | 0 | 4 | 0 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 0 |
| 14 | VirusShare_9077ed6af... | 23117 | 144 | 3 | 0 | 4 | 0 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 0 |
| 15 | VirusShare_efd86c905... | 23117 | 144 | 3 | 0 | 4 | 0 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 0 |
| 16 | VirusShare_ac5a6f8fc... | 23117 | 144 | 3 | 0 | 4 | 0 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 0 |
| 17 | VirusShare_efc9e836f... | 23117 | 144 | 3 | 0 | 4 | 0 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 0 |
| 18 | VirusShare_efc2528eb... | 23117 | 144 | 3 | 0 | 4 | 0 | 17744 | 0 | 332 | 1 | 29305 | 15462 | 19547 | 29295 |
| 19 | VirusShare_efa1b8660... | 23117 | 144 | 3 | 0 | 4 | 0 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 0 |
| 20 | VirusShare_f4cd9cc0e... | 23117 | 144 | 3 | 0 | 4 | 0 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 0 |
| 21 | VirusShare_eface11cd... | 23117 | 144 | 3 | 0 | 4 | 0 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 0 |
| 22 | VirusShare_efe4e3d61... | 23117 | 144 | 3 | 0 | 4 | 0 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 0 |
| 23 | VirusShare_efa2b659c... | 23117 | 144 | 3 | 0 | 4 | 0 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 0 |
| 24 | VirusShare_c297eee5e... | 23117 | 144 | 3 | 0 | 4 | 0 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 0 |
| 25 | VirusShare_2a7a8bfe5... | 23117 | 144 | 3 | 0 | 4 | 0 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 0 |
| 26 | VirusShare_ef79423bf... | 23117 | 144 | 3 | 0 | 4 | 0 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 0 |
| 27 | VirusShare_e763e9eeb... | 23117 | 144 | 3 | 0 | 4 | 0 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 0 |
| 28 | VirusShare_1b81fd14b... | 23117 | 80 | 2 | 0 | 4 | 15 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 26 |
| 29 | VirusShare_efbf43825... | 23117 | 144 | 3 | 0 | 4 | 0 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 0 |
| 30 | VirusShare_ef6cb2a3a... | 23117 | 144 | 3 | 0 | 4 | 0 | 65535 | 0 | 184 | 0 | 0 | 0 | 64 | 0 |

**Figura 5.2**: *Datasets almacenado en la BBDD*

Se genera múltiples métodos genéricos con el objetivo de recoger los datos almacenados en la base de datos de postgresql y generar los conjuntos de entrenamiento y de tests, que se utilizan posteriormente para la preparación y evaluación de los modelos creados por los

algoritmos de machine learning.

Un ejemplo de estos métodos es el siguiente, donde se muestra la separación del datasets, creado previamente, en dos subconjuntos diferentes, el primer subconjunto corresponde con los datos de entrenamiento que contiene una salida conocida y el modelo utiliza para aprender, y el segundo subconjunto corresponde con los datos de prueba que no contendrán dicha salida y se utilizan para evaluar la calidad del algoritmo. Para obtener modelos con mejores métricas se suele dar un mayor porcentaje al conjunto de entrenamiento que al conjunto de test, por lo general los porcentajes de división con los que se suelen trabajar son del 80/20 o 70/30, en esta investigación se utiliza los porcentajes de 80/20 a la hora de evaluar la calidad del modelo generado.

```
# Preparación de los datos

    def data_Preparation(datasets):

        X = datasets.iloc[:, 0:77].values
        y = datasets.iloc[:, 77].values

        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,random_state=0)
        scaler = MinMaxScaler()
        X_train = scaler.fit_transform(X_train)
        X_test = scaler.transform(X_test)

        return X_train, y_train, X_test, y_test
```

**Figura 5.3**: *Método para generar el conjunto de entrenamiento y test*

Se genera dos métodos diferentes de validación cruzada, una validación cruzada a la hora de crear el modelo probabilísticos y otra para evaluar al mismo.

El primer método utiliza la librería *KFold* para dividir el dataset en pliegues y generar un modelo que utilice los subconjuntos de entrenamiento y prueba de cada pliegue.

El segundo método se usa para validar el modelo obtenido, para ello se utiliza el por-

```python
def create_decision_trees(datasets):

    cv = KFold(n_splits=10)
    accuracies = list()
    max_attributes = len(list(datasets))
    depth_range = range(1, max_attributes + 1)


    for depth in depth_range:
        fold_accuracy = []
        tree_model = RandomForestClassifier(n_estimators = 50,
                                            criterion='entropy',
                                            min_samples_split=2,
                                            min_samples_leaf=1,
                                            max_depth = depth,
                                            class_weight={0:2.912809257781325},
                                            random_state = 0)
        for train_fold, valid_fold in cv.split(datasets):
            f_train = datasets.loc[train_fold]
            f_valid = datasets.loc[valid_fold]

            model = tree_model.fit(X = f_train.drop(['type'], axis=1),
                                   y = f_train["type"])
            valid_acc = model.score(X = f_valid.drop(['type'], axis=1),
                                    y = f_valid["type"])
            fold_accuracy.append(valid_acc)

        avg = sum(fold_accuracy)/len(fold_accuracy)
        accuracies.append(avg)


    return model
```

**Figura 5.4**: *Validación cruzada para generar el modelo*

centaje de división 80/20 y la librería *cross_val_predict* que permite generar estimaciones a través de validación cruzada. La librería *cross_val_predict* permite dividir el conjunto de pruebas en pliegues para poder evaluar el modelo introducido.

```python
def model_Accuracy(model,x, y):

    pred = cross_val_predict(model, x, y, cv=10)
    prob = cross_val_predict(model, x, y, cv=10, method='predict_proba')

    print('Pred : ', pred)
    print('Prob : ', prob)

    return pred, prob
```

**Figura 5.5**: *Validación cruzada para evaluar el modelo*

Para ambos métodos de validación se utiliza la contantes $K = 10$ (*cv* en el uso de cross_val_predict) en el número de pliegues, debido a que proporciona una variación en el conjunto de datos que permite el aprendizaje del modelo y al mismo tiempo obtener el conjunto de validación.

## 5.2. Red Neuronal

Como se ha comentado en el Capítulo 4, los algoritmos de redes neuronales son algoritmos con una estructura que asemejar el funcionamiento sináptico de un cerebro biológico.

Se ha realizado una descarga del dataset almacenado en la base de datos en formato .csv para ser utilizo por el algoritmo. La división de este se realiza separando el 80 % de las instancias que lo componen, para formar el conjunto de entrenamiento, y el 20 % restante compondrá el conjunto de testing de la propia red.

Debido a que el datasets cuenta con un desequilibrio de clases, con más instancias malware (14.599) que instancias benignas (5.012). Se procede a utilizar el método de refuerzo de aprendizaje, ya que se cuenta con un tamaño de muestra limitado para aplicar las técnicas de balanceo de datos, para dar más peso a las instancias de la clase inferior del dataset. Se aplica un peso para la clase benigna de *2.912809257781325*, la media por porcentaje del

total de las instancias utilizadas, frente a un peso de *1.0* para la clase malware.

```
y = dataset['type']
x = dataset.drop(['type'], axis=1)

print(x.shape)
print(y.shape)

# convert to numpy arrays
x = np.array(x)

# Weights
class_weights={0: 2.912809257781325, 1:1.0}
```

**Figura 5.6**: *Método de refuerzo de aprendizaje*

Se crea un modelo secuencial que permite la utilización de las predicciones obtenidas en una capa de entrada, u oculta, como si fueran variables explicitas o, de entrada, para las posteriores capas. De esta forma, el intervalo correspondientes a la primera capa de la red actúa como un intervalo informativo y el correspondiente a la segunda capa actúa como intervalo informado.

La estructura de la red neuronal implementada en esta investigación está compuesta por tres capas; una capa de entrada, una capa oculta y una capa de salida.

- La capa de entrada o capa inicial de la red cuenta con 77 entradas de datos, correspondiente a las 77 características que contiene cada instancia del datasets, con una función de activación de tipo *relu* que permite la rectificación de la unidad lineal en valores predeterminados.

- La capa oculta o intermedia esta compuesta por 39 neuronas, la media entre el número de características de la capa de entrada y de la capa final, al igual que la capa de entrada cuenta con una función de activación de tipo *relu*.

- La capa de salida cuenta con una sola neurona que permite mostrar el resultado del modelo. La función de activación de esta neurona es de tipo "Sigmoid"destinado a resultados binarios.

Se implementa el uso de la validación cruzada anidada, con 10 pliegues dividiendo el conjunto de datos en tres conjuntos diferentes; conjunto de datos de entrenamiento, conjunto de datos de testeo y conjunto de datos de validación, estos últimos se utilizan al final de la implementación del modelo para validar la eficiencia del mismo al utilizarse datos no antes utilizados para su entrenamiento y para su testeo.

```
# Definir la validación cruzada de K-fold
kfold = KFold(n_splits=10, shuffle=True)
class_weights={0: 2.912809257781325, 1: 1.0}

# Evaluación del modelo K-fold Cross Validation
fold_no = 1
for train, test in kfold.split(datasets):

    f_train = datasets.loc[train]
    f_valid = datasets.loc[test]
    x = f_train.drop(['type'], axis=1)
    model = Sequential()
    model.add(Dense(77, input_shape=(x.shape[1],), activation='relu'))
    model.add(Dense(39, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    model.summary()

    # Compilación del modelo

    model.compile(optimizer='Adam',
                    loss='binary_crossentropy',
                    metrics=['accuracy'])



    print('------------------------------------------------------------------------')
    print(f'Training for fold {fold_no} ...')

    # Entrenamiento del modelo
    es = EarlyStopping(monitor='val_accuracy',
                        mode='max',
                        patience=10,
                        restore_best_weights=True)

    z = np.array(x)

    history = model.fit(z,
                        y = f_train["type"],
                        callbacks=[es],
                        epochs=100,
                        batch_size=30,
                        class_weight=class_weights,
                        validation_split=0.2,
                        shuffle=True,
                        verbose=1)

    # Generar métricas de generalización
    scores = model.evaluate(x = f_valid.drop(['type'], axis=1),  y = f_valid["type"], verbose=0)
```

**Figura 5.7**: *Arquitectura de la red neuronal*

La red neuronal que se implementa esta orientando a la clasificación de determinadas clases de instanticas, para optimizar al maximo el reultado del modelo se utiliza los siguientes algoritmos y funciones.

- Algoritmos de optimización *Adam*: calcula la tasa de aprendizaje adaptativo individual, para diferentes parámetros, a partir de las estimaciones del primer y segundo momento de los gradientes. Los resultados empíricos se obtienen de manera rápida y son bastante eficaces.
- Función de pérdida *Entropía cruzada*: La entropía cruzada es una medida del campo de la teoría de la información, que se basa en la entropía, y que calcula la diferencia entre dos distribuciones de probabilidad para una variante aleatoria dada.
- Métricas de compilación, tipo *accuracy*: permite calcular la frecuencia con la que las predicciones son iguales a las etiquetas.

Para que el modelo no se quede sobreajustado se establece un callback que permita detener el entrenamiento cuando se detecte perdidas en la validación de los últimos 30 lotes.

```
# Compilación del modelo

model.compile(optimizer='Adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])



print('------------------------------------------------------------------------')
print(f'Training for fold {fold_no} ...')

# Entrenamiento del modelo
es = EarlyStopping(monitor='val_accuracy',
                   mode='max',
                   patience=10,
                   restore_best_weights=True)

z = np.array(x)

history = model.fit(z,
                    y = f_train["type"],
                    callbacks=[es],
                    epochs=100,
                    batch_size=30,
                    class_weight=class_weights,
                    validation_split=0.2,
                    shuffle=True,
                    verbose=1)
```

**Figura 5.8**: *Parametros de entrenamiento*

### 5.2.1. Matriz de Confusión

Para obtener las métricas del modelo de red neuronal generado; exactitud, precisión, sensibilidad y especificación, se utiliza la métrica matriz de confusión, ver capítulo 3 tipos de métricas. La matriz confusión muestra el número de verdaderos positivos, verdaderos negativos, falsos positivos y falsos negativos de un modelo, permitiendo obtener el resto de las métricas necesarias para determinar la efectividad del modelo evaluado.

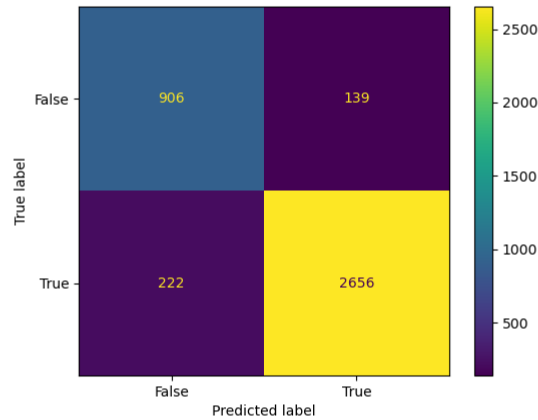



**Figura 5.9**: *Matriz de confusión Red Neuronal*

Con los datos obtenidos de la matriz de confusión se obtiene la exactitud del modelo generado de red neuronal. Este modelo cuenta con un porcentaje de exactitud elevado con 90,8 % de exactitud. Esta métrica puede resultar poco eficiente sin los porcentajes de precisión y sensibilidad que lo apoyen.

$$(906 + 2656)/(906 + 2656 + 222 + 139) = 3562/3923 = 0,9079 \simeq 90,80\,\%$$

Realizando la fórmula de precisión con los datos obtenidos de la matriz de confusión, se obtiene un porcentaje de precisión del 95,02 % de que el valor probado sea el correcto.

$$(2656)/(2656 + 139) = 2656/2795 = 0,95026 \simeq 95,02\,\%$$

Aplicando la métrica de sensibilidad para detectar la habilidad del modelo para identificar los casos relevantes, se obtiene un porcentaje del 92,28 % de probabilidades de que el modelo capture la mayor parte de los casos positivos.

$$(2656)/(222 + 2656) = 2656/2878 = 0,9228 \simeq 92,28\,\%$$

La métrica de especifidad muestra la capacidad de discriminación del modelo antes los casos negativos, Aplicando la fórmula de la especifidad da como resultado que el modelo de red neuronal cuenta con un porcentaje del 86,70 % de dificultad para obtener falsos positivos en una prueba de testeo.

$$(906)/(906 + 139) = 906 + 1045 = 0,8669 \simeq 86,70\,\%$$

### 5.2.2. Informe de clasificación

Se realiza un informe de clasificación con el modelo generado de red neuronal, donde se muestra las métricas de precisión, sensibilidad y media armónica por cada clase del datasets. La clase perteneciente a las muestras benignas, clase 0, cuenta con una precisión del 80 % para detectar las muestras de esta clase, y una sensibilidad del 87 % para capturar todos los casos positivos. En cambio, la clase de las muestras malwares, clase 1, cuenta con una precisión del 95 % para detectar las muestras de esta clase y con un nivel de probabilidad del 92 % de capturar todos los casos positivos de la misma.

