Detección Automática de Sitios Web Fraudulentos
Automatic Detection of Fraudulent Websites

TRABAJO FIN DE GRADO
GRADO EN INGENIERÍA INFORMÁTICA
CURSO 2019–2020

Héctor Hugo Coronado Huamán
Adina Han
Laura Sanz García

Directores

Luis Javier García Villalba
Esteban Alejandro Armas Vega

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Madrid, Junio de 2020

Agradecimientos

A nuestros directores de proyecto, Luis Javier y Esteban que desde el primer momento, mostraron una gran confianza en lo que podríamos conseguir con este trabajo de fin de grado, y supieron transmitirnos esa ilusión.

A Ana Sandoval por su notable ayuda y constante apoyo.

A diversos profesores a lo largo de estos años de carrera por habernos ayudado a adquirir los conocimientos necesarios que hemos aplicado para desarrollar este trabajo con éxito.

A la Facultad de Informática de la Universidad Complutense de Madrid por los recursos ofrecidos.

Este Trabajo de Fin de Grado ha sido realizado dentro del Grupo de Análisis, Seguridad y Sistemas (Grupo GASS, `https://gass.ucm.es/`, Grupo 910623 del catálogo de grupos reconocidos por la UCM) como parte de las actividades del proyecto de investigación THEIA (Techniques for Integrity and Authentication of Multimedia Files of Mobile Devices) con referencia FEI-EU-19-04.

Índice General

Índice de Figuras VII

Índice de Tablas IX

Lista de Acrónimos XI

Abstract XV

Resumen XVII

1. Introducción 1
    - 1.1. Introducción y Contexto Actual . . . 1
    - 1.2. Contexto . . . 5
    - 1.3. Plan de Trabajo . . . 5
    - 1.4. Estructura del Trabajo . . . 6

2. Phishing 9
    - 2.1. Historia del Phishing . . . 9
    - 2.2. ¿Qué es la Ingeniería Social? . . . 10
    - 2.3. Ingeniería Social Aplicada al Phishing . . . 11
    - 2.4. Impacto Económico y Social del Phishing . . . 11
    - 2.5. Ataques Empleados en la Actualidad . . . 13

3. Técnicas de Detección de Páginas Fraudulentas 17
    - 3.1. Sistemas de Detección Basados en Listas . . . 18
    - 3.2. Sistemas de Detección Basados en Algoritmos de Aprendizaje Automático . 19

4. Sistema Propuesto 25
    - 4.1. Descripción del Sistema Propuesto . . . 25
    - 4.2. Obtención de los Datos . . . 28
    - 4.3. Características Desarrolladas . . . 29
        - 4.3.1. Características de la Dirección de la Página Web . . . 29
        - 4.3.2. Características del Modelo de Objeto de Documento . . . 32
    - 4.4. Extracción de las Características . . . 37
    - 4.5. Configuración y Análisis de los Algoritmos Utilizados . . . 39
        - 4.5.1. Configuración de los Hiperparámetros . . . 40

4.6. Tecnologías Utilizadas 40
4.6.1. Desarrollo del Aprendizaje Automático 41
4.6.2. Desarrollo de la API y de la Aplicación Web 41

5. Experimentos y Resultados 43
5.1. Métricas Utilizadas 43
5.2. Experimentos 44
5.2.1. Pruebas con Diferentes Conjuntos de Características 56
5.2.2. Validation Curve 59
5.3. Evaluación de las Mejores Características 61
5.4. Resultados Obtenidos 62

6. Contribuciones Individuales 65
6.1. Héctor Hugo Coronado Huamán 66
6.2. Adina Han 68
6.3. Laura Sanz García 70

7. Conclusiones y Trabajo Futuro 73
7.1. Conclusiones 73
7.2. Trabajo Futuro 74

8. Introduction 77
8.1. Introduction and Current Context 77
8.2. Context 80
8.3. Workplan 80

9. Conclusion 83
9.1. Conclusions 83
9.2. Future Work 84

Bibliografía 85

Índice de Figuras

1.1. Sitios web con contenido malicioso [Bro20] 2
1.2. Gráfica de los sectores más atacados por phishing [APWG20] 2
1.3. Captura de pantalla de correo electrónico malicioso [INC20] 3
1.4. Número de sitios web con contenido malicioso de los últimos 3 años [Bro20] 3
1.5. Sitios web de contenido malicioso [APWG20] 4
1.6. Diagrama de Gantt 6

2.1. Ataques de phishing [KPRK19] 13

3.1. Sistemas de detección de phishing 17

4.1. Diagrama de flujo del sistema 26
4.2. Esquema del proceso de Machine Learning 27
4.3. Captura de pantalla de la aplicación 28
4.4. Componentes de una URL 30

5.1. Curva de Validación para el parámetro “n_estimators” 60
5.2. Curva de Validación para el parámetro “max_depth” 60
5.3. Curva de Validación para el parámetro “min_samples_split” 61
5.4. 20 mejores características por orden de importancia 62

8.1. Google Safe Browsing Statistics Graph [Bro20] 77
8.2. Chart of the most attacked sectors [APWG20] 78
8.3. Malicious Email Screenshot [INC20] 78
8.4. Malicious content websites [Bro20] 79
8.5. Malicious content websites [APWG20] 80

Índice de Tablas

3.1. Comparación de los trabajos estudiados . . . 22
3.2. Comparación de las ventajas y desventajas de los trabajos estudiados . . . 23

4.1. Conjunto de datos utilizados . . . 29
4.2. Características de la URL y sus funciones asociadas . . . 30
4.3. Relación de características de la URL y trabajos asociados . . . 32
4.4. Características del DOM y sus funciones asociadas . . . 36
4.5. Relación de características del DOM y trabajos asociados . . . 37

5.1. KNN con validación cruzada . . . 45
5.2. KNN con train_test_split . . . 46
5.3. BernoulliNB con validación cruzada . . . 47
5.4. BernoulliNB con train_test_split . . . 47
5.5. Árboles de decisión con validación cruzada . . . 48
5.6. Árboles de decisión con train_test_split . . . 48
5.7. Bosques aleatorios con validación cruzada . . . 49
5.8. Bosques aleatorios con train_test_split . . . 49
5.9. AdaBoost con validación cruzada . . . 50
5.10. AdaBoost con train_test_split . . . 50
5.11. MLP con validación cruzada . . . 51
5.12. MLP con train_test_split . . . 51
5.13. GradientBoosting con validación cruzada . . . 51
5.14. GradientBoosting con train_test_split . . . 52
5.15. LinearSVC con validación cruzada . . . 53
5.16. LinearSVC con train_test_split . . . 53
5.17. SGD con validación cruzada . . . 54
5.18. SGD con train_test_split . . . 54
5.19. GaussianNB con validación cruzada . . . 54
5.20. GaussianNB con train_test_split . . . 55
5.21. Análisis Discriminante Lineal con validación cruzada . . . 55
5.22. Análisis Discriminante Lineal con train_test_split . . . 55
5.23. Redes Neuronales con validación cruzada . . . 56
5.24. Redes Neuronales con train_test_split . . . 56

5.25. Resultados de los algoritmos con características de la URL con validación cruzada . . . 57
5.26. Resultados de los algoritmos con características de la URL con train_test_split 57
5.27. Resultados de los algoritmos con características del DOM con validación cruzada . . . 58
5.28. Resultados de los algoritmos con características de la DOM con train_test_split 58
5.29. Mejores resultados obtenidos con train_text_split . . . 62
5.30. Mejores resultados obtenidos con validación cruzada . . . 63
5.31. Comparación de resultados con otros trabajos . . . 64

Lista de Acrónimos

AOL	American Online
API	Application Programming Interface
APWG	Anti-Phishing Working Group
BN	Bayesian Network
COVID19	Coronavirus Desease
CSS	Cascade Style Sheets
DNS	Domain Name Service
DOM	Document Object Model
EEUU	Estados Unidos
FP	False Positives
GB	GigaByte
GBDT	Gradient Boosting Decision Tree
HS	Harmony Search
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol

HTTPS	Hypertext Transfer Protocol Secure
IA	Inteligencia Artificial
IDE	Integrated Development Environment
IP	Internet Protocol
KNN	K Nearest Neighbors
LR	Logistic Regression
MLP	Multilayer Perceptron
NLP	Natural Language Processing
PHaaS	Phishing as a Service
REST	Representational State Transfer
RF	Random Forest
SaaS	Software as a Service
SGD	Stochastic Gradient Descent
SMO	Sequential Minimal Optimization
SMS	Short Message Service
SQL	Structured Query Language
SSL	Secure Sockets Layer
SVC	Support Vector Classification
SVM	Support Vector Machines

TF-IDF	Term Frecuency - Inverse Document Frecuency
TLD	Top Level Domain
TLS	Transport Layer Security
TP	True Positives
URL	Uniform Resource Locator
vCPU	virtual Central Processing Unit
XML	Extensible Markup Language

Abstract

Over the last few years, there has been a considerable increase in communications and operations carried out through the Internet. Social networks or electronic commerce are an example of the type of management that can be carried out online. This increase is reflected in the fact that fraud attacks are more frequent every year. These attacks use social engineering strategies to steal sensitive information from the users pretending to be a trusted company or person. During the pandemic declared by the COVID-19 outbreak due to the increase of telecommuting and online shopping, these type of attacks have increased by 5.38 % [APWG20] with a maximum of 59,525 fraudulent websites detected in a single day. That is why the development of tools that detect phishing attacks has never been more important than it is now. There are currently blacklist detection systems that are very powerful, but do not have the ability to detect phishing web pages in real time, something necessary when the average duration of a phishing web page is around 20 hours [MC07]. There are also detection systems based on machine learning algorithms, which extract features from phishing web pages and, through machine learning algorithms, develop a model that allows predicting whether a web page is malicious or not. This type of detection systems allow to detect phishing web pages in real time. We propose a detection system that combines both systems. First we check that the suspicious web page is not on our blacklist, which is localized in our database. If it is not found, we search it in the Google Safe Browsing database. If the answer is negative, we use a prediction model to categorize the page as phish or non-phish. The model has been selected after testing 12 different machine learning algorithms which have been provided with features extracted from the web page address and the document object model. Later, we compare the results of the model with a set of selected papers. The best result has been obtained using the **Random Forest** algorithm. We achieved a percentage of true positives of **90.6 %** a percentage of false positives of **2.35 %** and a percentage of accuracy of **95,50 %**.

Keywords: Blacklists, Classification algorithms, Covid-19, Cybersecurity, Identity fraud, Identity theft, Internet fraud, Machine Learning, Phishing, Phishing attacks.