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.80 | 0.87 | 0.83 | 1045 |
| 1 | 0.95 | 0.92 | 0.94 | 2878 |
| accuracy | | | 0.91 | 3923 |
| macro avg | 0.88 | 0.89 | 0.89 | 3923 |
| weighted avg | 0.91 | 0.91 | 0.91 | 3923 |

**Figura 5.10**: *Informe de clasificación Red Neuronal*

## 5.3. Arboles de decisión

Los algoritmos de árboles de decisión son algoritmos con una estructura dividida en nodos interconectados, formando diferentes niveles de profundidad. Todo algoritmo de árbol de decisión comienza con un nodo raíz, que asemeja a la primera condición que evalúa el dato de entrada, desde ahí la muestra baja los diferentes niveles del algoritmo hasta alcanzar el nodo salida quien muestra el resultado final del modelo.

Al igual que con el resto de los algoritmos de esta investigación, se procesa el dataset almacenado en la base de datos y se divide en diversos conjuntos de entrenamiento y test, el porcentaje de este grupo es del 80 % para el conjunto de entrenamiento y el 20 % restante pertenecientes al grupo de test.

Al utilizarse la técnica de retropropagación para optimizar el modelo creado (se utiliza el conjunto de test para testear el modelo y ajustar los parámetros a razón de los resultados) y evitar precisiones incorrectas en el modelo final, se realiza una tercera división del dataset correspondiente al conjunto de validación. Este conjunto de validación será el conjunto de datos que se utiliza al final del testeo y entrenamiento del modelo para certificar la precisión real.

El uso del conjunto de validación dentro de la creación del modelo será a través de la validación cruzada anidada. La validación cruzada divide los datos en los conjuntos de entrenamiento y prueba, luego valida el árbol usando los pliegues del conjunto de entrenamiento, esto provoca que los resultados estén condicionados a la división principal de entrenamiento/test, conllevando que, si el dataset es pequeño, el rendimiento del árbol final tenga una estimación imprecisa de su rendimiento real. Por esta razón en este algoritmo se utiliza la validación cruzada anidada que permite dividir los datos en los $L$ tiempos de entrenamiento y prueba, utilizando el conjunto de validación en cada una de las $L$ divisiones, evitando así el inconveniente de la validación cruzada simple.

```
for depth in depth_range:
    fold_accuracy = []
    tree_model = tree.DecisionTreeClassifier(criterion='entropy',
                                             min_samples_split=2,
                                             min_samples_leaf=1,
                                             max_depth = depth,
                                             class_weight={0:2.912809257781325})
    for train_fold, valid_fold in cv.split(datasets):
        f_train = datasets.loc[train_fold]
        f_valid = datasets.loc[valid_fold]

        model = tree_model.fit(X = f_train.drop(['type'], axis=1),
                               y = f_train["type"])
        valid_acc = model.score(X = f_valid.drop(['type'], axis=1),
                                y = f_valid["type"]) # calcula la precision con el segmento de validacion
        fold_accuracy.append(valid_acc)
```

**Figura 5.11**: *Fragmento de código, Validación Cruzada*

Se da más peso a las instancias de clase benigna, debido a que existe un desequilibrio de clase respecto al número de instancias de la clase contraria. Para este algoritmo se procede a dar un peso de 2.912809257781325 sobre la clase benigna.

Los algoritmos de árboles de decisión tienen dos tipos de parametros configurables, que pueden mejorar la precisión del mismo.

- Parámetros que se ajustan durante el entrenamiento del modelo de manera automatica por el propio algoritmo.
- Parámetros que se ajustan antes del entrenamiento, tambien conocidos como hiperparameros, que se configuran de manera manual.

En este investigación nos centramos en la configuración de los hiperparametros para obtener un modelo lo más preciso posible. Para ello nos enfocamos en los siguientes hiperparametros:

- Profundidad máxima del árbol.

- Cantidad minima de muestras que debe tener un nodo para poder subdividir.
- Función para medir la calidad de un división

Se determina que la profundidad máxima del árbol es igual a la longitud total del datasets. Con lo que minimizamos el sobreajuste de este limitándolo a la longitud de los datos utilizados para su entrenamiento, testeo y validación.

Los hiperparametros de cantidad mínima para que un nodo se pueda subdividir, es el valor mínimo por defecto que se puede atribuir a este hiperparametro, 2 muestras para que cada nodo se pueda subdividir.

Existen dos funciones para medir la calidad de una división en algoritmos de árboles de decisión:

- Ganancia de información
- Índice Gini

La ganancia de información calcula la diferencia entre la entropía antes de la división y la entropía promedio después de la división del conjunto de datos, en función de los valores de atributos dados. Hay que tener en cuenta que la entropía es la medida de incertidumbre de un conjunto de datos dado y su valor describe el grado de aleatoriedad de un nodo en particular. Esto ocurre cuando al margen de la diferencia de un resultado es muy bajo y por lo tanto el modelo no tiene confianza en la precisión de la predicción. Cuanto mayor sea la entropía, mayor será la aleatoriedad en el conjunto de datos.

En cambio, el índice Gini considera una división binaria para cada atributo, calculando una suma ponderada de la impureza de cada participación.

Se elige para este modelo el criterio de ganancia de información, *entropy*, debido a que este criterio disminuye la aleatoriedad del conjunto de datos, entropía.

Para evitar que el modelo se sobreajuste por la cantidad de divisiones que hay, lo que resulta obtener predicciones deficientes, se procede a eliminar las ramas que tienen poca o ningún importancia en el proceso de toma de decisión, lo que se conoce como poda, existe un parámetro configurable a la hora de crear el modelo de árbol de decisión que es *min_ samples_ leaf* que permite realizar una poda previa mientras crece el árbol. Este parámetro se configura para que elimine los nodos finales que contengan menos de una muestra.

```
tree_model = tree.DecisionTreeClassifier(criterion='entropy',
                                         min_samples_split=2,
                                         min_samples_leaf=1,
                                         max_depth = depth,
                                         class_weight={0:2.912809257781325})
```

**Figura 5.12**: *Creación de árbol de decisión con los hiperparametros*

### 5.3.1. Matriz de Confusión

Para evaluar la exactitud, la precisión, sensibilidad y especificación del modelo generado, se utiliza la métrica de matriz de confusión. La matriz obtenida del modelo de árbol de decisión es la siguiente.

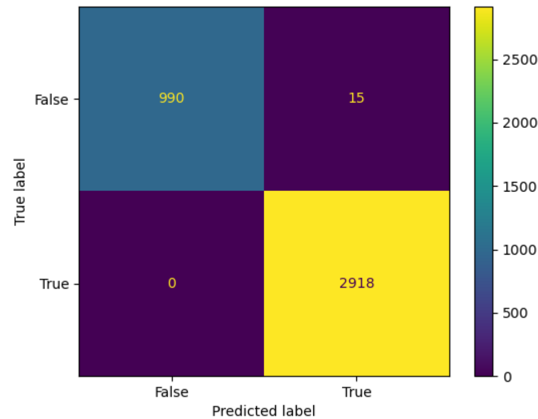



**Figura 5.13**: *Matriz de confusión Árboles de decisión*

Realizando la fórmula de exactitud (accuracy) para obtener el porcentaje de precisiones correctas frente al total, se obtiene un modelo bastante exacto con un porcentaje del 99,61 %. Esta métrica puede ser poco útil en casos en el que el conjunto de datos sufre de algún desequilibrio de clases, para certificar la eficiencia del modelo utilizamos las métricas de precisión y sensibilización que apoyen los resultados de la exactitud.

$$(990 + 2918)/(990 + 2918 + 15 + 0) = 3908/3923 = 0,99617 \simeq 99,61\,\%$$

Aplicando la fórmula de precisión sobre los datos obtenidos de la matriz de confusión, obtenemos un modelo con una precisión del 99,49 % de que el valor probado sea el correcto.

$$(2918)/(2918 + 15) = 2918/2933 = 0,99488 \simeq 99,49\,\%$$

Se implementa la métrica de sensibilidad para detectar la habilidad del modelo para identificar los casos relevantes, obteniendo un porcentaje del100 % donde se puede ver que el modelo implementado es bastante sensible, capturando la totalidad de los casos positivos.

$$(2918)/(0 + 2918) = 2918/2918 = 1 \simeq 100\,\%$$

La ultima métrica a evaluar es la métrica de especifidad, donde se muestra la capacidad de discriminación de casos negativos. Aplicando la formula vemos que el modelo cuenta con un porcentaje del 98,51 % de dificultad de obtener falsos positivo en una prueba de testeo.

$$(990)/(990 + 15) = 990/1005 = 0,98507 \simeq 98,51\,\%$$

Todos los resultados obtenidos de la matriz de confusión muestran un modelo bastante optimizado y eficiente con unos porcentajes bastante buenos.

### 5.3.2. Informe de clasificación

El informe de clasificación muestras las métricas obtenidas de la matriz de confusión a nivel de clases. La clase 0 perteneciente a las muestras de tipo benigno, cuenta con una precisión 100 % de detectar las muestras de esta clase, y una sensibilidad del 98,50 % para capturar todos los casos positivos.

En contra partida, nos encontramos con una probabilidad del 98,49 % de detectar las muestras de clase 1 o clase maligna, con un nivel de sensibilidad del 100 % para capturar todos los casos positivos de esta clase.

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 1.00 | 0.99 | 0.99 | 1005 |
| 1 | 0.99 | 1.00 | 1.00 | 2918 |
| accuracy | | | 1.00 | 3923 |
| macro avg | 1.00 | 0.99 | 0.99 | 3923 |
| weighted avg | 1.00 | 1.00 | 1.00 | 3923 |

**Figura 5.14**: *Informe de clasificación Árbol de decisión*

## 5.4. Bosque Aleatorio

Los bosques aleatorios tienen una arquitectura compuesta de múltiples árboles de decisión, con el objetivo de encontrar la mejor división para separar los datos en subconjuntos. Este algoritmo predice resultados más precisos que los propios algoritmos de árboles, que son más propensos al sobreajuste del modelo.

Para implementar el algoritmo de bosques aleatorios, se han utilizado dos métodos diferentes de aprendizaje por conjuntos, con el objetivo de mejorar los resultados predictivos del modelo.

- La técnica Bagging.
- La técnica Boosting.

La técnica Bagging es una técnica de conjunto elaborada en 1996 por Breiman para mejorar la precisión de la clasificación y generalizar patrones de datos. Bagging utiliza un grupo de clasificadores basados en árboles $h(x, \Theta_k), k = 1, ...$ donde $x$ es el vector de entrada y $\Theta_k$ son los vectores aleatorios independientes e idénticamente distribuidos.

Esta técnica genera un grupo de árboles de clasificación utilizando una técnica de arranque mediante el muestreo aleatorio de ciertas proporciones de los datos de calibración con reemplazo. Posteriormente, las asignaciones de clases múltiples para cada muestra generado por los árboles de clasificación se asignan a una clase para la que recibe un número máximo de votos del grupo de clasificadores.

La técnica Boosting es una técnica de aprendizaje de conjunto desarrollado por Freund y Schapire en 1996 con el objetivo de mejorar la precisión de clasificación al convertir el grupo de clasificadores débiles en una clasificador fuerte.

En contraste entre las dos técnicas, es que utiliza todo el conjunto de datos de calibración para la clasificación en lugar de volver a muestrear una determinada proporción de los datos de calibración. A las muestras de calibración se les asigna inicialmente pesos iguales para iteración de la clasificación, y posteriormente, los pesos se reajustan en función del ajuste a los datos de calibración en iteraciones posteriores del árbol de clasificación. Las muestras de calibración mal clasificadas en la iteración anterior reciben más peso que las muestras de calibración clasificadas correctamente. De esta forma, el algoritmo se centra en las muestras de calibración que se han clasificado correctamente en la iteración de prioridades, y los errores de clasificación que se produjeron en la iteración anterior se corrigen en la iteración posterior.

Al igual que con el resto de los algoritmos presentados en esta investigación el dataset se ha dividido en dos conjuntos de datos diferentes; el 80 % del dataset lo forma el conjunto de entrenamiento y el 20 % restante el de testeo.

Al aplicar técnicas de retropropagación, para optimizar el modelo generado y evitar porcentajes de precisión incorrectos, se realiza una tercera división del dataset correspondiente al conjunto de validación. Este conjunto de validación se utiliza para validar la calidad del modelo que se ha creado una vez finalizada la fase de testeo y ajuste.

Al igual que con el algoritmo de árboles de decisión, esta investigación se centra en la configuración de los hiperparametros del modelo para ajustar el mismo y mejorar su precisión. Para implementar los métodos de aprendizaje de conjuntos Bagging y Boosting, se utiliza los métodos de aprendizaje bosques aleatorios y aumento de gradiente(Gradient Boosting Machine) respectivamente, utilizando las clases *RandomForestClassifier* y *GradientBoostingClassifier* de la biblioteca sklearn.

Para la clase RandomForestClassifier se ha dado un peso de 2.912809257781325 a la instancia de la clase benigna, debido a que existe un desequilibrio de clase respecto al número de instancias de la clase contraria. Además de esto se elige el criterio de ganancia de información *entropy* para disminuir la aleatoriedad del conjunto de datos.

```
def create_decision_trees(datasets):

    cv = KFold(n_splits=10)
    accuracies = list()
    max_attributes = len(list(datasets))
    depth_range = range(1, max_attributes + 1)


    for depth in depth_range:
        fold_accuracy = []
        tree_model = RandomForestClassifier(n_estimators = 50,
                                            criterion='entropy',
                                            min_samples_split=2,
                                            min_samples_leaf=1,
                                            max_depth = depth,
                                            class_weight={0:2.912809257781325},
                                            random_state = 0)
        for train_fold, valid_fold in cv.split(datasets):
            f_train = datasets.loc[train_fold]
            f_valid = datasets.loc[valid_fold]

            model = tree_model.fit(X = f_train.drop(['type'], axis=1),
                                   y = f_train["type"])
            valid_acc = model.score(X = f_valid.drop(['type'], axis=1),
                                    y = f_valid["type"])
            fold_accuracy.append(valid_acc)

        avg = sum(fold_accuracy)/len(fold_accuracy)
        accuracies.append(avg)


    return model
```

**Figura 5.15**: *Fragmento de Código creación del Árbol de decisión*

Es frecuente el uso del método de aumento de gradiente con boosting en aplicaciones de detección de anomalías en entornos de aprendizaje supervisado, donde las clases suelen estar

desequilibradas[28]. Este método es más optimo en comparación con los bosques aleatorios porque realiza la optimización en el espacio de funciones en lugar del espacio de parámetros, facilitando el uso de funciones de pérdida personalizadas.

Los hiperparametros configurados en este método son la tasa de aprendizaje(learning_rate), involucrada en la reducción de la contribución que hace cada árbol en el conjunto del bosque aleatorio, este valor es el establecido por defecto, 0,1. El número de estimación (n_estimators) es el número de etapas de refuerzo a realizar, un gran número de etapas da como resultado un mejor rendimiento del modelo generado, se establece para este parámetro el valor por defecto 100.

```python
def create_decision_trees(datasets):

    cv = KFold(n_splits=10)
    accuracies = list()
    max_attributes = len(list(datasets))
    depth_range = range(1, max_attributes + 1)

    for depth in depth_range:
        fold_accuracy = []
        tree_model = GradientBoostingClassifier(n_estimators = 100, learning_rate=0.1, max_features=2, max_depth=2, random_state=0)
        for train_fold, valid_fold in cv.split(datasets):
            f_train = datasets.loc[train_fold]
            f_valid = datasets.loc[valid_fold]

            model = tree_model.fit(X = f_train.drop(['type'], axis=1),
                                   y = f_train["type"])
            valid_acc = model.score(X = f_valid.drop(['type'], axis=1),
                                    y = f_valid["type"])
            fold_accuracy.append(valid_acc)

        avg = sum(fold_accuracy)/len(fold_accuracy)
        accuracies.append(avg)

    return model
```

**Figura 5.16**: *Fragmento de Código creación Gradient Boosting Machine*

### 5.4.1. Matriz de Confusión

Se utiliza la matriz de confusión para obtener las métricas de evaluación de los modelos implementados. La matriz de evaluación del modelo bosque aleatorio es la siguiente.

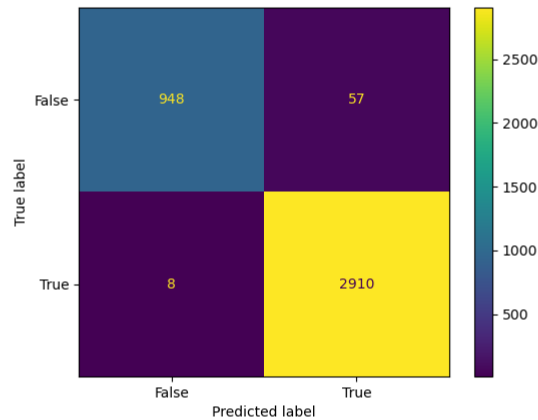



**Figura 5.17**: *Matriz de confusión Bosque Aleatorio*

Aplicando la fórmula de la exactitud, se muestra un modelo con un porcentaje para detectar muestras correctas del 98,34 %. Este porcentaje es bastante elevado y se requiere aplicar las fórmulas de las métricas de precisión y sensibilidad para apoyar dicho resultado.

$$(948 + 2910)/(948 + 2910 + 57 + 8) = 3858/3923 = 0,98343 \simeq 98,34\,\%$$

La formula de precisión muestra un modelo con un porcentaje del 98,08 % de que el valor clasificado por el algoritmo sea el correcto.

$$(2910)/(2910 + 57) = 2910/2967 = 0,98078 \simeq 98,078\,\%$$

Implementando la métrica de sensibilidad para detectar la habilidad del modelo para identificar los casos relevantes, se obtiene un porcentaje del 99,72 % de sensibilidad, lo que indica un modelo que captura la mayor parte de los casos positivos.

$$(2910)/(8 + 2910) = 2910/2918 = 0,99725 \simeq 99,72\,\%$$

La métrica de especifidad muestra un porcentaje del 94,33 % de dificultad del modelo para obtener falsos positivos en una prueba de testeo.

$$(948)/(948 + 57) = 948/1005 = 0,94328 \simeq 94,33\,\%$$

Evaluando de la misma forma que el algoritmo de bosque aleatorios, la matriz de confusión obtenida del modelo de aumento de gradiente es la siguiente.

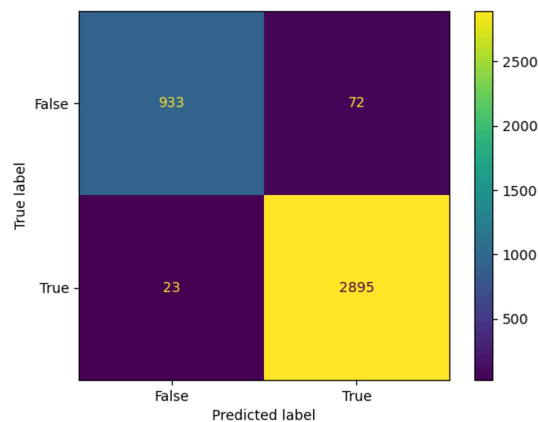



**Figura 5.18**: *Matriz de confusión Aumento de gradiente*

Desarrollando la formula de exactitud se obtiene un porcentaje del 97,58 % de probabilidades de que el modelo de aumento de gradiente pueda detectar los casos positivos. Al igual

que con el modelo de bosque aleatorio, se procede a realizar las formulas de las métricas de precisión y sensibilidad.

$$(933 + 2895)/(933 + 2895 + 72 + 23) = 3828/3923 = 0,97578 \simeq 97,58\,\%$$

La métrica de precisión muestra que el modelo de aumento de gradiente tiene un porcentaje de precisión del 97,57 % de que el valor probado sea el correcto.

$$(2895)/(2895 + 72) = 2895/2967 = 0,97573 \simeq 97,57\,\%$$

Aplicando la fórmula de sensibilidad se obtiene que el modelo es capaz de captura el 99,21 % de los casos positivos que pasan por el algoritmo.

$$(2895)/(23 + 2895) = 2895/2918 = 0,99211 \simeq 99,21\,\%$$

La métrica de especifidad muestra que el modelo tiene una probabilidad del 92,94 % de dificultad de obtener falsos positivos en pruebas de testeo.

$$(948)/(948 + 72) = 948/1020 = 0,92941 \simeq 92,94\,\%$$

### 5.4.2. Informe de clasificación

El informe de clasificación muestra las métricas obtenidas de la matriz de confusión a nivel de cada clase que compone el dataset. Como se muestra en la siguiente imagen perteneciente al informe de clasificación de bosque aleatorio, se puede ver que para la clase benigna, o clase 0, se obtiene un porcentaje de precisión a la hora de detecta los casos positivos del 99 %, y una sensibilidad para capturar dichos casos positivos del 94 % lo que indica que cierto porcentaje de los casos de muestras benignas son verdaderos negativos pertenecientes a la clase malware. En cambio para la clase maligna, o clase 1, el porcentaje de precisión para detectar casos positivos es del 98 % y la sensibilidad del modelo es del 100 %, lo que indica que los casos detectados de tipo maligno se puede asegurar que son de esta misma clase.

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.99 | 0.94 | 0.97 | 1005 |
| 1 | 0.98 | 1.00 | 0.99 | 2918 |
| accuracy | | | 0.98 | 3923 |
| macro avg | 0.99 | 0.97 | 0.98 | 3923 |
| weighted avg | 0.98 | 0.98 | 0.98 | 3923 |

**Figura 5.19**: *Informe de clasificación Bosque Aleatorio*

En cambio, el informe mostrado para el modelo de aumento de gradiente, muestra un porcentaje de precisión para la clase benigna, o clase 0, del 98 % para detectar los casos positivos y una sensibilidad del 93 % para capturar dichos casos, lo que indica que cierto porcentaje de las muestras detectadas como benignas son en realidad verdaderos negativos. Para la clase maligna, o clase 1, se obtiene un porcentaje de precisión del 98 % para detectar los casos positivos y una sensibilidad del 99 % para captura dichos casos, lo que también indica un porcentaje inferior de casos detectados como falsos positivos.

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.98 | 0.93 | 0.95 | 1005 |
| 1 | 0.98 | 0.99 | 0.98 | 2918 |
| accuracy | | | 0.98 | 3923 |
| macro avg | 0.98 | 0.96 | 0.97 | 3923 |
| weighted avg | 0.98 | 0.98 | 0.98 | 3923 |

**Figura 5.20**: *Informe de clasificación Aumento de gradiente*

## 5.5. K-Nearest Neighbor

El algoritmo de K-Nearest Neighbors es un algoritmo de aprendizaje supervisado no parametrizado, que utiliza la proximidad de los vectores(vecinos) para determinar el tipo de clase del vector de consulta (valor de entrada).

Se asigna, al vector de consulta, la clase que se representa con mayor frecuencia alrededor de un punto de datos determinados, esto se conoce en la literatura como *voto de la mayoría.*

El sesgo inductivo del algoritmo en los problemas de clasificación binaria, supone que el espacio que engloba las instancias que compone el dataset se divida en dos subespacios diferente, un subespacio con la clase benigna y otro con la clase maligna(clase 0 y 1 respectivamente).

Como el dataset utilizado en esta investigación es un conjunto de datos desequilibrados con más instancias pertenecientes a la clase maligna que a la clase benigna, la probabilidad de que los vectores vecinos más cercanos de cualquier vector de consulta pertenezcan a la clase mayoritaria es mayor, pero esto no implica que el vector de consulta no este cerca de vectores pertenecientes a la clase minoritaria. Esto implica que el algoritmo K-Nearest Neighbor no toma en cuenta el desequilibrio de clases a la hora de implementarse.

Al igual que con el resto de los algoritmos, el conjunto de datos del dataset es dividido

en dos subconjuntos diferentes, siendo 80 % de las instancias el conjunto de entrenamiento del modelo y el 20 % restantes el de testeo.

El hiperparametro configurado para el algoritmo K-Nearest Neighbor es el valor de K. Este valor permite definir cuantos vecinos verificar para determinar la clasificación de un vector de consulta especifico. Una mala determinación del valor puede conllevar sobreajustes excesivos del modelo o insuficientes.

Este valor está muy influenciado por los valores de entrada, ya que estos si están compuestos por valores atípicos, o ruidos, el algoritmo funcionara mejor con valores de K más elevados.

En esta investigación se elabora un método utilizando validación cruzada para elegir el valor de K óptimo para el conjunto de datos.