Resumen

A lo largo de los últimos años se ha observado un aumento considerable en las comunicaciones y operaciones que se realizan diariamente a través de Internet. Las redes sociales o el comercio electrónico son un ejemplo del tipo de gestiones que se pueden llevar a cabo en la red. Este aumento ha supuesto que cada año sean más frecuentes los ataques de *phishing*. Estos ataques utilizan ingeniería social para robar información personal o confidencial al usuario, haciéndose pasar por una empresa o persona de confianza. Durante la pandemia declarada por el brote de Coronavirus Desease (COVID19), debido al aumento del teletrabajo y de las compras en línea, este tipo de ataques se ha incrementado en un 5.38 % [APWG20], con un máximo de 59,525 sitios web fraudulentos detectados en un sólo día. Por eso cada día es más importante el desarrollo de herramientas que permitan detectar estos ataques. Actualmente existen sistemas de detección basados en listas negras que son muy potentes, pero que no tienen la capacidad de detectar páginas web de *phishing* en tiempo real, algo necesario cuando la duración media de una página web de *phishing* es en torno a 20 horas [MC07]. También, existen sistemas de detección basados en algoritmos de aprendizaje automático, que extraen características de las páginas web de *phishing* y desarrollan un modelo que permite predecir si una página web es maliciosa o no. Este tipo de sistemas de detección permite identificar páginas web fraudulentas en tiempo real. Este trabajo propone un sistema de detección que combina ambos métodos. Primero se comprueba que la página web sospechosa no está en la lista negra localizada en una base de datos almacenada localmente. En caso de no ser encontrada se realiza una búsqueda en la base de datos de *Google Safe Browsing*. Si la respuesta es negativa se utiliza un modelo de predicción para categorizar la página como *phishing* o no *phishing*. El modelo ha sido seleccionado tras probar 12 algoritmos diferentes de aprendizaje automático a los cuales se les ha suministrado características extraídas de la dirección de la página web y del modelo de objeto de documento. Posteriormente se comparan los resultados del modelo con un conjunto de trabajos seleccionados. El mejor resultado se ha obtenido con el algoritmo de **Bosques aleatorios** o *Random Forest*. Se ha logrado un porcentaje de aciertos del **90.6 %**, un porcentaje de falsos positivos del **2.35 %** y una precisión de **95,50 %**.

Palabras clave: Algoritmos de aprendizaje automático, Algoritmos de clasificación, Ataques de phishing, Ciberseguridad, Covid-19, Fraude por Internet, Listas negras, Páginas web fraudulentas, Robo de identidad, Suplantación de identidad.

Capítulo 1

Introducción

1.1. Introducción y Contexto Actual

El mundo actual está completamente digitalizado. Se estima que a finales del año 2019 había un total de 4,574,150,134 [NO20] usuarios conectados. La mayoría de las personas se comunican entre ellas utilizando Internet a través de un dispositivo móvil o un ordenador. Además, el número de personas que realizan movimientos bancarios en línea o utilizan el comercio electrónico, ha aumentado considerablemente debido a la comodidad y a la asistencia que ofrecen muchos de estos servicios.

Desafortunadamente, este incremento de la actividad también se ha visto reflejado en un aumento del número de ataques, en concreto, los ataques de *phishing*. En este tipo de ataques, se utiliza ingeniería social para intentar robar información personal o confidencial al usuario, haciéndose pasar por una empresa o persona de confianza. Algunos ejemplos de información que intentan robar suele ser: información sensible de empresas, contraseñas, datos de cuentas bancarias o datos de tarjetas de crédito. La toma de contacto con el usuario suele tener lugar mediante correos electrónicos, sistemas de mensajería instantánea, redes sociales o incluso llamadas telefónicas.

Para robar la información, la técnica más utilizada consiste en derivar al usuario a una página web fraudulenta, muy similar a la página web de confianza para que el usuario introduzca información sensible pensando que lo está haciendo en el sitio web de confianza. En este trabajo se utiliza el término página web de *phishing* para hacer referencia a una página web fraudulenta o maliciosa.

La Figura 1.1 muestra la gráfica de sitios web con contenido malicioso detectados por *Google Safe Browsing* en enero de los últimos años. La línea de color rojo representa el número de páginas web fraudulentas detectadas.

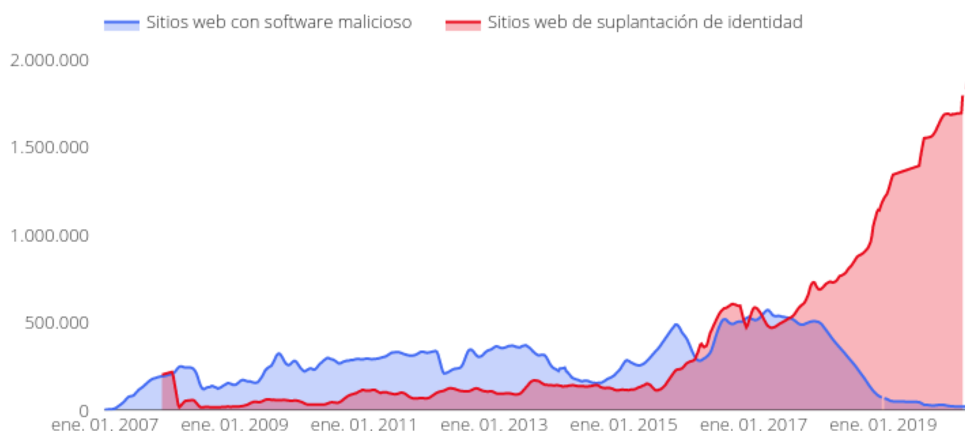


Figura 1.1: Sitios web con contenido malicioso [Bro20]

Desde principios del año 2020, la pandemia declarada por el brote de COVID19 ha tenido confinadas en sus casas a millones de personas en todo el mundo, con el propósito de controlar su propagación. Para poder continuar con la actividad económica en la medida de lo posible, muchas empresas y sus correspondientes trabajadores han tenido que adaptarse al teletrabajo, y el comercio electrónico ha aumentado considerablemente.

Según el artículo publicado el 11 de Mayo de 2020 por el grupo de investigación Anti-Phishing Working Group (APWG), [APWG20] en el primer periodo de 2020, el número de reportes de *phishing* que tienen ha subido un 5,38 % y el sector con más ataques es el que engloba Software as a Service (SaaS) y los correos electrónicos, con un 33.5 % del número total de ataques de *phishing*.

La Figura 1.2 muestra los sectores que más ataques de *phishing* han recibido en el primer trimestre de 2020.

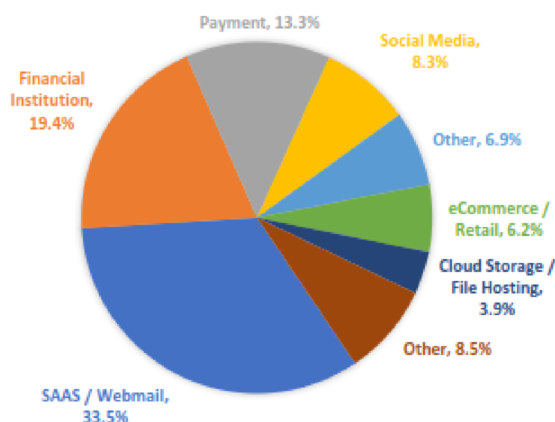
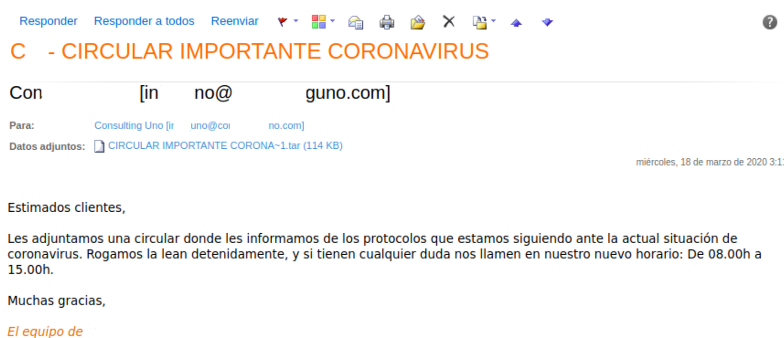


Figura 1.2: Gráfica de los sectores más atacados por phishing [APWG20]

La Figura 1.3 muestra un ejemplo real de un correo electrónico malicioso que estuvo circulando en el mes de marzo de 2020.


Responder Responder a todos Reenviar

C - CIRCULAR IMPORTANTE CORONAVIRUS

Con [in no@ guno.com]

Para: Consulting Uno [ir uno@coı no.com]

Datos adjuntos: CIRCULAR IMPORTANTE CORONA~1.tar (114 KB)

miércoles, 18 de marzo de 2020 3:11

Estimados clientes,

Les adjuntamos una circular donde les informamos de los protocolos que estamos siguiendo ante la actual situación de coronavirus. Rogamos la lean detenidamente, y si tienen cualquier duda nos llamen en nuestro nuevo horario: De 08.00h a 15.00h.

Muchas gracias,

El equipo de

Figura 1.3: Captura de pantalla de correo electrónico malicioso [INC20]

La Figura 1.4 muestra la gráfica de sitios web con contenido malicioso detectados en estos tres últimos años (Mayo 2018 a Mayo 2020). La línea de color rojo representa el número de páginas web fraudulentas detectadas. Se pueden observar dos grandes picos durante la pandemia de la COVID19, con un máximo de 59,525 sitios web fraudulentos identificados en un día, en concreto el 26 de Abril.

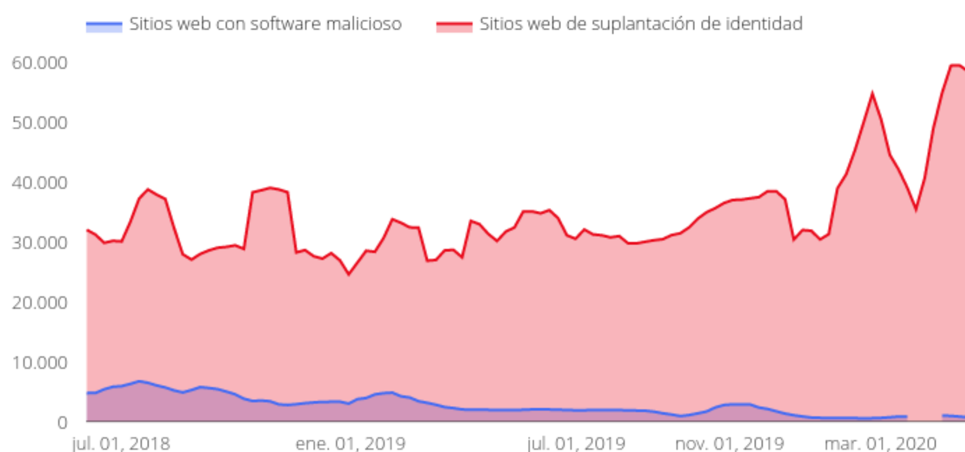


Figura 1.4: Número de sitios web con contenido malicioso de los últimos 3 años [Bro20]

Se observa constantemente que a medida que pasa el tiempo, los ataques de *phishing* se van perfeccionando. Actualmente, un 74 % de los ataques de *phishing* ya utilizan Hypertext Transfer Protocol Secure (HTTPS) [APWG20].