Este método da como resultado los porcentajes de precisión utilizando un rango de valores de entre 1 a 77.

```
def best_KValue(X_train, y_train, X_test, y_test, datasets):

    max_attributes = len(list(datasets))
    print("Maximno valor", max_attributes)
    k_range = range(1,  max_attributes)
    scores = []
    for k in k_range:
        knn = KNeighborsClassifier(n_neighbors = k)
        knn.fit(X_train, y_train)
        scores.append(knn.score(X_test, y_test))
        print(k, ":", scores[k-1])
```

**Figura 5.21**: *Método para obtener el K más optimo*

Como se muestra a continuación los resultados obtenidos de este método indican una serie de porcentajes óptimos. Para evitar empates a la hora de realizar el voto mayor de los vectores se escoge un valor impar, con un porcentaje elevado, en este caso el valor K = 3.

| 1 : 0.9796074432832016 | 21 : 0.9709406066785623 | 41 : 0.961509049197043 | 61 : 0.9569207239357634 |
|---|---|---|---|
| 2 : 0.9765485597756819 | 22 : 0.9694111649248024 | 42 : 0.961509049197043 | 62 : 0.9571756308947235 |
| 3 : 0.9796074432832016 | 23 : 0.9694111649248024 | 43 : 0.9607443283201631 | 63 : 0.9571756308947235 |
| 4 : 0.9762936528167219 | 24 : 0.9686464440479226 | 44 : 0.9599796074432833 | 64 : 0.9571756308947235 |
| 5 : 0.9790976293652817 | 25 : 0.9686464440479226 | 45 : 0.9597247004843232 | 65 : 0.9569207239357634 |
| 6 : 0.9770583736936018 | 26 : 0.9671170022941626 | 46 : 0.9592148865664033 | 66 : 0.9564109100178435 |
| 7 : 0.9768034667346418 | 27 : 0.9666071883762427 | 47 : 0.9597247004843232 | 67 : 0.9564109100178435 |
| 8 : 0.976038745857762 | 28 : 0.9658424674993628 | 48 : 0.9592148865664033 | 68 : 0.9559010960999236 |
| 9 : 0.974509304104002 | 29 : 0.9660973744583227 | 49 : 0.9589599796074433 | 69 : 0.9556461891409636 |
| 10 : 0.975019118021922 | 30 : 0.9658424674993628 | 50 : 0.9589599796074433 | 70 : 0.9556461891409636 |
| 11 : 0.974254397145042 | 31 : 0.9653326535814428 | 51 : 0.9579403517716034 | 71 : 0.9556461891409636 |
| 12 : 0.9737445832271221 | 32 : 0.9643130257456028 | 52 : 0.9581952587305633 | 72 : 0.9561560030588835 |
| 13 : 0.9737445832271221 | 33 : 0.9640581187866428 | 53 : 0.9587050726484833 | 73 : 0.9561560030588835 |
| 14 : 0.9724700484323222 | 34 : 0.9635483048687229 | 54 : 0.9576854448126434 | 74 : 0.9559010960999236 |
| 15 : 0.9727249553912822 | 35 : 0.963038490950803 | 55 : 0.9579403517716034 | 75 : 0.9561560030588835 |
| 16 : 0.9709406066785623 | 36 : 0.962273770073923 | 56 : 0.9571756308947235 | 76 : 0.9561560030588835 |
| 17 : 0.9711955136375223 | 37 : 0.963038490950803 | 57 : 0.9564109100178435 | 77 : 0.9564109100178435 |
| 18 : 0.9711955136375223 | 38 : 0.962018863114963 | 58 : 0.9566658169768034 | |
| 19 : 0.9714504205964822 | 39 : 0.9627835839918429 | 59 : 0.9566658169768034 | |
| 20 : 0.9699209788427224 | 40 : 0.961509049197043 | 60 : 0.9576854448126434 | |

El entrenamiento y el testeo del algoritmo se realiza a través del métodos de validación cruzada, consiguiendo un modelo lo más eficiente posible.

```
def create_K(datasets):

    cv = KFold(n_splits=10)
    accuracies = list()
    max_attributes = len(list(datasets))
    depth_range = range(1, max_attributes + 1)


    for depth in depth_range:
        fold_accuracy = []
        knn = KNeighborsClassifier(n_neighbors = 3)
        for train_fold, valid_fold in cv.split(datasets):
            f_train = datasets.loc[train_fold]
            f_valid = datasets.loc[valid_fold]

            model = knn.fit(X = f_train.drop(['type'], axis=1),
                                y = f_train["type"])
            valid_acc = model.score(X = f_valid.drop(['type'], axis=1),
                                y = f_valid["type"])
            fold_accuracy.append(valid_acc)

        avg = sum(fold_accuracy)/len(fold_accuracy)
        accuracies.append(avg)

    return model
```

**Figura 5.22**: *Código validación cruzada K-Nearest Neighbor*

### 5.5.1. Matriz de Confusión

Los resultados de la matriz de confusión del modelo generado es la siguientes.

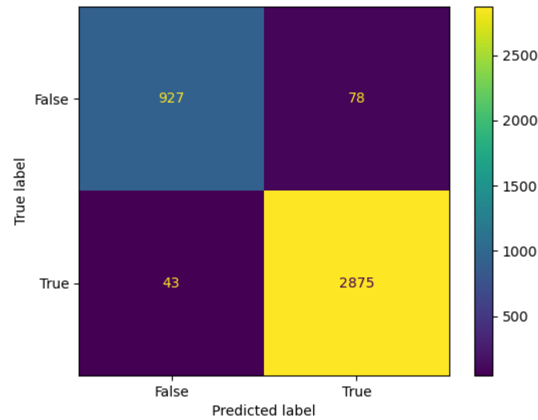



**Figura 5.23**: *Matriz de confusión K-nearest Neighbor*

Al igual que se ha realizado con el resto de los algoritmos, de esta investigación, se aplica la fórmula de exactitud con los datos obtenidos de la matriz de confusión. La fórmula indica que el modelo cuenta con un porcentaje de exactitud del 96,91 % de precisiones correctas frente al total. Al igual que se ha realizado con el resto de los modelos, se realiza las fórmulas de precisión y sensibilidad.

$$(927 + 2875)/(927 + 2875 + 78 + 43) = 3802/3923 = 0,96915 \simeq 96,91\,\%$$

Tras aplicar la fórmula de precisión sobre los datos obtenidos de la matriz, se muestra que el modelo de K-Nearest Neighbors cuenta con un porcentaje de precisión del 97,36 % de probabilidades de que los valores de entrada aplicados en el algoritmo sean clasificado de manera correcta por el mismo.

$$(2875)/(2875 + 78) = 2875/2953 = 0,97358 \simeq 97,35\,\%$$

Al implementar la fórmula de sensibilidad con los datos recopilados, se muestra un mo-

delo bastante sensible con una probabilidad del 98,52 % de capturar la mayor parte de los casos positivos que van entrando y reduciendo la perdida a un 1,48 %.

$$(2875)/(43 + 2875) = 2875/2918 = 0,98526 \simeq 98,53\,\%$$

La fórmula de especifidad indica, que el modelo generado, cuenta con un porcentaje del 92,24 % de dificultad a la hora de obtener falsos positivos en el momento de clasificar los valores de entrada, ya que esta métrica es la proporción entre los casos negativos bien clasificados frente al total de todos los casos negativos.

$$(927)/(927 + 78) = 927/1005 = 0,92238 \simeq 92,24\,\%$$

### 5.5.2. Informe de clasificación

El informe de clasificación muestra los porcentajes de precisión y sensibilidad que tiene el modelo frente a las instancias de cada clase. Como se muestra en el siguiente informe, el modelo cuenta con una precisión del 96 % de probabilidad para detectar los casos positivos pertenecientes a las muestras de la clase benigna, o clase 0, y con una sensibilidad para capturar dichos casos del 92 %, lo que indica que el modelo clasifica de manera correcta gran parte de los datos de entrada de esta clase. De la clase contraria, el modelo cuenta con porcentaje del 97 % de precisión para detectar los casos positivos de esta clase y un porcentaje del 99 % para capturar dichos casos, lo que implica que el modelo detectar como muestra de malware el 97 % de las muestras de este tipo y las clasifica como malware solo el 99 %.

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.96 | 0.92 | 0.94 | 1005 |
| 1 | 0.97 | 0.99 | 0.98 | 2918 |
| accuracy | | | 0.97 | 3923 |
| macro avg | 0.96 | 0.95 | 0.96 | 3923 |
| weighted avg | 0.97 | 0.97 | 0.97 | 3923 |

**Figura 5.24**: *Informe de clasificación K-Nearest Neighbors*

## 5.6. Naive Bayes

Los clasificadores Naive Bayes, son algoritmos de aprendizaje automático basados en la probabilidad condicional y en el teorema de bayes, donde se aplican los conceptos de resultados esperado e independencia, con el fin de clasificar muestras usando la probabilidad de que pertenezca a una determinada clase.

Al igual que con el resto de los algoritmos, el conjunto de datos se ha dividido en dos subconjuntos diferentes. El primero está formado por el 80 % de los datos que componen el conjunto de entrenamiento del algoritmo, el segundo lo componen el 20 % restantes que forman el segundo subconjunto, que integra el conjunto de datos de tipo test.

Para implementar el algoritmo se utiliza el método *GaussianNB* de la biblioteca *sklearn*, se mantiene los valores por defecto y se utiliza validación cruzada para mejorar la optimización del modelo durante el entrenamiento.

```python
def createFold(datasets):
    cv = KFold(n_splits=10)
    accuracies = list()
    max_attributes = len(list(datasets))
    depth_range = range(1, max_attributes + 1)

    for depth in depth_range:
        fold_accuracy = []
        gnb =   gnb = GaussianNB()
        for train_fold, valid_fold in cv.split(datasets):
            f_train = datasets.loc[train_fold]
            f_valid = datasets.loc[valid_fold]

            model = gnb.fit(X = f_train.drop(['type'], axis=1),
                                   y = f_train["type"])
            valid_acc = model.score(X = f_valid.drop(['type'], axis=1),
                                   y = f_valid["type"])
            fold_accuracy.append(valid_acc)

        avg = sum(fold_accuracy)/len(fold_accuracy)
        accuracies.append(avg)

    return model

def statistics(model, x, y):

    pred = cross_val_predict(model, x, y, cv=10)
    prob = cross_val_predict(model, x, y, cv=10, method='predict_proba')

    print('Pred : ', pred)
    print('Prob : ', prob)

    matrix, report = Algorithm.Algorithm_Evaluation(y, pred)
    Algorithm.Curve_ROC(pred, prob, y, 'KNearest Neighbor')
    Algorithm.Curve_Prec_Recall(pred, prob, y, 'KNearest Neighbor')
    # ------------------------------------------------------------------------------------
    cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = matrix, display_labels = [False, True])
    cm_display.plot()
    plt.show()
    #-----------------------------------------------------------------------------------------------------
```

**Figura 5.25**: *Creación del modelo Naive Bayes*

Se divide el dataset en un tercer grupo que englobe el conjunto de datos de validación, para su posterior testeo.

```python
def statistics(model, x, y):

    pred = cross_val_predict(model, x, y, cv=10)
    prob = cross_val_predict(model, x, y, cv=10, method='predict_proba')

    print('Pred : ', pred)
    print('Prob : ', prob)

    matrix, report = Algorithm.Algorithm_Evaluation(y, pred)
    Algorithm.Curve_ROC(pred, prob, y, 'KNearest Neighbor')
    Algorithm.Curve_Prec_Recall(pred, prob, y, 'KNearest Neighbor')
    # ------------------------------------------------------------------------------------
    cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix = matrix, display_labels = [False, True])
    cm_display.plot()
    plt.show()
    #-----------------------------------------------------------------------------------------------------
```

**Figura 5.26**: *Testeo del modelo Naive Bayes*

### 5.6.1. Matriz de Confusión

Se aplica la Matriz de confusión para obtener las métricas de medición del algoritmo.

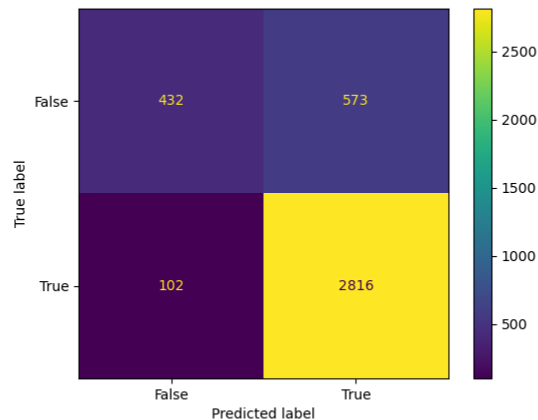



**Figura 5.27**: *Matriz de confusión Naive Bayes*

La exactitud del modelo es del 82,73 % de precisiones correctas sobre el total, indicando que este modelo es capaz de determinar con un porcentaje elevado las características comunes de las clases dentro de las características de la muestra.

$$(432 + 2816)/(432 + 2816 + 102 + 576) = 3248/3926 = 0,82730 \simeq 82,73\,\%$$

La métrica de precisión muestra un modelo con un porcentaje de precisión del 82,02 % para clasificar de manera correcta los valores de entrada del modelo.

$$(2816)/(2816 + 576) = 2816/3392 = 0,83018 \simeq 83,02\,\%$$

Aunque el modelo cuenta con una exactitud y una precisión por debajo del 90 %, el porcentaje de sensibilidad es del 96,50 %, lo que indica que el modelo es capaz de capturar

la mayor parte de casos positivos, reduciendo la perdida a un 3,5 %

$$(2816)/(102 + 2816) = 2816/2918 = 0,96504 \simeq 96,50\,\%$$

El porcentaje de especifidad está por debajo del 50 %, con un 42,98 %, lo que indica que existe un 57,01 % de probabilidades de producirse un falso positivo a la hora de clasificar una muestra de entrada.

$$(432)/(432 + 573) = 432/1005 = 0,42985 \simeq 42,98\,\%$$

### 5.6.2. Naive Bayes

Como se muestra en el siguiente informe, el modelo cuenta con una precisión del 81 % de probabilidad para detectar los casos positivos clasificados dentro de la clase benigna, o clase 0, y una sensibilidad del 43 %, lo que indica que la mayor parte de las muestras clasificadas como archivos benignos se trata de falsos positivos. En cambio para la clase maligna, o clase 1, el porcentaje de precisión para detectar las muestras de tipo malware es del 83 % y con una sensibilidad del 97 %, lo que se puede interpretar que la mayor parte de muestras detectadas como malignas son correctas

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.81 | 0.43 | 0.56 | 1005 |
| 1 | 0.83 | 0.97 | 0.89 | 2918 |
| accuracy | | | 0.83 | 3923 |
| macro avg | 0.82 | 0.70 | 0.73 | 3923 |
| weighted avg | 0.83 | 0.83 | 0.81 | 3923 |

**Figura 5.28**: *Informe de clasificación Naive Bayes*

# Capítulo 6

# Resultados y discusión

Este capítulo tiene como objetivo determinar que algoritmo de machine learning, expuesto en esta investigación, es el apropiado para resolver el problema de clasificación. Se utiliza los modelos generados en el capítulo anterior y las métricas de evaluación, *Curva de precisión/sensibilidad*(o *Curva PR*) y *Curva ROC* para comparar los resultados y concluir cual es el más óptimo para esta tarea.