HTTPS es un protocolo de aplicación basado en Hypertext Transfer Protocol (HTTP) que utiliza un cifrado Secure Sockets Layer (SSL)/Transport Layer Security (TLS). En el informe de amenazas de *Webroot* [APWG20] se hace hincapié en que aunque la S signifique "seguro" realmente está relacionado con la privacidad. Cuando se visualiza el icono de un candado verde en el navegador, sólo significa que la información que se va a transmitir a ese sitio está cifrada y se entrega de forma segura pero no garantiza que el destino sea fiable.

Como se puede observar en la Figura 1.5, el porcentaje de páginas web de *phishing* que utilizan HTTPS llega hasta el 74 % actualmente. Por eso es muy importante desarrollar un sistema de detección de *phishing* que no sólo considere el protocolo utilizado, sino que

también tenga en cuenta todas las características que presentan las páginas web de *phishing* que hacen que se puedan distinguir de las legítimas. Además, muchas de las soluciones actuales utilizan listas negras para categorizar páginas web, debido a que presentan un bajo porcentaje de falsos positivos. Un estudio del 2009 demostró que entre el 47 % y el 83 % del *phishing* tarda 12 horas en aparecer en las lista negras desde la primera vez que se detectó [SWW+18]. Es probable que esto haya mejorado con el tiempo, pero sigue siendo un problema actualmente. Por lo tanto sería un factor determinante el poder detectar las páginas web de *phishing* en tiempo real.

Figura 1.5: Sitios web de contenido malicioso [APWG20]

En este trabajo se propone un sistema de detección en tiempo real de páginas web de *phishing* basado en la extracción de características del Document Object Model (DOM) y de la Uniform Resource Locator (URL) de una página web y su posterior análisis utilizando algoritmos de aprendizaje automático. El objetivo final de este proyecto es comparar este sistema de detección con otros trabajos existentes e intentar en la medida de lo posible, igualar o superar sus porcentajes de True Positives (TP) o verdaderos positivos y reducir al máximo el porcentaje de False Positives (FP) o falsos positivos.

Un TP es un resultado en el que el modelo predice correctamente el dato positivo. Por el contrario, un FP, es un resultado en el que el modelo predice incorrectamente el dato negativo. Por lo tanto aplicado a este caso un TP tendría lugar cuando analizando una página de *phishing*, el modelo detectara que efectivamente lo es y un FP sería cuando analizando una página el modelo detectara que es *phishing* cuando no lo es. Cuanto mayor sea el porcentaje de TP y menor sea el porcentaje de FP, mejor es el modelo.

Para realizar los experimentos, se han utilizado varios conjuntos de datos. Un primer conjunto con unas 18,000 URLs, un segundo conjunto con 96,000 URLs y un conjunto final con unas 129,270 URLs. Estos conjuntos se han ido refinando a lo largo de los diferentes experimentos realizados antes de aplicar los algoritmos de aprendizaje automático para reducir el ruido y evitar errores.

Además, para completar el sistema de detección propuesto, se ha desarrollado una Application Programming Interface (API) Representational State Transfer (REST) con una aplicación web sencilla. Esta aplicación no sólo integra el modelo de aprendizaje automático, sino que además cuenta con dos bases de datos de listas negras, una de las listas negras es proporcionada por la API *Google Safe Browsing*, y la otra ha sido elaborada con algunas URLs de *phishing* encontradas.

1.2. Contexto

Este Trabajo Fin de Grado ha sido realizado dentro del Grupo de Análisis, Seguridad y Sistemas (Grupo GASS, https://gass.ucm.es/, Grupo 910,623 del catálogo de grupos reconocidos por la UCM) como parte de las actividades del proyecto de investigación THEIA (Techniques for Integrity and Authentication of Multimedia Files of Mobile Devices) con referencia FEI-EU-19-04.

1.3. Plan de Trabajo

En esta sección se explican las fases que han tenido lugar durante todo el proyecto.

1. **Investigación**: Esta primera fase se llevó a cabo en un periodo aproximado de tres meses y se realizó sobre la temática del trabajo: detección de amenazas. La investigación inicial se basó tanto en la detección de amenazas de *phishing* como de *ransomware*. Hasta entonces los trabajos sobre *phishing* eran menos que los trabajos sobre *ransomware*, y esta fue una de las causas principales por las cuales junto con los tutores se decidió profundizar en la detección de amenazas de *phishing*. Una vez hecha la investigación se identificaron los trabajos que se habían realizado hasta el momento y sus resultados y se avanzó a partir de este punto. Los numerosos artículos de revistas científicas, de congresos y algunos libros sobre la materia sirvieron de ayuda para ampliar los conocimientos. Gracias al buscador *Google Scholar*, especializado en documentos académicos, así como a la biblioteca de la Universidad Complutense de Madrid se obtuvo la información necesaria. Se buscaron principalmente publicaciones sobre la detección de ataques de *phishing* y las medidas y herramientas para evitar estos ataques. Muchos de los trabajos utilizados en la fase de investigación fueron encontrados gracias a las referencias de otras publicaciones ya estudiadas. Se estudió en profundidad el lenguaje de programación *Python* y la librería que sería usada durante el desarrollo llamada *Scikit Learn*.

2. **Desarrollo**: La segunda fase, la del desarrollo de la propuesta, se comenzó una vez que la fase de investigación estuvo suficientemente avanzada y se adquirieron todos los conocimientos necesarios. En este momento se inició la implementación del sistema propuesto: el desarrollo de un método que permita identificar si una página web es categorizada como *phishing* o *no phishing*. La investigación siguió en un segundo plano mientras se desarrolló y codificó la propuesta, solo buscando y consultando conceptos puntuales o en momentos que la implementación se vio dificultada. En esta fase se llevó a cabo la búsqueda y creación de los conjuntos de datos necesarios para entrenar y evaluar el sistema propuesto. También se creó la base de datos local y se buscó una API que proporcionara una lista negra lo más completa posible.

3. **Pruebas**: En esta tercera fase, se preseleccionaron 12 algoritmos para los cuales se realizaron todo tipo de pruebas con diferentes características y parámetros para obtener la mejor combinación para cada uno. Se desarrolló, codificó, probó y rediseñó el sistema en numerosas ocasiones hasta lograr el diseño definitivo. Una vez obtenidas las métricas de los diferentes algoritmos, se seleccionó el algoritmo que mejor resultado obtuvo. Este resultado se verificó utilizando curvas de validación.

4. **Resultados**: En la última fase se procedió al análisis de todos los resultados obtenidos durante las pruebas realizadas y a la extracción de conclusiones a partir de dichos resultados. Según los datos obtenidos, se volvieron a hacer pequeños cambios en el código de la fase anterior y a realizar nuevas pruebas que pudiesen incrementar los resultados.

La Figura 1.6 muestra el diagrama de Gantt correspondiente al flujo de trabajo realizado.

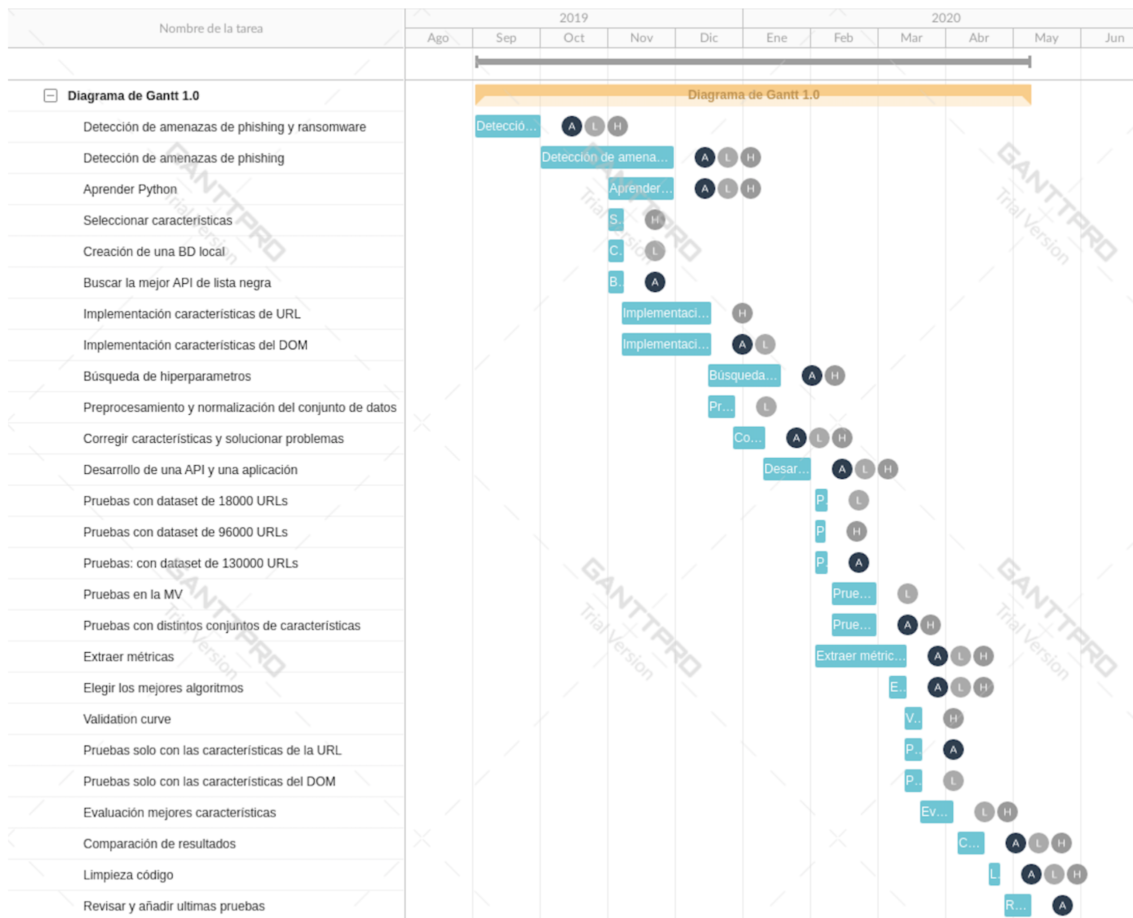


Figura 1.6: Diagrama de Gantt

1.4. Estructura del Trabajo

La parte restante de la memoria está organizada en 8 capítulos y tiene la siguiente estructura:

El Capítulo 2 introduce el concepto de *phishing*, ingeniería social, la relación entre ambos y los ataques de *phishing* que se emplean en la actualidad para comprender el panorama desde el cual se empieza.

El Capítulo 3 describe los distintos tipos de sistemas de detección de *phishing*: basados en listas blancas o negras y los basados en algoritmos de aprendizaje automático. Por último, se indican los trabajos que han sido utilizados como referencia y una breve descripción de cada uno de ellos.

El Capítulo 4 comienza describiendo el sistema propuesto indicando las principales características y los pasos que se han seguido. Primero se describe brevemente el sistema. A continuación, se incluye una pequeña explicación del funcionamiento y la forma utilizada para obtener el conjunto de datos. En este capítulo se explica detalladamente el proceso de extracción de características de la URL y del DOM. Para cada una de las 40 características, de las cuales 12 corresponden a la URL y 28 al DOM, se indica la funcionalidad. Asimismo, se presentan los diferentes problemas que han surgido y la solución encontrada. También se incluye una breve descripción de cada algoritmo utilizado tanto su configuración como su análisis y se finaliza con una sección que especifica las tecnologías utilizadas.

El Capítulo 5 inicia con la definición de las métricas utilizadas y posteriormente se exponen los resultados obtenidos para evaluar la efectividad de los algoritmos propuestos en el capítulo 4.

El Capítulo 6 presenta las contribuciones hechas por cada miembro del grupo junto con lo aprendido durante estos meses.

El Capítulo 7 expone las conclusiones alcanzadas tras el análisis de los resultados obtenidos en el Capítulo 5 y plantea las posibles líneas de investigación así como las ampliaciones que se pueden llevar a cabo partiendo de este trabajo.

Finalmente los Capítulos 8 y 9 contienen un resumen en inglés de la introducción y las conclusiones.

Capítulo 2

Phishing

El *phishing* es un ataque que utiliza ingeniería social para el robo de datos personales o confidenciales de un usuario mediante la suplantación de identidad de una empresa o persona de confianza. Del mismo modo, este tipo de ataque se emplea para sustraer información sensible a empresas, ya sea mediante la extracción de contraseñas, datos de cuentas bancarias o de tarjetas de crédito. A continuación se incluye una breve historia de cómo surgió el *phishing*, qué es la ingeniería social y la relación que existe entre ambos conceptos. Además se incluye una sección comentando el impacto económico y social del *phishing*.

Los ataques de *phishing* se dividen principalmente en tres fases. En la primera fase el atacante envía un mensaje suplantando la identidad de la empresa o persona de confianza. En la segunda fase se lleva a cabo la toma de contacto con la víctima, que puede ser mediante correos electrónicos, sistemas de mensajería instantánea o redes sociales. Estos medios pueden incluir un enlace a una dirección web fraudulenta o un *malware* adjunto. En la tercera fase el atacante aprovecha para extraer esta información y sacar beneficio de ella, ya sea usándola o vendiéndola a terceros.

2.1. Historia del Phishing

Siguiendo a Myers [JM06] (véase también [GAP18]), se puede establecer el año 1996 como la fecha de origen del *phishing*, ya que, en ese año, grupos de ciberdelincuentes utilizaron ese término para hacer referencia a una estrategia de ingeniería social dirigida a obtener la información de las cuentas que los usuarios de American Online (AOL), una empresa que daba acceso a Internet y ofrecía servicios por Internet, tenían en dicha empresa, para así poder acceder de manera gratuita a la red de AOL y a los servicios (por ejemplo, correo electrónico, noticias, software, juegos, grupos de discusión, contenidos culturales, recursos educativos) que tenían contratados esos usuarios e, incluso, poder contratar nuevos servicios cuyo costes se cargaban en las cuentas de sus legítimos dueños. Para obtener la información de acceso (por ejemplo, las contraseñas), los ciberdelincuentes se ponían en contacto con los usuarios a través de correo electrónico o el propio servicio de mensajería instantánea de AOL y les informaban, por ejemplo, que por motivos de seguridad necesitaban verificar la contraseña del usuario y necesitaban que el usuario se la proporcionase. Los correos electrónicos y los mensajes, además, estaban elaborados para parecer que procedían de un empleado de AOL y, dado que parecía haber un motivo fundado y que la apariencia de la petición también parecía genuina y procedía de una fuente autorizada, muchos usuarios dieron voluntariamente sus contraseñas a los ciberdelincuentes que, desde aquel momento, empezaron a conocerse como *phishers*.

Obviamente, no todos los usuarios proporcionaban sus contraseñas, por lo que, para que la estrategia funcionase los ciberdelincuentes debían mandar esos correos o mensajes fraudulentos a muchos usuarios, elaborar bien su apariencia —el "cebo"— y esperar que las potenciales víctimas "mordiesen el anzuelo", de ahí, que se denominara a esa estrategia *phishing*, término que proviene de la palabra inglesa *fishing* (pesca), dado que, además, entre los ciberdelincuentes de entonces era muy habitual utilizar las letras "ph" para sustituir a la letra "f", como así había ocurrido en una antigua forma de *hacking* telefónico conocido como *phreaking*, término que, a su vez, proviene de la unión de las palabras inglesas *phone* (teléfono) y *freak* (en su acepción de entusiasta, fanático). Esta estrategia se desarrolló cuando AOL extremó sus medidas de seguridad para evitar que los ciberdelincuentes, registrados con una identidad falsa y proporcionando un número de tarjeta de crédito falso generado automáticamente, accedieran a los servicios de AOL, ya que, inicialmente, esta creación falsa de identidades y tarjetas de créditos era suficiente para pasar las medidas de seguridad de AOL y activar una cuenta, y solo era detectada por AOL cuando la empresa intentaba, más tarde, cargar a la tarjeta fraudulenta el primer recibo por los servicios prestados. De hecho, el primer mensaje que se conoce que incluye la palabra *phishing* para referirse precisamente a la estrategia de ingeniería social descrita aludía precisamente a ese cambio de estrategia que tuvieron que hacer los ciberdelincuentes para acceder a los servicios de AOL. En concreto, ese mensaje, puesto por un ciberdelincuente en un grupo de noticias de *Usenet* denominado alt.2600 (*The Hacker Quarterly*), decía lo siguiente: "Antes podías crear una cuenta falsa en AOL siempre que tuvieras un generador de tarjetas de crédito. Sin embargo, AOL se ha vuelto más listo. Ahora verifican cada tarjeta con el banco después de introducirla. ¿Alguien conoce alguna forma de conseguir una cuenta que no sea phishing?" (mk590, "AOL for free?," alt.2600, 28 de enero de 1996; [JM06] p. 3). Aparentemente, el éxito que tuvieron estos ataques hizo que el *phishing* se extendiera de manera dramática a muchos otros ámbitos, incluyendo los delictivos, y que sus ataques se perfeccionaran. Desde entonces, por ejemplo, el *phishing* se extendió a cualquier usuario de Internet y no solo a los usuarios de AOL; trató de suplantar no solo a AOL y a sus empleados, sino a un número muy elevado de comercios en línea, instituciones bancarias, organismos oficiales, etc. Trató de obtener no solo información para acceder gratis a AOL, sino todo tipo de información como número de cuentas bancarias, número de tarjetas de créditos, números de documentos oficiales de identidad, etc., con el propósito de cometer robos, estafas, lavado de dinero, espionaje industrial y muchos otros comportamientos delictivos. Es más, el *phisher* ya no es el típico joven fanático de los ordenadores que intenta obtener acceso gratis a Internet, sino que son grupos organizados delictivos entre los cuales, incluso, puede haber grupos terroristas involucrados, y que, además, han generado un mercado del *phishing* en el que ilegalmente se comercia con datos personales [JM06].

2.2. ¿Qué es la Ingeniería Social?

En el campo de la seguridad de la información, la ingeniería social es un término que se usa para describir una intrusión de tipo no técnico que descansa en la manipulación de la gente para que divulgue información confidencial y realice acciones involuntarias que pueden ser dañinas para las propias personas [GAP18] véase también [OS11]. La ingeniería social es efectiva porque es una actividad de bajo riesgo que implica un ataque indirecto aprovechando las vulnerabilidades humanas como, por ejemplo, el miedo, la inexperiencia, el deseo de ayudar, el deseo de reconocimiento, la curiosidad o los múltiples sesgos y atajos cognitivos que emplean las personas para tomar decisiones. Estas tácticas permiten

al ingeniero social evitar que se despierten las sospechas sobre sus verdaderos objetivos. Los ataques más habituales basados en la ingeniería social son el *phishing*, el *malware*, el *hacking* y el *e-mail scam*.

2.3. Ingeniería Social Aplicada al Phishing

La ingeniería social se aplica en el *phishing* precisamente explotando las vulnerabilidades humanas que se comentaban en el epígrafe anterior. Por ejemplo, un correo electrónico de *phishing* puede ser enviado suplantando al banco de un usuario y hacerle creer que, si no realiza determinada acción, no podrá utilizar su tarjeta de crédito o sacar dinero, o puede ser enviado suplantando al servidor de correo o de almacenamiento de un usuario y hacerle creer que, si no realiza determinada acción, todos sus correos, fotos o vídeos se eliminarán. En este caso, la estrategia de *phishing* apela al miedo de las personas, al temor como emoción universal, puesto que para la mayoría de las personas esas son verdaderas amenazas, ya que, de ser ciertas, supondrían pérdidas irreparables para dichas personas. En otros casos, el *phishing* apela a los deseos humanos de ayuda, de manera que los correos solicitan ayuda al usuario para paliar una situación humanamente trágica (por ejemplo, un desastre natural, una enfermedad de una persona, etc.) y suplantan a instituciones benéficas y de ayuda. En otros casos, el *phishing* apela la curiosidad y, en el mensaje, se da a entender que, por ejemplo, en el adjunto se envía la foto de una persona famosa desnuda, el vídeo de una persona famosa que ha sido pillada en una situación comprometida o, simplemente, un paquete que ha sido enviado para él. También pueden apelar a la avaricia, y se proponen negocios que, aunque disparatados, son formas sencillas de ganar dinero, etc. En casi todos los casos, además, el *phishing* suele aprovechar la inexperiencia informática de los usuarios o su falta de conocimiento sobre las instituciones, empresas u organizaciones que supuestamente envían los correos. Finalmente, el *phishing* también se suele basar, desde el punto de vista de la ingeniería social, en los atajos y sesgos cognitivos que suelen utilizar las personas para tomar decisiones. Cialdini [Cia14] ha estudiado las múltiples estrategias persuasivas que se utilizan en *marketing* para aprovechar precisamente esos atajos y sesgos cognitivos para conseguir que las personas acepten una petición. Muchas de esas estrategias son también utilizadas por el *phishing* como, por ejemplo, apelar a que hay un tiempo limitado para realizar una acción, puesto que, si no, la oportunidad se pierde o la amenaza se cumple, o apelar al poder de persuasión que tiene el creer que los mensajes vienen de amigos o conocidos.