## 6.1. Resultado

Como se mencionó en el capítulo 3, *tipos de medición*, la curva de precisión/sensibilidad traza el valor predictivo positivo(sensibilidad) frente a la tasa de verdadero positivo(precisión) de un determinado modelo. Esta curva muestra gráficamente la relación entre ambas métricas para determinar la calidad de un modelo a lo largo del rango de precisión y sensibilidad, utilizando el paradigma lógico de a mayor sensibilidad menor precisión y viceversa.

Para determinar el rendimiento del modelo, la curva de precisión/sensibilidad establece una zona inicial predeterminada, donde para alcanzar una sensibilidad del 100 % la precisión del modelo debe reducirse al 66,5 %, ya que en este rango se puede producir predicciones falsas positivas.

Para obtener con mayor precisión el rendimiento predictivo del modelo, se calcula la

métrica AUC-PR, que proporciona dicho valor e influye en el área bajo la curva.

Los resultados obtenidos de la curva PR de los modelos generados por los algoritmos, nos muestran unos resultados bastante óptimos para el problema planteado, siendo los modelo con menor rendimiento predictivo, los algoritmos de Naive Bayes y Redes neuronales.

El algoritmo Naive Bayes cuenta con rendimiento del 96,6 %, llegando a alcanzar una sensibilidad del 90 % sin apenas predicciones falsas positivas.

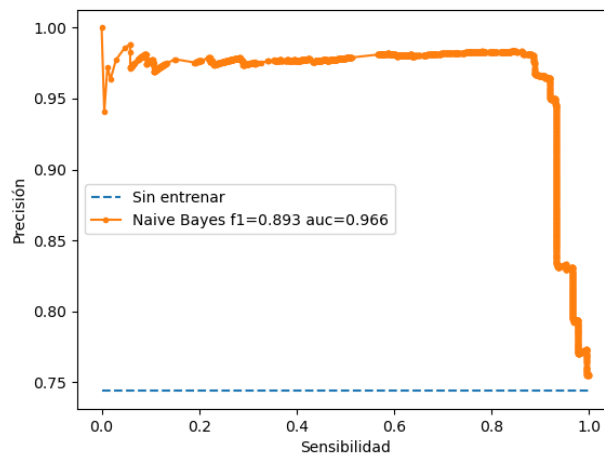



**Figura 6.1**: *Curva PR - Naive Bayes*

En cambio el algoritmo de red neuronal tiene un rendimiento del 96,9 %, alcanzando una sensibilidad del 90 %, con un aumento exponencial de casos falsos positivos.

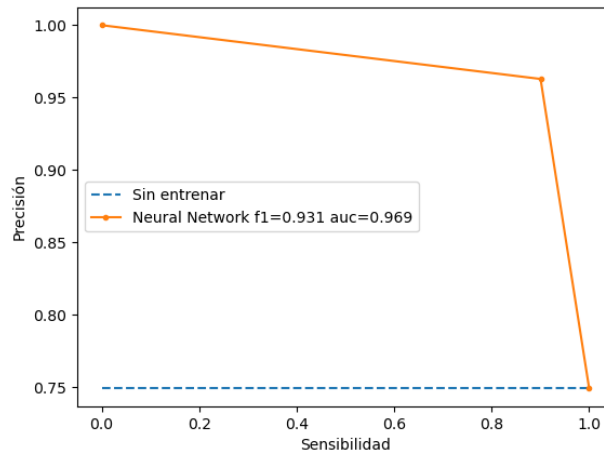



**Figura 6.2**: *Curva PR - Redes Neuronales*

Otro de los algoritmo que aumenta exponencialmente el número de casos de falsos positivos, hasta alcanzar una alta sensibilidad, es el algoritmo de K-Nearest Neighbor. Este algoritmo tiene un rendimiento del 97,9 % y alcanza una sensibilidad cercana al 100 % con un aumento exponencial de los falsos positivos.

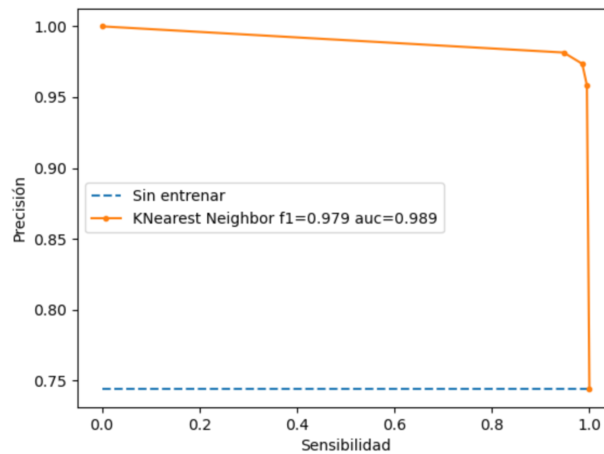



**Figura 6.3**: *Curva PR - K-Nearest Neighbor*

El algoritmo de Bosque aleatorio bajo las técnicas de aprendizaje Bagging y Boosting, muestran un rendimiento del 99,7 % y 99,4 % respectivamente, alcanzando una sensibilidad cercana al 100 % sin predicciones falsas positivas.

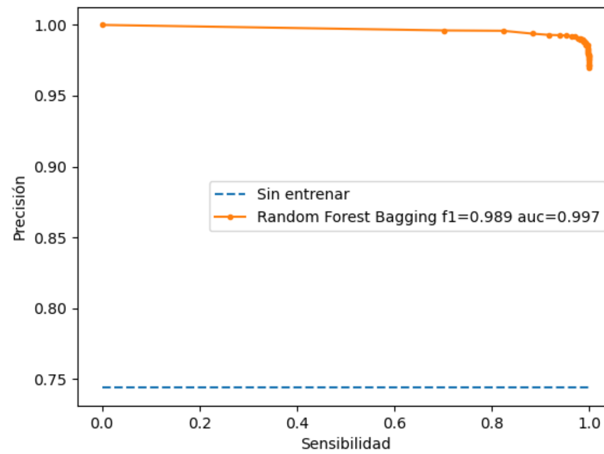



**Figura 6.4**: *Curva PR - Random Forest Bagging*

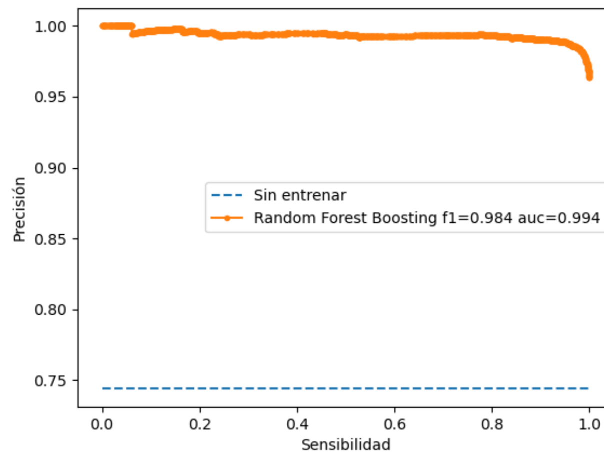



**Figura 6.5**: *Curva PR - Random Forest Boosting*

Sin embargo, el modelo con mejor rendimiento mostrado por la curva de precisión/sensibilidad,

es el generado por el algoritmo de árbol de decisión. Este modelo cuenta con un rendimiento del 99,7 % y un porcentaje de sensibilidad mayor al obtenido con los bosques aleatorios, rozando el 100 % de casos positivos sin resultados falsos negativos.

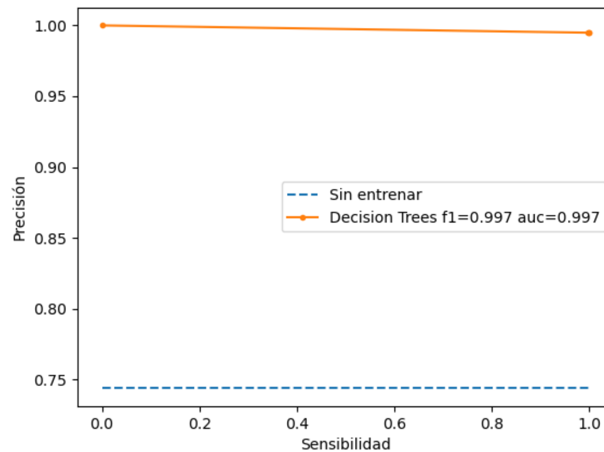



**Figura 6.6**: *Curva PR - Decision Trees*

Debido a que la curva de precisión y sensibilidad discrimina los valores de verdaderos negativos, afectando a la especifidad de los modelos creados. Se utiliza la curva ROC, como métrica más veraz para establecer una relación entre precisión y sensibilidad de los modelos.

La curva ROC marca con una diagonal de 45 grados el resultado obtenido de un modelo de clasificación aleatoria, donde el rango de verdaderos positivos y falsos positivos es del 50 %. Esta diagonal se utiliza como medida de rendimiento del modelo evaluado.

Al igual que con la curva PR, se calcula el AUC de la curva ROC. El AUC de la curva ORC equivale a la probabilidad de que una instancia positiva, elegida al azar, tenga una clasificación más alta que una instancia negativa al azar.

Se comienza el analisis con el algoritmo Naive Bayes, clasificado por la curva PR como uno de los modelos menos eficaces generados en esta investigación, para generar la curva

ROC y calcular el valor AUC. Este dato muestra un modelo con un nivel de probabilidad del 92,6 % para distinguir entre clases positivas y negativas diferenciando ambas en pruebas de testeo. Este es un valor muy alto alejándose de los modelos clasificadores aleatorio (AUC: 50 %) que no tienen la capacidad de discriminación para distinguir entre clases positivas y negativas.

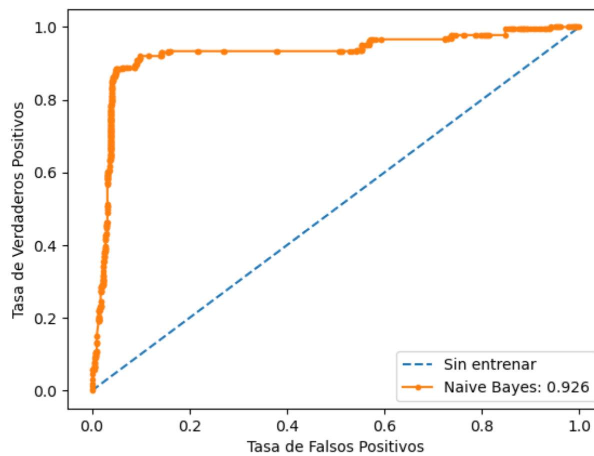



**Figura 6.7**: *Curva ROC - Naive Bayes*

El siguiente algoritmo es la red neuronal, un algoritmo que muestra un mejor rendimiento en la curva PR, que el algoritmo Naive Bayes, con un 96,6 % de probabilidad. Al aplicarse gráficamente la curva ROC, este algoritmo muestra un AUC del 89,8 % de capacidad de discriminación entre las clases positivas, de las clases negativas, lo que expone un modelo con un porcentaje más elevado para conseguir falsos positivos a la hora de clasificar las muestras.

La curva ROC muestra un modelo clasificatorio, en el algoritmo K-Nearest Neighbor, con una porcentaje de discriminación, entre las clases positivas y negativas, del 96,9 %, lo que indica un modelo con una alto rendimiento y unas altas probabilidades para clasificar las

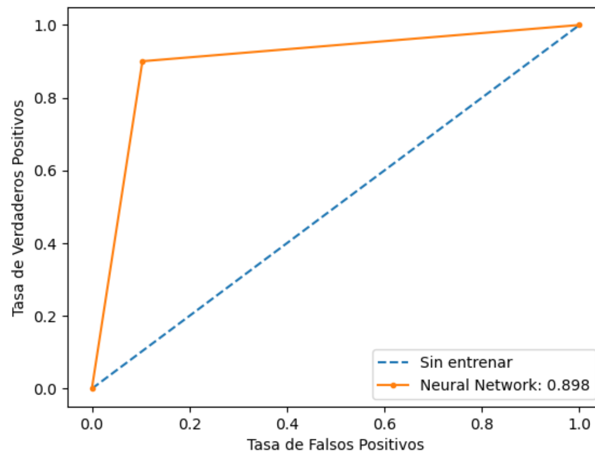



**Figura 6.8**: *Curva ROC - Neural Network*

muestras de manera correcta, dejando un porcentaje bastante bajo al a hora de clasificaciones erróneas de falsos positivos.

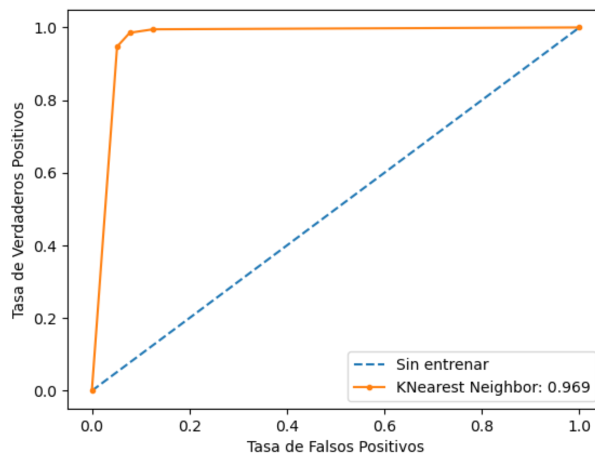



**Figura 6.9**: *Curva ROC - K-Nearest Neighbor*

Los modelos clasificadores de bosques aleatorios, bajo las técnicas de aprendizaje Bagging y Boosting, muestran unos modelos con una porcentaje de discriminación rozando la efectividad del clasificador, con un 99,3 % y 98,9 % respectivamente. Comparando ambas

técnicas de aprendizaje podemos concluir que los modelos de bosques aleatorios generados bajo la técnica bagging, son más eficientes que los generados por las técnicas boosting, ya que estos últimos cuentan con un mayor porcentaje de obtener más falsos positivos a la hora de clasificar las muestras.