2.4. Impacto Económico y Social del Phishing

Varios estudios han tratado de estimar el impacto económico y social del *phishing*, obteniéndose resultados muy dispares pero que, en cualquier caso, implican grandes costes económicos y sociales para la sociedad. Algunos estudios han estimado el impacto económico y social del *phishing* mediante encuestas realizadas a la población general o a la población que utiliza Internet y que preguntaban a las personas si habían sufrido ataques de *phishing* y qué cantidad de dinero habían perdido debido a dichos ataques. Uno de los estudios de este tipo más publicitados y citados fue el realizado en 2007 por Gartner, una empresa internacional de consultoría e investigación en tecnología de la información [Gar07] véase también [HF08]. En este estudio se entrevistó a una muestra de 4,500 adultos de Estados Unidos (EEUU) que utilizaba Internet (y que era representativa de la población adulta de EEUU con Internet) y se encontró que, entre las personas que habían recibido correos electrónicos de *phishing* en 2017, el 3.3 % informaron de haber

perdido dinero con dicho ataque, lo que representaba un 2.8 % de la muestra total. La cantidad media de dinero perdido por esas personas ascendía a 886$. A partir de estos datos y del número de personas que componen la población de adultos con Internet en EEUU (aproximadamente 165 millones en 2007), el estudio de Gartner estimaba que, en 2007, 3.6 millones de estadounidenses habían perdido dinero debido a ataques de *phishing*, lo que implicaba que en los EEUU se había perdido, en total, 3,200 millones de dólares solo en 2017, cifras que, como se comentaba con anterioridad, han sido muy citadas y publicitadas [FC10]; [OS11]. Otros estudios, sin embargo, han encontrado cifras menores de afectados por *phishing* y también pérdidas económicas totales menores. Por ejemplo, la consultora Javelin Strategy y Research [Str]; véase también [HF08] realizó en 2005 una encuesta telefónica a 4000 adultos de EEUU, de los cuales 4.25 % informaron que habían sido víctimas de un robo de identidad (de cualquier tipo) en el último año. De esas víctimas, 1.7 % habían sido víctimas de *phishing*, las cuales habían perdido, como media, 2,320$. En el artículo de Javelin Strategy y Research [Str] se menciona explícitamente la cifra de 2,320$ como la media del dinero perdido por *phishing* por persona, pero, en el trabajo de Herly y Florencio [HF08], al mencionar el estudio de Javelin Strategy y Research [Str], se recoge, probablemente por error, la cifra de 2,820$ en lugar de 2,320$. Por tanto, en el estudio de Javelin [Str], el 0.07 % de los encuestados habían sido víctimas de *phishing*, lo que extrapolado a la población de EEUU, suponía que tan solo 158.100 estadounidenses habían sido víctimas de *phishing* en 2015 y que, en los EEUU, se habían perdido un total de 367 millones de dólares en ese año. De hecho, Herley y Florencio [HF08], tras revisar nueve de esos estudios, incluidos los realizados por Gartner no solo en 2007, sino también en 2005 y 2006, y el realizado por Javelin en 2005, encontraron que las estimaciones del número de personas afectadas por *phishing* y del impacto económico del *phishing* presentan una gran variabilidad debido, por ejemplo, a que, a veces, se incluyen como *phishing* otros tipos de delitos informáticos (por ejemplo, el robo de la identidad por parte de un familiar). Así, las cifras de usuarios de Internet que, anualmente, han sufrido *phishing* pueden oscilar entre tan solo 0.07 % y 2.18 %, y la cantidad media de dinero que han perdido por *phishing* puede variar entre tan solo 125$ y 2,320$. No obstante, aunque es cierto que esas cifras varían mucho, a partir de los datos de la Tabla 1 de Herley y Florencio [HF08], se puede estimar que, anualmente, en EEUU, se están perdiendo, por *phishing*, entre 367 millones de dólares. En [Str] se recoge la cifra que aparece explícitamente en dicho artículo, en lugar de calcularla a partir de los datos de la Tabla 1 por el error tipográfico que ya se comentaba respecto a la cantidad media de dinero perdido por persona) y 3.200 millones de dólares [Gar07], cifras que, en cualquier caso, son ambas enormes. A estos datos que estiman las pérdidas económicas sufridas por usuarios individuales de Internet, habría que sumar las pérdidas económicas que pueden sufrir las empresas debido a los ataques de *phishing*. Por ejemplo, un estudio del *Ponemon Institute* [Ins15] determinó que, en EEUU, para una empresa de tamaño medio (aproximadamente 10.000 empleados), los costes estimados del *phishing* eran de 3.77 millones de dólares por año. Estos costes se desglosaban en los costes incurridos por las empresas para contener el *malware* (208,174$ por año), los costes debidos al fracaso en contener el *malware* (338,098$ por año), los costes asociados a la pérdida de productividad de los trabajadores (1,819,923$ por año), los costes de contener los riesgos de las credenciales, principalmente de contener el robo de certificados y de claves criptográficas (381,920$ por año), y los costes debidos al fracaso en contener los riesgos de las credenciales (1,020,705$ por año). Como indican las cifras anteriores, la mayoría de los costes del *phishing* para una empresa tienen que ver con la pérdida de productividad de sus empleados, ya que se estimó que los empleados malgastan una media de 4.16 horas anualmente debido al *phishing*, con un rango de entre menos de una hora y más de 25

horas. Las pérdidas económicas que pueden sufrir las empresas debido al *phishing* podrían ser también más indirectas, en la medida que los ataques de *phishing* sobre una empresa podrían disminuir la confianza en la empresa, en especial, en sus servicios en línea, y, por tanto, muchos de sus clientes podrían, por ejemplo, dejar de utilizar las páginas web legítimas de dichas empresas por el miedo a ser víctimas de un ataque de *phishing* [OS11].

2.5. Ataques Empleados en la Actualidad

Con el paso de los años, los ataques de *phishing* han ido evolucionando y se han adaptado a las nuevas plataformas y tecnologías que se utilizan actualmente. Además los ataques de *phishing* se conciben como ataques de bajo riesgo que ofrecen grandes ganancias con poco esfuerzo. Los atacantes pueden hacer páginas web de *phishing* fácilmente reutilizando código de otras páginas o contratando un servicio Phishing as a Service (PHaaS) que les proporciona directamente la campaña de *phishing* ya desarrollada. El servicio PHaaS también es ofrecido por algunas empresas para realizar campañas de *phishing* dentro de otras empresas y evaluar el grado de concienciación de los trabajadores. A continuación se mencionan algunas de las técnicas más utilizadas por los ciberdelincuentes en los ataques de *phishing*.

La Figura 2.1 muestra los tipos de ataques de *phishing* más empleados en la actualidad.

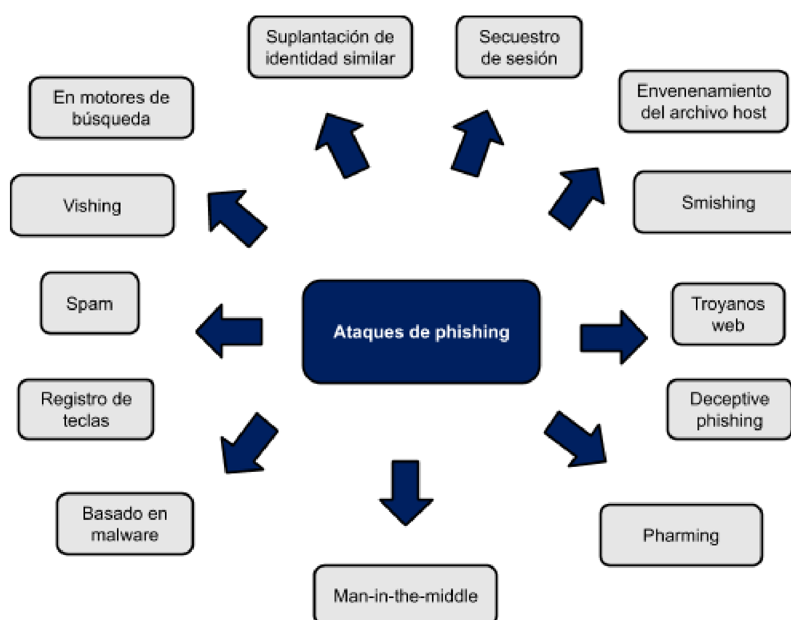


Figura 2.1: Ataques de phishing [KPRK19]

- En el ataque *similar spoofing* o suplantación de identidad similar, los atacantes se toman su tiempo en desarrollar páginas web casi idénticas a las páginas web legítimas, que poseen una URL similar a la del sitio legítimo. Estas dos características hacen que sea muy difícil para los usuarios poder identificar este tipo de ataque, y finalmente acaben introduciendo sus datos.

- En los ataques de *spam* o correo basura, los atacantes envían mensajes de correo electrónico a un gran número de usuarios solicitándoles información que requiera de su intervención, por ejemplo, que cambien su contraseña a través del enlace incluido

en el correo debido a una suplantación de identidad en su cuenta. En muchos casos el usuario, preocupado por el mensaje recibido, introduce sus datos confidenciales sin pensarlo detenidamente.

- Los ataques *deceptive phishing* son similares a los ataques de *spam* pero en este caso los atacantes envían un correo electrónico a la víctima pidiendo que verifique su cuenta. Cuando introduce sus datos, la página web se muestra otra vez como si los datos no se hubiesen introducido. En estos casos el usuario suele creer que es debido a algún fallo en el sistema y vuelve a introducir sus datos.

- El ataque *malware* o de software malicioso consiste en enviar al usuario un enlace o un archivo adjunto que contiene código malicioso y que se instala y ejecuta dentro del dispositivo de la víctima. Este tipo de ataque es muy común en pequeñas y medianas empresas ya que estas no siempre aplican las actualizaciones requeridas por las aplicaciones o el sistema operativo de sus dispositivos y por tanto el código malicioso se ejecuta sin problemas.

- Los troyanos web o *web trojans* son un tipo de ataque *malware* que camuflan código malicioso en un *software* legítimo. El código malicioso se queda implantado en el dispositivo de la víctima recogiendo todo tipo de información relevante que posteriormente es enviada a los atacantes.

- Los ataques de envenenamiento del archivo host o *host file poisoning attack*, tienen como objetivo atacar sistemas operativos *Windows*. Para ello interceptan la URL que escribe el usuario antes de que la comunicación se establezca. En el sistema operativo *Windows* hay un archivo "hosts" que es analizado antes de resolver el Domain Name Service (DNS). Los atacantes envenenan este archivo incluyendo en él direcciones de páginas web de *phishing*. Cuando la víctima introduce la URL, es redirigido a la página web de *phishing*.

- Los ataques de registro de teclas o *keylogger* extraen información sensible del usuario leyendo la entrada del teclado del dispositivo objetivo. Cada tecla pulsada por la víctima, incluyendo contraseñas o información confidencial, es registrada, almacenada y posteriormente enviada al atacante quien utiliza esta información para acceder a las cuentas de la víctima.

- Otro ataque muy utilizado es el secuestro de sesión o *hijacking* donde los atacantes obtienen la *cookie* de sesión de la víctima y pueden acceder a las cuentas donde ésta haya iniciado sesión.

- En el ataque de intermediario, conocido como *man-in-the-middle*, los atacantes se colocan entre la víctima y el sitio web de confianza. Mientras el usuario está introduciendo la información o haciendo operaciones, no nota ningún cambio, pero por detrás los atacantes están guardando la información para posteriormente venderla.

- En los ataques de *phishing* en motores de búsqueda o *search engine phishing*, los atacantes desarrollan sitios web muy atractivos que incluyen mensajes de ofertas especiales, que aparecen en los motores de búsqueda. Cuando la víctima intenta adquirir un producto de esa oferta e introduce sus datos bancarios estos son recogidos por los atacantes.

- El ataque *pharming* es una combinación de los términos *phishing* y *farming* en el que el atacante manipula el tráfico de un sitio web para robar información confidencial. Mediante un *exploit*, ataca el proceso de conversión de una dirección de Internet a una dirección Internet Protocol (IP). Para ello puede utilizar el método *host file poisoning attack* visto anteriormente, o envenenar el servidor DNS.

- El ataque *smishing* procede de la unión de Short Message Service (SMS) y *phishing*. En este tipo de ataque se intenta robar información confidencial a través de SMS o aplicaciones de mensajería instantánea como por ejemplo *Whatsapp*. El atacante envía un mensaje con un enlace a una página web maliciosa al dispositivo del usuario. Cuando el usuario accede al enlace, un troyano se inserta en el dispositivo permitiendo que éste sea controlado por el atacante.

- El ataque *vishing*, que procede de la unión de los términos *voice* y *phishing*, consiste en el robo de información confidencial a través de una web o correo electrónico fraudulentos y el uso de llamada de voz. El ataque comienza con el robo de información a través de una web o correo malicioso, después el atacante llama a la víctima haciéndose pasar como personal de la entidad solicitando una clave recibida (a través de SMS o *token* digital) necesaria para autorizar la operación realizada por el atacante.

Como se ha mencionado al principio, todos los ataques de phishing tienen un gran componente de ingeniería social y se adaptan perfectamente conforme van evolucionando las tecnologías. Por todo esto es importante poder contar con un sistema de detección que tenga la misma capacidad de adaptación.

Capítulo 3

Técnicas de Detección de Páginas Fraudulentas

En la actualidad existen múltiples técnicas que tienen como objetivo evitar los ataques de *phishing*. Los mecanismos de protección *anti-phishing* pueden tener dos enfoques. El primer enfoque consiste en desarrollar campañas de concienciación dirigidas a los usuarios. Lamentablemente, la constante evolución de los ataques de *phishing*, va un paso por delante de las campañas de concienciación, y cuando los usuarios consiguen identificar ciertos patrones, aparecen nuevos patrones aun más difíciles de detectar. El segundo enfoque consiste en utilizar software especializado para detectar y filtrar los posibles ataques de *phishing* antes que el usuario pueda introducir sus datos.

Este trabajo se centra en el segundo enfoque, que no depende del usuario final, sino del desarrollo de un buen sistema de detección.

La Figura 3.1 muestra los sistemas de detección *anti-phishing* que existen en la actualidad *phishing*

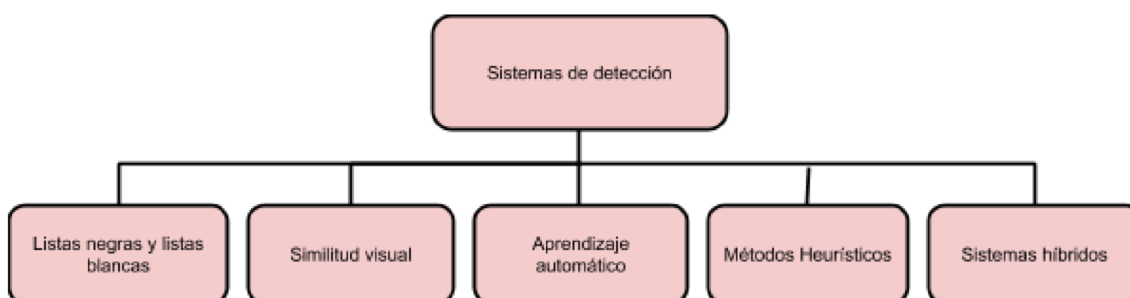


Figura 3.1: Sistemas de detección de phishing

Los sistemas para la detección de *phishing* más comunes encontrados actualmente [SBDD19] se pueden dividir en: sistemas de detección basados en listas (*list based*), basados en similitudes visuales (*visual similarity-based*) como podría ser el procesamiento de imágenes o *Image processing*, sistemas basados en algoritmos de aprendizaje automático (*machine learning*), sistemas basados en métodos heurísticos (*heuristics*) y sistemas de detección híbridos (*hybrid approaches*) que combinan varios métodos en un único sistema de detección.

El sistema de detección propuesto en este trabajo se encuentra en esta última categoría, puesto que combina listas negras y algoritmos de aprendizaje automático. En esta sección se incluyen algunos de los trabajos que se han analizado y que han servido para desarrollar

nuevas ideas que se han aplicado posteriormente a este trabajo.

En concreto se explican los métodos de detección basados en listas y los métodos de detección basados en algoritmos de aprendizaje automático. Muchas veces los sistemas de detección basados en similitudes visuales y los basados en métodos heurísticos utilizan algoritmos de aprendizaje automático, por lo que se incluyen algunos casos en esta sección.

3.1. Sistemas de Detección Basados en Listas

Los sistemas de detección basados en listas, utilizan dos tipos de listas para clasificar las páginas de *phishing*: listas blancas y listas negras.

- **Sistemas de detección basados en listas blancas**: las listas blancas o *whitelists* recopilan un conjunto de páginas web de confianza. Cada página web que no esté en la lista blanca es considerada como sospechosa.

  Jain y Gupta [JG16] proponen una solución basada en una lista blanca que se actualiza automáticamente con los sitios web que no se encuentra en la lista, envía un aviso al usuario para que no comparta información sensible. Además extraen los hipervínculos (*hyperlinks*) del código de la página web para combinarlos por otro lado con aprendizaje automático. Presenta un porcentaje de verdaderos positivos del 86.02 % y de falsos positivos del 1.48 %.

  El trabajo [AKF19] propone "*WhiteNet*", un sistema de detección basado en "listas blancas visuales" que analiza la similitud entre capturas de pantalla de páginas web utilizando redes neuronales convolucionales, mejorando la capacidad de detectar páginas web de *phishing* recién aparecidas (ataques *phishing* de *zero-day*). Este trabajo también forma parte de la categoría de sistemas de detección de similitud visual (*visual similarity*).

  En general, en todos los sistemas de detección basados en listas blancas el navegador deja acceder al usuario a una determinada página web sólo si esta se encuentra dentro de la lista blanca. Debido a este comportamiento, estos sistemas de detección presentan un gran porcentaje de FP.

- **Sistemas de detección basados en listas negras**: también llamadas *blacklists*, son listas de URLs fraudulentas conocidas. Proporcionan un método de control de acceso para evitar que el usuario visite estas páginas.

  Las listas negras más conocidas y utilizadas actualmente son *Google Safe Browsing*, *PhishTank* y *OpenPhish*. De acuerdo con [BK20], que analiza la efectividad de estas tres listas negras, la que mejor resultado obtiene es *Google Safe Browsing*. Además es con diferencia la que cuenta con más URLs, de media contiene 1.6 millones de URLs, mientras que *PhishTank* 12,433 y *OpenPhish* 5,861. El informe de transparencia que publica *Google Safe Browsing* no especifica exactamente el porcentaje de TP que obtiene [AKF19] pero aclara que el número de FP es mínimo. Por estas razones, se ha seleccionado *Google Safe Browsing* como la lista negra externa a la que consultar en la primera etapa del proceso de detección.

  Uno de los trabajos más actuales de este tipo de sistemas de detección, [RFF+20], presenta la extensión "3H1M" que compara la URL de la página web visitada con una lista negra de páginas web almacenada en una base de datos. Si el usuario visita una página web maliciosa, salta un *pop-up* en la pantalla con una alerta y se redirige al usuario a la página principal del buscador.

Si bien es cierto que las listas negras presentan un bajo porcentaje de FP, la tasa de detección suele ser baja, y no responden bien a ataques de *phishing* que acaban de ser creados y todavía no han sido categorizados (ataques *phishing* de *zero-day*). Además actualizar las listas negras en algunos casos conlleva un gran esfuerzo humano.

3.2. Sistemas de Detección Basados en Algoritmos de Aprendizaje Automático

Otra de las técnicas más populares para detectar páginas web fraudulentas es el uso de algoritmos de aprendizaje automático o *machine learning*. Esta técnica dinámica se basa en la implementación de este tipo de algoritmos y de métodos heurísticos, normalmente utilizados para extraer del conjunto de datos previamente recogidos, las características que diferencian a una página de *phishing* de otra que no lo es.

Los sistemas de aprendizaje automático se pueden clasificar acorde a la cantidad y al tipo de supervisión que reciben durante la fase de entrenamiento. Pueden ser supervisados, no supervisados o semi-supervisados. En el aprendizaje automático supervisado el conjunto de datos con el que se entrena al algoritmo incluye las soluciones deseadas, llamadas etiquetas o *labels*. Por lo tanto el algoritmo conoce a priori las entradas y salidas, y su tarea consiste en encontrar un patrón que las relacione para poder predecir las salidas de futuras entradas. Por otro lado, en el aprendizaje automático no supervisado los datos que se introducen no vienen con etiquetas predefinidas. El algoritmo entonces, describe la estructura de los datos y con ello intenta encontrar algún tipo de organización que simplifique su análisis, por ejemplo agrupando los datos. El aprendizaje semi-supervisado es una combinación de los dos anteriores.

La detección de una página web de *phishing* se trata de un problema de aprendizaje supervisado, en concreto un problema de clasificación. El algoritmo tendrá que generar un modelo dado un conjunto de datos que permita predecir si una página web es *phishing* o no. Para obtener el mejor resultado de esta técnica, es muy importante tener un buen conjunto de datos y en la medida de lo posible que el conjunto de datos sea grande. También influyen notablemente otros factores como el conjunto de características que se extraigan de estos datos y el tipo de clasificadores que se utilicen.

Estos sistemas de detección, tienen la ventaja de que permiten detectar páginas web fraudulentas de *zero-day*, es decir que acaban de aparecer y no han sido detectadas con anterioridad. Como desventaja se puede decir que en algunos casos, esta técnica tiene el riesgo de clasificar erróneamente las páginas web legítimas como páginas web de *phishing* (falsos positivos) ya que en ocasiones no hay garantía de que las características extraídas se encuentren en todas las páginas web de *phishing*. Este error se puede minimizar teniendo un conjunto de datos lo más actualizado posible y un conjunto de características preciso que sea revisado constantemente para adaptarlo a nuevas páginas web de *phishing* que vayan saliendo.