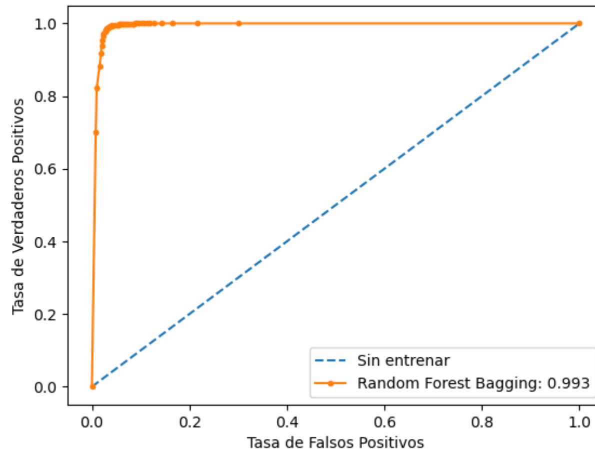



**Figura 6.10**: *Curva ROC - Random Forest Bagging*

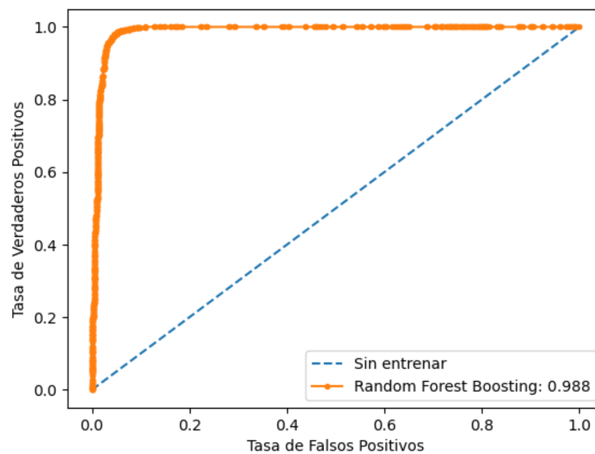



**Figura 6.11**: *Curva ROC - Random Forest Boosting*

El último modelo evaluado, es el modelo generado por el algoritmo de árbol de decisión. Este modelo muestra un rendimiento elevado con la curva PR y un porcentaje aproximado del 99 % de probabilidades de obtener los casos positivos. Evaluando ahora, el mismo modelo bajo la curva ROC, se puede ver un modelo con unos resultados rozando el 100 % de predicción y sensibilidad, lo que indica un modelo capaz de clasificar las muestras, con un porcentaje del 99,3 % de probabilidad, de manera correcta, disminuyendo los casos de falsos positivos en un 0,7 %.

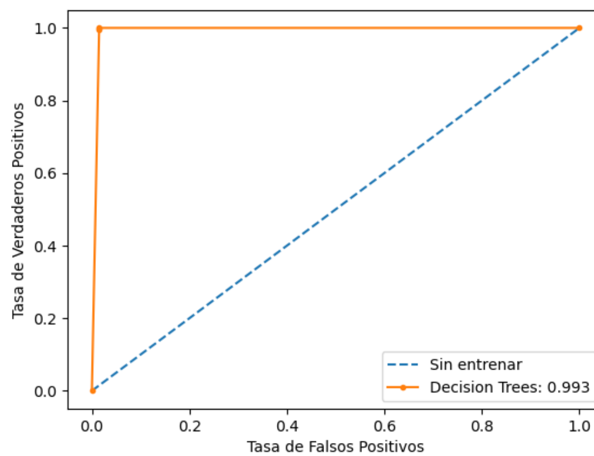



**Figura 6.12**: *Curva ROC - Decisión Trees*

## 6.2. Discusión

Tras evaluar los seis modelos generados por los cinco algoritmos de machine learnning a través de las métricas de Matriz de confusión, curvas de precisión/sensibilidad y curva ROC, podemos concluir que el mejor modelo para clasificar archivos en dos categorías, malware o benignas, es el modelo generado por al algoritmo de árboles de decisión con unos porcentajes de precisión y sensibilidad elevados (99,49 % y 100 % respectivamente ), lo que nos permite clasificar los archivos de una manera correcta, con una escasa probabilidad de obtener falsos

positivos y con un nivel de exactitud del 99,61 %. Además es uno de los algoritmos más efectivos mostrados por la métrica de curva ROC con un resultado del 99,3 %.

Otros de los algoritmos que mostraron unos resultados bastante buenos a la hora de clasificar las muestra y con unos niveles de efectividad similares a los obtenidos con el algoritmo de árboles de decisión, son los modelos generados por los bosques aleatorios, tanto con la técnica bagging como boosting, mostrando unos índices de precisión y sensibilidad del 98,078 % y 99,72 % respectivamente para la técnica bagging y 97,57 % y 99,21 % respectivamente para la técnica boosting, siendo el modelo generado por la técnica bagging el más efectivo de los dos con unos porcentajes del 99,3 % y unos índices de precisión y sensibilidad más elevados que el aumento de gradiente.

El modelo generado por el algoritmo K-Nearest Neighbor, es un modelo "bueno"donde las métricas muestran unos índices de precisión y sensibilidad del 97,35 % y 98,53 % respectivamente, exponiendo un modelo menos eficiente que los generados por los algoritmos de árbol de decisión y bosque aleatorio con un porcentaje de efectividad del 96,9 %, lo que indica que los índices de clasificación que tiene este modelo para clasificar las muestras de manera correcta son inferiores a los modelos de los algoritmos anteriores, con un índice más elevado para obtener falsos positivos en la clasificación de algunas muestras. Aun así, las métricas recopiladas de este modelo están en un rango “decente” para poder clasificar la mayor parte de las muestras de manera correcta. Lo mismo ocurre con el modelo generado por la red neuronal donde los porcentajes de precisión y sensibilidad son elevados (95,02 % y 92,28 % respectivamente) pero no perfectos, debido a que este modelo contiene un porcentaje aproximado del 14 % de probabilidad para obtener falsos positivos en la clasificación de archivos.

Las métricas obtenidas del modelo generado por el algoritmo Naive Bayes indican un modelo con una mayor sensibilidad en comparación con la precisión, con un 96,50 % de

sensibilidad y un 83,02 % de precisión. Debido a que nos interesa un modelo de clasificación que tenga unos porcentajes de falsos positivos cercanos a 0, el porcentaje de especifidad nos indica que este modelo tiene una probabilidad del 57,01 % de producirse un falso positivo a la hora de clasificar una muestra de entrada, debido a este hecho, este algoritmo es el menos preparado para clasificar muestras de manera correcta entre clases malignas y benignas.

Se puede concluir que los algoritmos de árboles de decisión y bosques aleatorios son los mas adecuados a la hora de clasificar programas en clases benignas o malignas, en un ambiente de laboratorio con modelos entrenados de un dataset con un desequilibrio de clase con mayor muestras de archivos malignos que benignos, lo que deja la puerta abierta a estudios más profundos con diferentes tamaños de clase.

También se concluye que el algoritmo menos adecuado para esta clase de problemas es el de Naive Bayes, el nivel de precisión y especifidad expone un modelo deficiente para la clasificación de software, detectando la naturaleza del mismo y su clasificación con altos porcentajes de error.

# Capítulo 7

# Conclusión y trabajos futuros

## 7.1. Conclusión

Los resultados obtenidos de los diversos algoritmos de machine learning son prometedores, mostrado modelos bastante eficientes para la resolución al problema planteado en esta investigación. El estudio de los diferentes métodos de detección actual de archivos maliciosos, los diferentes algoritmos de machine learning de clasificación, la obtención y extracción de características de las muestras obtenidas, la implementación y ajuste de los diversos algoritmos de machine learning y la creación de un sistema web que permita la detección de archivos a menor escala, a permitido llegar a los siguientes resultados en la investigación.

- Se ha logrado el objetivo principal de construir un sistema capaz de detectar la naturaleza de un determinado archivo, utilizando para ello modelos generados por algoritmos de machine learning, con unos índices de error de un 0,7 %.
- Se consigue el objetivo secundario de la investigación, desarrollar seis modelos de aprendizaje automático de cinco tipos de algoritmos diferentes (red neuronal, árboles de decisión, bosques aleatorios, K-Nearest Neighbor y Naive bayes), con un porcentaje de precisión para clasificar muestras superior al 80 %.
- Se realiza un estudio de las técnicas de detección de archivos malware actuales y una investigación de la literatura publicada de técnicas de aprendizaje automático.

- Se realiza una investigación de las posibles características que se puedan utilizar para determinar si un archivo puede clasificarse como benigno o malware.
- Se realiza una investigación de los algoritmos más adecuados para resolver el problema planteado, al igual que de la preparación de las características necesarias para poder ser tratadas.

Por otro lado, durante el proceso de implementación de la investigación se han presentado algunas situaciones que han desembocado en las siguientes conclusiones.

- Las características obtenidas de las cabeceras PE de los archivos ejecutables, son datos en formato hexadecimal, lo que es un inconveniente utilizarlas directamente en los algoritmos de machine learning, ya que estos solo pueden tratar características decimales. Se ha implementado un método de virtualización para virtualizar la información hexadecimal de entrada, a través de la biblioteca pefile, a características decimales que puedan ser entendidas por los algoritmos de machine learning.
- Para aumentar la precisión y disminuir el número de falsos negativos que puede acarrear un modelo sobreajustado, se ha procedido a implementar métodos de división de dataset en tres grupos diferentes; grupos de entrenamiento, grupos de testeo y grupos de validación, en un porcentaje de 60 %, 20 %, 20 %, respectivamente. Este método de división es grupal para todos los algoritmos y proporciona unos resultados inferiores a los obtenidos con unos grupos de entrenamiento y testeo (80 %,20 % respectivamente) lo que ha llevado a concluir, que los algoritmos que en un principio no contaban con las pruebas de testeo final con los grupos de validación, utilizan el grupo de datos de testeo para configurar de manera automática los parámetros internos, mientras las pruebas con el conjunto de validación muestras la eficiencia y las métricas del modelo en un entorno real, con datos que el modelo no ha utilizado en el momento de su implementación.

- Además del uso del conjunto de validación del dataset para obtener resultados más fieles en entornos más reales. Se utiliza otros métodos de aprendizaje que favorezcan la implementación de modelos más eficaces, como las técnicas Bagging y Boosting con el algoritmo de Bosque aleatorio, o la implementación de la validación cruzada en algoritmos de árbol de decisión o K-Nearest Neighbor.

- Los resultados obtenidos de la matriz de confusión, la curva ROC y la curva PR, exponen que el algoritmo más eficiente a la hora de detectar archivos maliciosos es el modelo generado por el algoritmo de árbol de decisión, ya que cuenta con un porcentaje de sensibilidad y de precisión de más del 99 % y con una tasa inferior al 1 % a la hora de obtener falsos positivos en la clasificación de archivos. De manera inversa los modelos generado por los algoritmos de Naives Bayes y Redes neuronales, son los menos eficientes la hora de poder clasificar los diversos archivos. Los resultados de estos dos últimos modelos no son resultados malos y están dentro de los umbrales de modelos eficientes, superando del 80 % de eficiencia, pero debido a la criticidad del problema planteado en esta investigación y con el fin de obtener los menores porcentajes de falsos positivos a la hora de clasificar los archivos, los algoritmos más precisos para este fin son los modelos creados por los algoritmos de árbol de precisión y bosque aleatorio.

## 7.2. Trabajos Futuros

El desarrollo de esta investigación ha concluido con la construcción de un prototipo de sistema web capaz de clasificar las muestras ejecutables de formato .exe, en archivos benignos y archivos malignos. Esta investigación junto con el prototipo es un primer paso para la construcción de softwares capaces de analizar sistemas informáticos completos en busca de archivos malware, y dar un paso tecnológica en el campo de la ciberseguridad al implementar algoritmos capaces de detectar nuevas amenazas de manera más eficiente.

Esta investigación expone teorías y evidencias ya explicadas en la literatura, aplicando

esta información a la práctica, con un sistema web diseñado para clasificar archivos en dos categorías, benignos y malignos. A futuro este prototipo, será capaz de incorporarse a sistemas más complejos, como equipos informáticos o sistemas de red privadas, siendo un complemento a otras herramientas de ciberseguridad implementadas en estos ecosistemas cibernéticos, como puede ser los IPS, IDS o cortafuegos.

# Bibliografía

[1] https://github.com/iosifache/dikedataset/tree/main/files. Ultimo acceso: 22/09/2022.

[2] https://github.com/ytisf/thezoo. Ultimo acceso: 22/09/2022.

[3] Sadhik Shaik Chandrasekhar Rao Jetti, Rehamatulla Shaik. Disease prediction using naïve bayes - machine learning algorithm. In *International Journal of Science and Healthcare Research*, 2021.

[4] Aurelien Geron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, volume 2nd Ed. O'Reilly Media, Inc, USA, 2019.

[5] Deguang Kong Guanhua Yan, Nathan Brown. Exploring discriminatory features for automated malware classification. 7967, 2013.

[6] Hanyang Li Haixing Yin, Fan Fan and Ting Fung Lau. The importance of domain knowledge. 2020.

[7] Connor Hayes. Neural networks. In *Neural Networks*, 2019.

[8] Rafiqul Islam, Ronghua Tian, Lynn Batten, and Steve Versteeg. Classification of malware based on string and function feature selection. pages 9–17, 2010.

[9] J. J. Jaccard and S. Nepal. A survey of emerging threats in cybersecurity. 80:973–993, 2014.

[10] kenneth Leung. *Micro, Macro Weighted Averages of F1 Score, Clearly Explained.* Towards Data Science, 2022.

[11] S. Kim. *PE Header Analysis for Malware Detection.* PhD thesis, San José State University, 2018.

[12] Samuel Kim. Pe header analysis for malware detection. In *Master's Project, San Jose State University*, 2018.

[13] Teuvo Kohonen and Panu Somervuo. Self-organizing maps of symbol strings. *Neurocomputing*, 21(1):19–30, 1998.

[14] Jia Li. Classification/decision trees. In *Department Of Statistics, The Pennsylvania State University*, 2019.

[15] Zane A. Markel. Machine learning based malware detection. In *Trident Scholar Project rept. no. 440*, 2015.

[16] Andrew Honig Michael Sikorski. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, volume 1nd Ed. No Starch Press, 2012.

[17] Tom M. Mitchell. Generative and discriminative classifiers: Naive bayes and logistic regression machine learning. In *Machine Learning, T.M*, 2005.

[18] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective (Adaptive Computation and Machine Learning series).* The MIT Press, 2012.

[19] Marlon Núñez. The use of background knowledge in decision treeinduction. In *Machine Learning*, 1991.

[20] Edward Raff, Jared Sylvester, and Charles Nicholas. Learning the pe header, malware detection with minimal domain knowledge. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pages 121–132, 2017.

[21] Nurali Rahimov. Random forest applied multivariate statistics. In *IT Technology and Computer science.*, 2012.

[22] Jaime Ramírez. *Curvas PR y ROC*. Medium, 2018.

[23] Igor Santos, Yoseba Penya, Jaime Devesa, and Pablo Bringas. N-grams-based file signatures for malware detection. pages 317–320, 01 2009.

[24] Neha Seth. Estimation of neurons and forward propagation in neural net. In *International Journal of Science and Healthcare Research*, 2021.

[25] Thomas Stibor. A study of detecting computer viruses in real-infected files in the n-gram representation with machine learning methods. pages 509–519, 2010.

[26] Sabu Emmanuel Tony Thomas, Athira P. Vijayaraghavan. *Machine Learning Approaches in Cyber Security Analytics*. Springer, 2020.

[27] Paul E. Utgoff. Incremental induction of decision trees. In *Machine Learning*, 1989.

[28] Parag Verma, Shayan Anwar, Shadab Khan, and Sunil B Mane. Network intrusion detection using clustering and gradient boosting. In *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pages 1–7, 2018.

[29] Joel Yonts. Attributes of malicious files. 2012.

[30] Jean-Yves Ramel Zeina Abu-Aisheh, Romain Raveaux. Efficient k-nearest neighbors search in graph space. In *Pattern Recognition Letters, Elsevier*, 2018.

# Apéndice A

# Encabezados PE

| Encabezado DOS | |
|---|---|
| e_magic | Firma que marca el archivo como ejecutable de MS-DOS. |
| e_cblp | Longitud del tamaño del ejecutable |
| e_cp | Tamaño del ejecutable |
| e_crlc | Ubicación del ejecutable |
| e_cparhdr | Tamaño del encabezado |
| e_minalloc | Mínimo de párrafos adicionales necesarios |
| e_maxalloc | Máximo de párrafos adicionales necesarios |
| e_ss | Valor SS inicial |
| e_sp | Valor inicial del SP |
| e_csum | suma de control |
| e_ip | Valor IP inicial |
| e_cs | Valor CS inicial |
| e_lfarlc | Dirección de archivo de la tabla de reubicación |
| e_ovno | Número de superposición |
| e_oemid | Identificador OEM |
| e_oeminfo | Información del OEM |
| e_lfanew | Dirección de archivo del nuevo encabezado |

| Encabezado PE | |
|---|---|
| Machine | Número que identifica el tipo de máquina de destino |
| NumberOfSections | Indica el tamaño de la tabla de secciones, que sigue inmediatamente a los encabezados. |
| TimeDateStamp | Indica la fecha y la hora de cuando se creó el archivo. |
| PointerToSymbolTable | Desplazamiento del archivo de la tabla de símbolos COFF, o cero si no hay tabla de símbolos COFF presente |

| Encabezado Opcional | |
|---|---|
| Magic | Identifica el estado del archivo de imagen |
| MajorLinkerVersion | El número de versión principal del enlazador |
| MinorLinkerVersion | El número de versión secundaria del enlazador |
| SizeOfCode | El tamaño de la sección de código o la suma de todas las secciones de código si hay varias secciones |
| SizeOfInitializedData | El tamaño de la sección de datos inicializados, o la suma de todas esas secciones si hay varias secciones de datos |
| SizeOfUninitializedData | El tamaño de la sección de datos no inicializados (BSS), o la suma de todas esas secciones si hay varias secciones BSS |
| AddressOfEntryPoint | La dirección del punto de entrada relativa a la imagen base cuando el archivo ejecutable se carga en la memoria |
| BaseOfCode | La dirección que es relativa a la imagen base de la sección de comienzo de código cuando se carga en la memoria |
| ImageBase | La dirección del primer byte de la imagen cuando se carga en la memoria |
| SectionAlignment | La alineación (en bytes) de las secciones cuando se cargan en la memoria |
| FileAlignment | El factor de alineación (en bytes) que se usa para alinear los datos sin procesar de las secciones en el archivo de imagen |
| MajorOperatingSystemVersion | El número de versión principal del sistema operativo requerido |
| MinorOperatingSystemVersion | El número de versión menor del sistema operativo requerido |
| MajorImageVersion | El número de versión principal de la imagen |
| MinorImageVersion | El número de versión secundaria de la imagen |
| MajorSubsystemVersion | El número de versión principal del subsistema |
| MinorSubsystemVersion | El número de versión secundaria del subsistema |

| Encabezado Opcional | |
|---|---|
| SizeOfHeaders | Tamaño de la cabecera incluyendo el código auxiliar de MS-DOS, encabezado PE y encabezados de sección |
| CheckSum | Suma de comprobación del archivo de imagen |
| SizeOfImage | Tamaño (en bytes) de la imagen, incluidos todos los encabezados |
| Subsystem | El subsistema necesario para ejecutar esta imagen |
| DllCharacteristics | Caracteristicas DLL |
| SizeOfStackReserve | El tamaño de la pila que se va a reservar |
| SizeOfStackCommit | El tamaño de la pila que se va a confirmar |
| SizeOfHeapReserve | El tamaño del espacio local que se va a reservar |
| SizeOfHeapCommit | El tamaño del espacio local que se va a confirmar |
| LoaderFlags | Reservado, debe ser cero. |
| NumberOfRvaAndSizes | Número de entradas de directorio de datos en el resto del encabezado opcional |

| Encabezados de Sección | |
|---|---|
| SectionsLength | Longitud total de la cabecera de sección |
| SectionMinEntropy | Número mínimo de entropía de la cabecera de sección |
| SectionMaxEntropy | Número máximo de entropía de la cabecera de sección |
| SectionMinRawsize | Número mínimo del tamaño de la sección (para los archivos de objeto) o del tamaño de los datos inicializados en el disco (para los archivos de imagen) |
| SectionMaxRawsize | Número máximo del tamaño de la sección (para los archivos de objeto) o del tamaño máximo de los datos inicializados en el disco (para los archivos de imagen) |
| SectionMinVirtualsize | Número mínimo del tamaño de la sección cuando se carga en la memoria |
| SectionMaxVirtualsize | Número máximo del tamaño de la sección cuando se carga en la memoria |
| SectionMaxPhysical | Número máximo de la dirección física del archivo |

| Encabezados de Sección | |
|---|---|
| SectionMinPhysical | Número mínimo de la dirección física del archivo |
| SectionMaxVirtual | Número máximo de dirección virtual del archivo |
| SectionMinVirtual | Número mínimo de dirección virtual del archivo |
| SectionMaxPointerData | Número máximo del puntero de archivo a la primera página de la sección del archivo |
| SectionMinPointerData | Número mínimo del puntero de archivo a la primera página de la sección del archivo |
| SectionMaxChar | Número máximo de marcas que describen las características de la sección |
| SectionMainChar | Número mínimo de marcas que describen las características de la sección |
| DirectoryEntryImport | DLL importadas por el ejecutable |
| DirectoryEntryImportSize | Tamaño de las DLL importadas por el ejecutable |
| DirectoryEntryExport | DLL exportadas por el ejecutable |
| ImageDirectoryEntryExport | En caso de imágenes ejecutables, dirección del primer bytes de la sección relativa a las imágenes exportada desde memoria |
| ImageDirectoryEntryImport | En caso de imágenes ejecutables, dirección del primer bytes de la sección relativa a las imágenes importadas desde memoria |
| ImageDirectoryEntryResource | En caso de imágenes ejecutables, son los recursos de entrada del directorio de imágenes |
| ImageDirectoryEntryException | En caso de imágenes ejecutables, directorio de excepciones |
| ImageDirectoryEntrySecurity | En caso de imágenes ejecutables, directorio de seguridad |

# Apéndice B

# Guía de Usuario

## B.1. Introducción

La siguiente guía de usuario tiene como objetivo mostrar las principales funcionalidades del servicio web creado a lo largo del desarrollo de esta investigación. Este servicio se encuentra en fase de desarrollo y puede darse casos de excepción en el uso de ciertas funcionalidades o casos no probados.

## B.2. Login de Usuario

Para acceder al servicio web, se debe acceder a través del siguiente enlace:

- https://machinelearninganalyzer.herokuapp.com/

Tras seleccionar el enlace se muestra la siguiente pantalla.

**Figura B.1**: *Pantalla de Login del servidor*

Para poder acceder a las funcionalidades del servicio, debe estar registrado en el mismo. Para ello seleccione el botón "*Sign in*" situado en la parte superior derecha de la pantalla. Una vez seleccionado se mostrara la siguiente pantalla.



**Figura B.2**: *Pantalla de registro*

Rellene los campos requeridos y seleccione el botón "*Sign in*".

Una vez que este registrado, complete las credenciales de email y contraseñas en la página de login y seleccione el botón "*Login*".

Una vez accedido al servicio, se mostrara la siguiente pantalla, mostrando las diferentes funcionalidades y algoritmos en el panel superior.

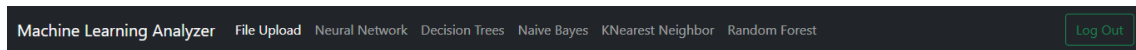



**Figura B.3**: *Pantalla principal*

## B.3. File Upload

*machinelearninganalyzer* cuenta con la funcionalidad de enriquecer la base de datos que compone el dataset y descargarlo desde la base de datos en formato .csv.

### B.3.1. Descargar excel del dataset

Seleccione la pestaña "*File Upload*", del panel superior de la pantalla principal. Una vez seleccionada se mostrara la siguiente pantalla.

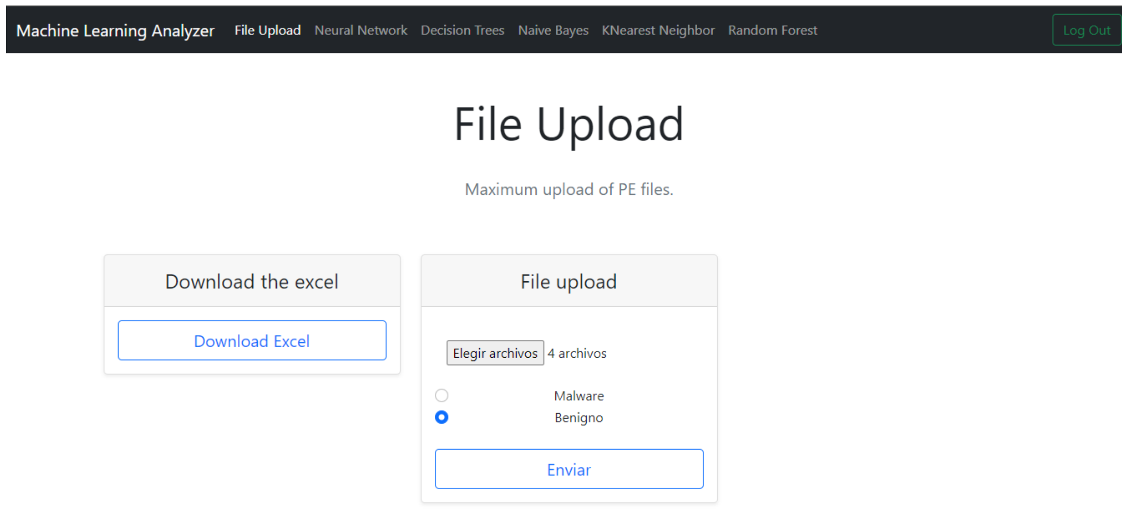



**Figura B.4**: *Pantalla File Upload*

Seleccione el botón "*Download Excel*". El dataset se descarga en formato .csv, directamente en la carpeta de dropbox asociado a la página.

### B.3.2. Subida de archivos al dataset

Se permite realizar subidas máximas de archivos, en formato .exe, para enriquecer la base de datos. Para realizar esta funcionalidad se debe seleccionar el botón "*Elegir archivos*" del apartado "*File Upload*" y seleccionar los archivos que se desea subir.

Tras seleccionar los archivos, se debe marcar la naturaleza de los mismos antes de subirlos. Si estos archivos son de tipo Benigno se deberá seleccionar la categoría "*Benigno*"; en cambio si se trata de archivo malignos se debe seleccionar la categoría "*Malware*".

Una vez seleccionado los archivos que se desean subir y la categoría a la que pertenecen, se debe seleccionar el botón "*Enviar*".