A continuación se mencionan los siete trabajos más interesantes que se han usado para sacar ideas de cómo implementar las características. Posteriormente estas características se utilizan con los algoritmos de aprendizaje automático y se comparan sus resultados con la solución propuesta.

CANTINA [ZHC07]. Este trabajo implementa un método de detección de páginas web de *phishing* basado en contenido aplicando un clasificador lineal simple de 8 características heurísticas. CANTINA y su posterior versión, CANTINA+, han sido de los trabajos más mencionados y comparados por otros estudios. En concreto, CANTINA se enfoca en desarrollar y evaluar una nueva heurística basada en Term Frecuency - Inverse Document

Frecuency (TF-IDF) y posteriormente aplicarla para la creación de hipervínculos robustos. TF-IDF que se traduce como frecuencia de término-frecuencia de término inversa, es un algoritmo que permite medir cómo de importante es una palabra en un conjunto de documentos. Dada una página web, lo que se hace es calcular el valor TF-IDF de cada palabra que aparece y posteriormente se generan hipervínculos robustos añadiendo a la URL de la página dada, los 5 términos con valor TF-IDF más alto. Después se busca la URL resultante en *Google* y si se encuentra en los primeros 30 resultados, se considera como una página web legítima. En caso contrario se categoriza como *phishing*. Para implementar este método de detección se utiliza un conjunto de datos de 100 URLs de *phishing* y 100 URLs de páginas web legítimas. Los resultados obtenidos con sólo TF-IDF fueron de 89 % de verdaderos positivos (TP) y en torno a un 1 % de falsos positivos (FP). Los resultados obtenidos combinando TF-IDF y las características heurísticas incrementan el número de verdaderos positivos (TP) que resulta en un 97 % pero también incrementa el porcentaje de falsos positivos (FP) que pasa a ser del 6 %.

CANTINA+ [XHRC11]. Este trabajo es la continuación de CANTINA. Entre las mejoras que propone, se encuentra el aumento del número de características heurísticas de 8 a 15 y el aumento en el conjunto de datos utilizados, que en este trabajo se divide en dos conjuntos de datos, uno de 8,118 URLs y otro de 14,883 URLs. Se extraen estas características del Hypertext Markup Language (HTML) del DOM de la página web, motores de búsqueda y servicios de terceros. Además se da mucha importancia al filtrado en la fase inicial de las páginas web dependiendo si tienen formulario o no. Esta nueva versión utiliza algoritmos de aprendizaje automático en vez de un clasificador lineal simple. Entre los algoritmos de aprendizaje automático que se han utilizado, se encuentran los algoritmos de máquina de vectores de soporte o Support Vector Machines (SVM), Regresión Lógica o Logistic Regression (LR), Redes Bayesianas o Bayesian Network (BN), Árboles de decisión J48 (*Decision Trees*), Bosques aleatorios o Random Forest (RF) y Adaboost. Además se hacen dos evaluaciones, una aleatorizada y otra basada en el tiempo. El mejor resultado se obtiene con el algoritmo de Redes Bayesianas y la evaluación aleatorizada con un 92 % de verdaderos positivos (TP) y un 0.4 % de falsos positivos (FP).

[HHF+11] combina características de CANTINA y PILFER (un sistema para la detección de *emails* fraudulentos) pero además se han desarrollado unas características del DOM que no aparecen en otros trabajos y que han servido de referencia, como *ID foreign anchors* o *ID page address* que se mencionan después. Cuenta con un total de 12 características que incluyen URL, DOM y características extraídas de motores de búsqueda. Además aparte de extraer el árbol del DOM, también renderiza el *JavaScript*. Al igual que CANTINA, utiliza el algoritmo TF-IDF para extraer lo que se denomina como "Conjunto de términos de identidad" que devuelve los 5 términos más importantes de la página. Se utiliza un conjunto de datos de 535 páginas web, con 200 páginas web legítimas y 325 de *phishing*. Sólo se aplica el algoritmo SVM y se obtiene un porcentaje del 97 % de TP y 4 % de FP.

En [SBDD19] se propone un sistema de detección que combina aprendizaje automático con procesamiento natural del lenguaje o Natural Language Processing (NLP) analizando exclusivamente la URL. Utilizan tres tipos de características que incluyen vectores de palabras, características que utilizan procesamiento natural del lenguaje y características híbridas. Se presenta un enfoque novedoso porque implementa una serie de módulos para detectar características en las palabras que aparecen en la URL de la página web sospechosa, como encontrar palabras generadas aleatoriamente o descomponer palabras que tengan más de 7 letras. Por último, realiza un análisis que se focaliza en detectar *typosquatting* (cuando un usuario accede a una página web diferente a la que deseaba

acceder debido a que ha introducido erróneamente la URL). Se utiliza uno de los conjuntos de datos más completos de los trabajos mencionados. Cuenta con 73,575 URLs de las cuales 36,400 son legítimas y 37,175 de *phishing*. Los algoritmos comparados han sido: Bayesiano Ingenuo o Naive Bayes, RF, K Nearest Neighbors (KNN) con n=3, K-star, Adaboost, Árboles de decisión y Optimización de la secuencia mínima o Sequential Minimal Optimization (SMO). El mejor resultado se ha obtenido con las características de NLP y el algoritmo de Bosques aleatorios, con una tasa de precisión de 97.98 %.

En [RP19] se propone un modelo de clasificación que, al igual que CANTINA+, extrae las características heurísticas de las URL, del DOM y de servicios de terceros. Cuenta con 16 características. De este estudio destaca que hay varias características enfocadas a detectar enlaces dentro de la página web que no funcionan o que no tienen ningún valor como sería por ejemplo `<a href="#"></a>`. En particular la característica *Broken links ratio* que examina cada enlace dentro de una página web para ver el porcentaje de aquellos que no responden, ha resultado bastante costosa de implementar, como se explica posteriormente. También tiene otra característica muy interesante que detecta si se ha sustituido el contenido de la página web por una imagen, sin necesidad de hacer una comparación con logos o con captura de pantalla. Esta estrategia de sustituir todo o parte del contenido suele ser utilizada por algunos atacantes. Cuentan con un conjunto de datos inicial de 3,526 URLs, de las cuales 2,119 son de *phishing* y el resto legítimas. Los algoritmos comparados han sido: Árboles de decisión J48, SMO, LR, Adaboost, RF, SVM, BN y Percepción Multicapa o Multilayer Perceptron Multilayer Perceptron (MLP). El mejor resultado se ha obtenido con el algoritmo de RF, con porcentaje de aciertos del 99.55 %.

Asimismo, en [LYC+19] se ha desarrollado un modelo de detección basado en apilamiento de algoritmos (*Stacking model*). Como otros trabajos ya mencionados, se extraen las características tanto de las URL como del DOM. Cuenta con 12 características de las cuales destaca HTML *string embedding* que está inspirada en un modelo llamado *Word2Vec* que utiliza redes neuronales y se basa en la incrustación de cadenas que extrae de los documentos. Se emplea un primer conjunto de datos de 49,947 páginas web y otro con 53,103. La diferencia entre ambos es que el segundo no sólo guarda la URLs y el código HTML sino también una captura de pantalla de la página web final renderizada. Esto hace de este trabajo, un ejemplo de mezcla entre aprendizaje automático y similitud visual. El modelo de apilamiento combina los algoritmos Gradient Boosting Decision Tree (GBDT), XGBoost y LightGBM. El mejor resultado se obtuvo con el conjunto de datos que incluye las capturas de pantalla con una tasa de precisión del 98,60 %.

Por último, [BAS19] propone una estrategia heurística de regresión no lineal para detectar las páginas web de *phishing*. Este trabajo, a diferencia del resto, no ha implementado el desarrollo de las características sino que ha utilizado un conjunto de 11,055 páginas web cada una con 30 características ya extraídas por otro estudio. De estas, se han seleccionado las 20 más eficientes que luego han sido probadas con el algoritmo de Búsqueda armónica o Harmony Search (HS) basado en una regresión no lineal y el algoritmo SVM. Este trabajo hace mucho hincapié en la selección de las características más eficientes. El mejor resultado se ha obtenido con el algoritmo de HS con una tasa de precisión del 92.80 %.

En las Tablas 3.1 y 3.2 se comparan todos los trabajos de aprendizaje automático mencionados anteriormente.

Tabla 3.1: Comparación de los trabajos estudiados

Trabajo	**Número de características**	**Independencia del lenguaje**	**Algoritmo ds Clasificación**	**Conjunto de datos**	**Tasa de predicción**
CANTINA [ZHC07]	8	No, TF-IDF implementado sólo para páginas en inglés	Clasificador lineal simple	200 URLs (100 legítimas y 100 de phishing)	TF-IDF + heurístics: TP = 89 % FP = 1 % TF-IDF: TP = 97 % FP = 6 %
CANTINA+ [XHRC11]	15	Si	Evaluación aleatorizada en Redes Bayesianas	-D1: 8,118 URLs -D2: 14,883 URLs	TP = 92 % FP = 0.4 %
[HHF+11]	12	No, TF-IDF implementado sólo para páginas en inglés	SVM	535 URLs (200 legítimas y 325 de phishing)	TPR = 97 % FPR = 4 %
[SBDD19]	3 tipos: NLP, WordVectors e híbridas	Fácilmente adaptable	Random Forest con características NLP	73,575 (36,400 legítimas y 37,175 de phishing)	Tasa de precisión del 97,98 %
[RP19]	16	Si	Random Forest	3,526 URLs (2,119 phishing y 1,407 legítimas)	Detection rate of 99,55 % false positive rate: 0.45
[LYC+19]	12	Si	Modelo de apilamiento con: GBDT, XGBoost y LightGBM	D1:49,947 URLs y HTML D2:53,103 URLs, HTML y capturas de pantalla de pág. renderizadas	Tasa de precisión del 97.30 %
[BAS19]	20	Si	Harmony Search basado en una regresión no lineal	11,055 URLs que vienen con las características ya extraídas	Tasa de precisión del 92,80 %