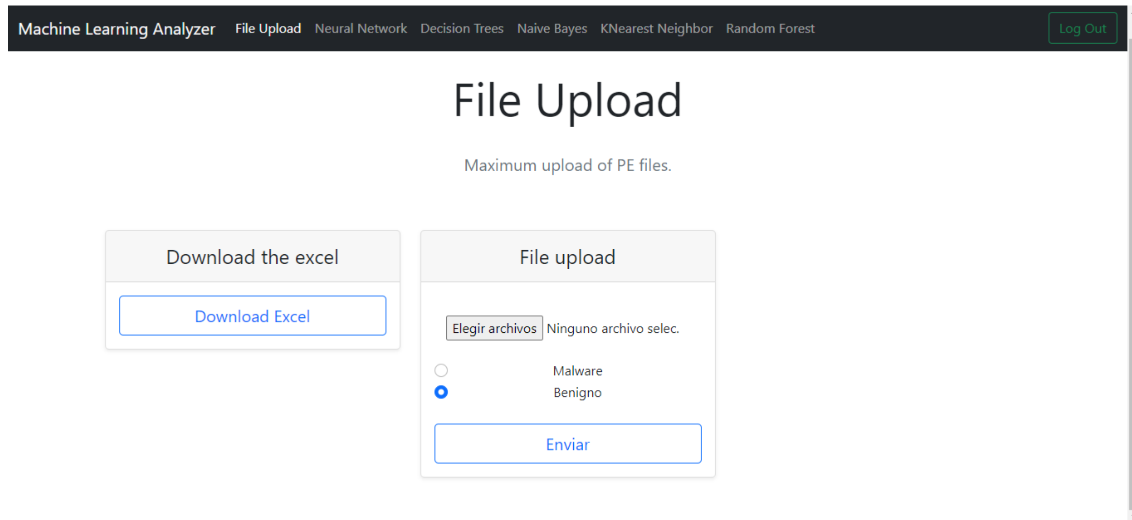



**Figura B.5**: *Pantalla File Upload, Subida de archivos*

Si la subida del archivo ha ido correctamente se mostrara un mensajes indicando que la subida se ha realizado satisfactoriamente. En caso contrario se muestra la excepción producida.

## B.4. Algoritmos de inteligencia artificial

El uso de los modelos generados por los algoritmos de inteligencia artificial se compone de 3 funcionalidades diferentes: la creación del modelo, el informe de clasificación de este y el escaneo de archivos.

Para acceder al algoritmo que se desea probar, se debe seleccionar la pestaña correspondiente en el panel principal.

Tras seleccionar el algoritmo se muestra la siguiente pantalla.

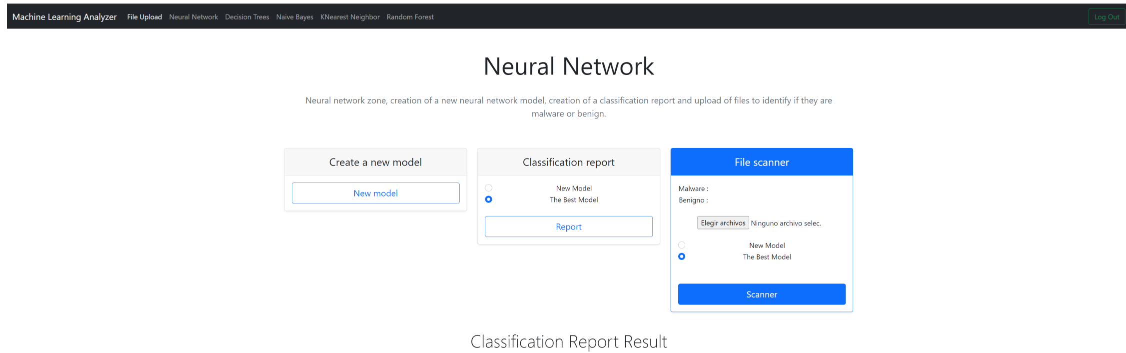



**Figura B.6**: *Pantalla Neural Network*

### B.4.1. Generar un nuevo modelo

Si se desea generar un nuevo modelo probabilístico, se debe seleccionar el botón "*New Model*" situado en el apartado "*Create a new model*" a la izquierda de la pantalla. Esta funcionalidad recoge el dataset almacenado en la base de datos y genera un nuevo modelo probabilístico del algoritmo, que se utilizara para el resto de las funcionalidades del página.

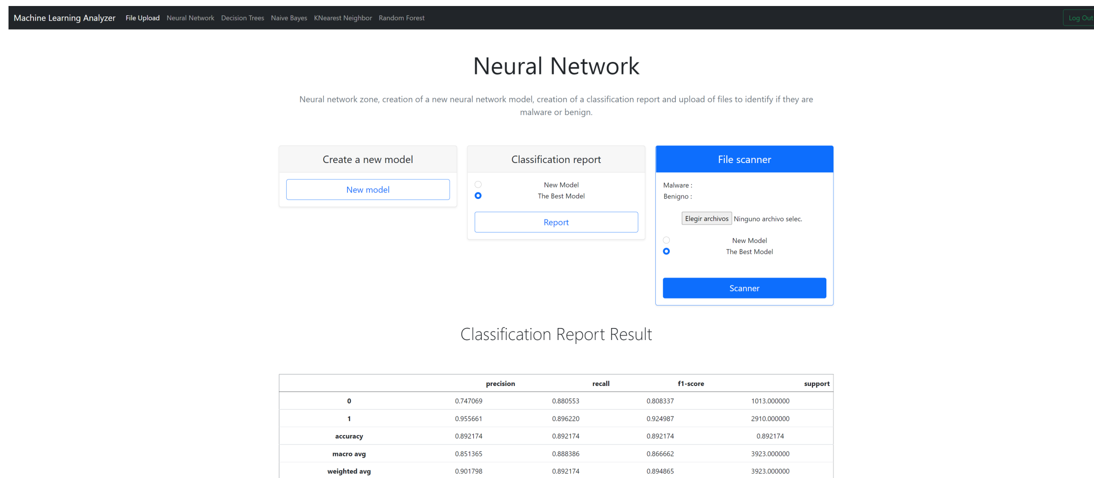
### B.4.2. Informe de Clasificación

El informe de clasificación corresponde con las métricas del modelo almacenado en el servidor. En este apartado se explica la segunda funcionalidad de la parte de los algoritmos, situado en el centro de la pantalla.

Como se puede ver en la siguiente imagen, se puede seleccionar dos tipos de modelos que están actualmente almacenados en el servidor. El nuevo modelo o "*New Model*" corresponde con el modelo generado tras seleccionar el botón "*New Model*" del apartado anterior. Tras generar un nuevo modelo este tiene métricas diferentes a los mostrados en esta investigación ya sea por naturaleza del algoritmo o por cambios en el dataset. La siguiente

categoría, "*The Best Model*" corresponde con el modelo cuyas métricas son los mostradas en esta investigación.

Tras seleccionar la categoría se debe seleccionar el botón "*Report*" para mostrar el informe de clasificación del modelo, exponiendo el informe por pantalla.




| | precision | recall | f1-score | support |
|---|---|---|---|---|
| **0** | 0.747069 | 0.880553 | 0.808337 | 1013.000000 |
| **1** | 0.955661 | 0.896220 | 0.924987 | 2910.000000 |
| **accuracy** | 0.892174 | 0.892174 | 0.892174 | 0.892174 |
| **macro avg** | 0.851365 | 0.888386 | 0.866662 | 3923.000000 |
| **weighted avg** | 0.901798 | 0.892174 | 0.894865 | 3923.000000 |

**Figura B.7**: *Pantalla Neural Network, Informe de clasificación*

### B.4.3. Análisis de archivos

Cada algoritmo es capaz de analizar muestras de archivos que se vayan subiendo al servidor. Para ello la categoría "*File scanner*" situado a la derecha de la pantalla permite seleccionar múltiples archivos para clasificarlos según los resultados que muestre el modelo del algoritmo.

Se debe seleccionar el botón "*Elegir archivos*" y elegir el archivo a análizar. Una vez subido el archivo, seleccione el modelo con el que lo quiera analizar, "*New Model*" o "*The best model*" tras lo cual seleccione el botón "*Scanner*".

Tras analizar el archivo se mostrarán los resultados por pantalla.

Machine Learning Analyzer File Upload Neural Network Decision Trees Naive Bayes KNearest Neighbor Random Forest Log Out

Neural Network

Neural network zone, creation of a new neural network model, creation of a classification report and upload of files to identify if they are malware or benign.

Create a new model

New model

Classification report

New Model
The Best Model

Report

File scanner

**Message:** Los archivos son benignos

Malware :
Benigno : 2

Elegir archivos Ninguno archivo selec.

New Model
The Best Model

Scanner

Classification Report Result

**Figura B.8**: *Pantalla Neural Network, Análisis de archivo*

# Apéndice C

# Guía de Usuario, Ejecución en Local

## C.1. Introducción

Debido a las limitaciones de subscripción con el servidor Heroku, ciertas funcionalidades del servicio no están operativas para la versión web, pero si para la ejecución en local.

Las siguientes funcionalidades están inoperativas en la versión web:

- Enriquecimiento de la base de datos: Debido al exceso número de registros que componen el dataset, se ha excedido el límite permitido de la base de datos de 10.000 registros a 84.000.
- Creación de los modelos: debido a que la creación de un nuevo modelo estadístico requiere el uso de una cantidad elevada de recursos computacionales, esta función de los algoritmos no se debe realizar en la versión online del servicio, ya que produce una caída temporal del mismo.

## C.2. Requisitos e Instalación

Se debe descargar el código de la aplicación, almacenado en github, a través del siguiente enlace, https://github.com/ssanzgar/machinelearninganalyzer.git. Además de ello es recomendable tener acceso a los modelos y dataset(excel.csv) almacenados en Dropbox ya que es aquí donde se irán almacenando cuando se creen nuevos modelo y se descargue el datasets.

Para poder ejecutar el proyecto en local se debe realizar los siguientes pasos:

- Se debe instalar la herramienta python(https://www.python.org/downloads/) con una versión "3.8.8.º superior, para poder ejecutar el proyecto en local.

- Se debe instalar las bibliotecas requeridas por el proyecto, para ello abra una ventana Dos, como administrador, con el comando cmd y acceda a la carpeta del código con los comandos cd.

```
(machinelearninganalyzer) C:\Repo\machinelearninganalyzer\src>dir
 El volumen de la unidad C no tiene etiqueta.
 El número de serie del volumen es: A083-B197

 Directorio de C:\Repo\machinelearninganalyzer\src

24/09/2022  19:28    <DIR>          .
24/09/2022  19:28    <DIR>          ..
14/08/2021  00:42    <DIR>          app
14/08/2021  00:42               108 entrypoint.py
14/08/2021  00:42                28 Procfile
24/09/2022  19:28             1.188 requirements.txt
14/08/2021  00:42                12 runtime.txt
14/08/2021  00:42    <DIR>          __pycache__
               4 archivos          1.336 bytes
               4 dirs  57.339.064.320 bytes libres

(machinelearninganalyzer) C:\Repo\machinelearninganalyzer\src>
```

**Figura C.1**: *Guía de usuario, directorio de contenidos*

- Una vez accedido a la carpeta del código entre dentro de la carpeta src y escriba el comando pip install -r requirements.txt

```
(machinelearninganalyzer) C:\Repo\machinelearninganalyzer\src>dir
 El volumen de la unidad C no tiene etiqueta.
 El número de serie del volumen es: A083-B197

 Directorio de C:\Repo\machinelearninganalyzer\src

24/09/2022  19:28    <DIR>          .
24/09/2022  19:28    <DIR>          ..
14/08/2021  00:42    <DIR>          app
14/08/2021  00:42               108 entrypoint.py
14/08/2021  00:42                28 Procfile
24/09/2022  19:28             1.188 requirements.txt
14/08/2021  00:42                12 runtime.txt
14/08/2021  00:42    <DIR>          __pycache__
               4 archivos          1.336 bytes
               4 dirs  57.339.064.320 bytes libres

(machinelearninganalyzer) C:\Repo\machinelearninganalyzer\src>pip install -r requirements.txt
WARNING: Ignoring invalid distribution -rllib3 (c:\programdata\anaconda3\envs\machinelearninganalyzer\lib\site-packages)

WARNING: Ignoring invalid distribution -rllib3 (c:\programdata\anaconda3\envs\machinelearninganalyzer\lib\site-packages)

Requirement already satisfied: absl-py==0.13.0 in c:\programdata\anaconda3\envs\machinelearninganalyzer\lib\site-package
s (from -r requirements.txt (line 1)) (0.13.0)
Requirement already satisfied: astunparse==1.6.3 in c:\programdata\anaconda3\envs\machinelearninganalyzer\lib\site-packa
ges (from -r requirements.txt (line 2)) (1.6.3)
Requirement already satisfied: cachetools==4.2.2 in c:\programdata\anaconda3\envs\machinelearninganalyzer\lib\site-packa
ges (from -r requirements.txt (line 3)) (4.2.2)
Collecting certifi==2021.5.30
```

**Figura C.2**: *Guía de usuario, instalación de los requisitos*

- Una vez finalizada la instalación de los componente, ejecute el comando python entrypoint.py
- Una vez ejecutado aparecerá los siguientes datos. Seleccione la dirección del hostlocal e introdúzcalo en una herramienta de navegación que tenga instalada.

```
(machinelearninganalyzer) C:\Repo\machinelearninganalyzer\src>python entrypoint.py
 * Serving Flask app 'app' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 537-283-885
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

**Figura C.3**: *Guía de usuario, Ejecución del archivo entrypoint.py*

- Con esto la aplicación queda operativa.

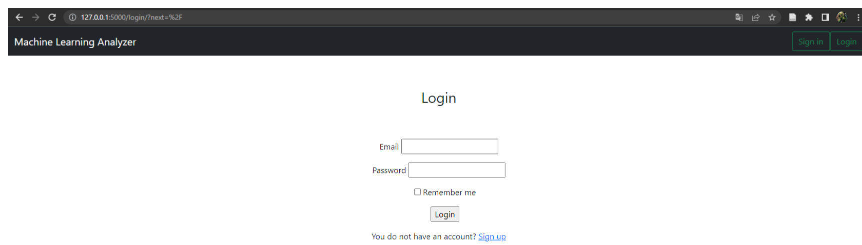



**Figura C.4**: *Guía de usuario, Pantalla login en local*