Tabla 3.2: Comparación de las ventajas y desventajas de los trabajos estudiados

Trabajo	**Ventajas**	**Desventajas**
CANTINA[ZHC07]	–Característica hipervínculos robustos usando el algoritmo TF-IDF mejora la eficiencia	-No se consigue una tasa de precisión alta manteniendo un número de falsos positivos aceptable. -Conjunto de datos muy limitado (200 URLs) -Problemas de latencia debido al uso de motores de búsqueda y servicios de terceros. -No renderiza JavaScript
CANTINA+[XHRC11]	-Comparan 6 algoritmos de clasificación diferentes -Evaluación aleatorizada y evaluación basada en tiempo. -Filtrado en fase inicial de páginas web con formulario -Renderiza JavaScript	-Problemas de latencia debido al uso de motores de búsqueda y servicios de terceros. -Aunque mejor que CANTINA, el conjunto de datos sigue siendo muy limitado
[HHF+11]	-Uso de algoritmo TF-IDF -Búsqueda de páginas web exclusivamente de login para añadirlas a su conjunto de datos	-Problemas de latencia debido al uso de motores de búsqueda y servicios de terceros. -No hace una comparación de diferentes algoritmos de clasificación -Conjunto de datos muy pequeño -Renderiza JavaScript
[SBDD19]	-Comparan 7 algoritmos de clasificación diferentes -Módulo de detección de palabras aleatorias -Características NLP que aumentan la eficacia. -Conjunto de datos de tamaño considerable -Independiente de servicios de terceros -Detección de typosquatting	-Un poco limitado, sólo analiza características de la URL y no del DOM.
[RP19]	-Compara 8 algoritmos de clasificación diferentes -Uso de algoritmo TF-IDF -Conjunto de características muy completo	-Característica broken links ratio bastante costosa -Conjunto de datos limitado -Depende aún más del tamaño del conjunto de datos -Problemas de latencia debido al uso de motores de búsqueda y servicios de terceros. -No renderiza JavaScript
[LYC+19]	-Modelo basado en el apilamiento de algoritmos que combina hasta 3 algoritmos. -Característica novedosa “HTML string embedding” -Similitud visual comparando las capturas de pantalla renderizadas -Independiente de servicios de terceros -Conjunto de datos de tamaño considerable	-Guardar las capturas de pantalla y el HTML de cada página web (a parte de la URL) hace que ocupe mucho espacio -Comparación de las capturas de pantalla ralentiza el sistema de detección
[BAS19]	-Selección de las características más eficientes con Árboles de decisión y wrapper -Cantidad de características considerable	-No implementa sus propias características

Capítulo 4

Sistema Propuesto

Después de analizar las fortalezas y debilidades de las técnicas de detección existentes en la literatura el sistema propuesto de aprendizaje automático debe tener al menos las siguientes características:

- Extracción de características heurísticas, dado que los trabajos estudiados que han obtenido más porcentaje de TP han desarrollado sus propias características heurísticas sacadas de la URL, del DOM o de motores de búsqueda
- Cantidad de características considerable (más de 12 características)
- Independencia del lenguaje
- Comparación de diferentes algoritmos de clasificación y análisis de los resultados
- Conjunto de datos de gran tamaño
- Detección de ataques de *phishing* de *zero-day*

4.1. Descripción del Sistema Propuesto

El objetivo del sistema propuesto es la detección de páginas web fraudulentas a través de la extracción de características usando los algoritmos de aprendizaje automático. El sistema cumple con la extracción de características de la URL y del DOM, siendo 40 el número total de estas. El conjunto de datos utilizado tiene un tamaño de 129,270 URLs y se aplica a 12 algoritmos de aprendizaje automático que permiten hacer una amplia comparación de los resultados obtenidos e incluso es capaz de detectar los ataques *phishing* de *zero-day*. Los resultados indican la tasa de aciertos que se consigue y esta se puede mejorar filtrando las características o configurando los parámetros de los algoritmos.

La Figura 4.1 representa el flujo del sistema de detección y sigue los pasos detallados a continuación:

1. El usuario introduce la URL que va a ser evaluada por el sistema.
2. El sistema consulta la base de datos local que almacena una lista negra con URLs de *phishing* recopiladas. Si está, se notifica al usuario que la URL introducida pertenece a una página web de *phishing*.
3. Si la URL no está en en esta lista negra, se hace una consulta a la lista negra que nos proporciona la API de *Google Safe Browsing*. En caso de localizar la URL se notifica al usuario que se trata de una página web de *phishing*.

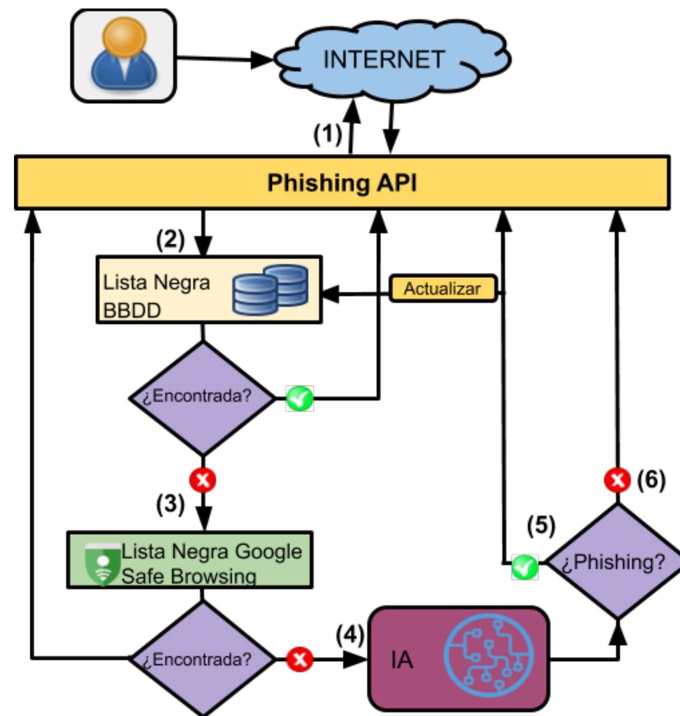


Figura 4.1: Diagrama de flujo del sistema

4. De no ser localizada en la lista negra de *Google Safe Browsing* se procede a evaluar la URL con el modelo de aprendizaje automático.

5. Si el modelo de aprendizaje automático que ha sido desarrollado detecta que se trata de una página web de *phishing* entonces se almacena la URL en la base de datos local y se notifica al usuario que la URL introducida pertenece a una página web de *phishing*.

6. Si el modelo indica que es una URL legítima se notifica al usuario y no se realiza ninguna acción sobre la base de datos.

La Figura 4.2 muestra el esquema de desarrollo del sistema de aprendizaje automático, esto es, la parte correspondiente a la Inteligencia Artificial (IA) del flujo anterior.

Aunque se explica más en detalle cada paso en los siguientes capítulos, se incluye a continuación una breve explicación de cómo funciona el proceso.

Primero, se reúne un conjunto de datos compuesto de dos columnas. La primera columna corresponde a la URL de la página web y la segunda a la etiqueta que indica si dicha URL es *phishing* o no, ya que es un sistema de aprendizaje automático supervisado. De ese conjunto de datos se extraen las características implementadas. Hay 12 características que extraen información de la URL, y 28 características que extraen información del contenido HTML del DOM de la página web. De estas últimas 28, la característica `broken_links_ratio` se ha excluido. Por lo tanto hay un total de 39 características. Una vez extraídas las características, se evalúan para ver cuáles son las mejores. Después, se dividen en dos conjuntos: de prueba y de entrenamiento. Una vez entrenado el modelo, el conjunto de prueba se utiliza para su evaluación final.

El conjunto de entrenamiento es el que se va a usar para afinar y entrenar el modelo. En este caso, del conjunto de características extraídas para cada URL, el 30 % será el

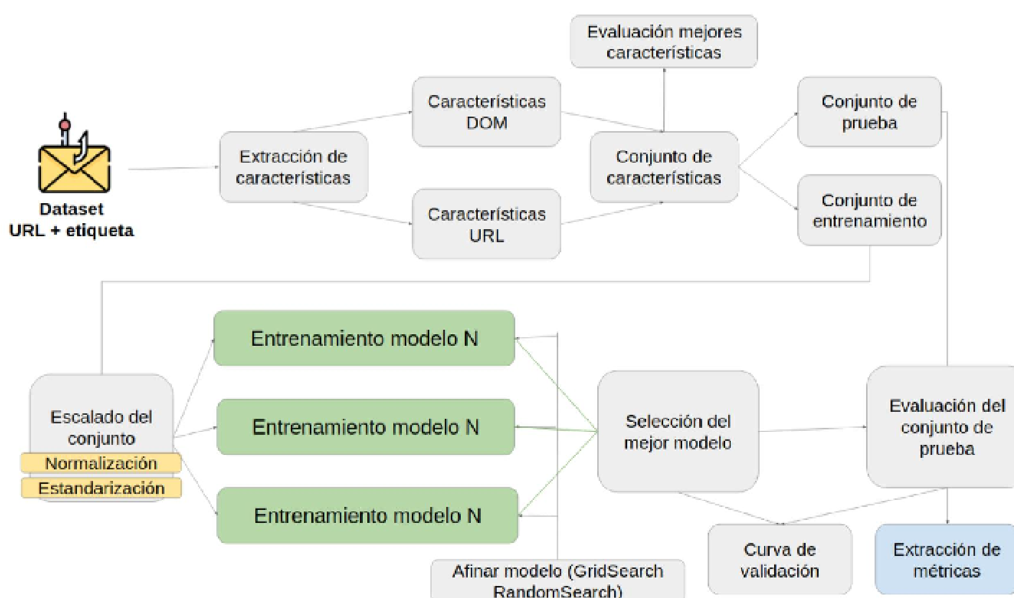


Figura 4.2: Esquema del proceso de Machine Learning

conjunto de prueba y el 70 % el conjunto de entrenamiento. Una vez seleccionado el conjunto de entrenamiento, se procede a aplicar una escala. Durante los experimentos, en la mayoría de los casos se aplica la normalización, aunque también en algún experimento se ha probado a aplicar la estandarización. Una vez aplicada la escala, se selecciona la lista de algoritmos de clasificación que se quiere probar y para cada uno, se aplica la estratificación del muestreo o *stratified sampling* y la validación cruzada o *cross-validation* con `k=10`. Para encontrar los hiperparámetros más adecuados para los algoritmos seleccionados, se utilizan los optimizadores de hiperparámetros que ofrece *Python*: *GridSearchCV* y *RandomizedSearchCV*, que dados unos rangos de prueba, encuentran la mejor combinación de parámetros. Una vez seleccionados los hiperparámetros, se entrenan los modelos y se selecciona el que dé mejor porcentaje de precisión, mayor número de TP y menor número de FP. Por último, se evalúa el modelo con el conjunto de prueba y se comparan ambos resultados utilizando la curva de validación o *validation curve*.

La API REST se ha implementado con *Python* utilizando *Flask*, *Connexion* y *SQLAlchemy*, componentes que serán descritos al final de este capítulo. Para la comunicación con la base de datos de *Google Safe Browsing*, utilizamos su API gratuita: *Safe Browsing Lookup API* que mediante una API *key* y una petición POST, devuelve si su lista negra contiene la URL o no. La base de datos local es una base de datos *SQLite* que extrae la información de un .csv al que se le adjuntará las URLs de *phishing* detectadas. Se ha implementado una API y una aplicación web sencilla para favorecer la interacción del usuario con la aplicación.

La Figura 4.3 muestra la apariencia de la aplicación web sencilla. Se accede a ella añadiendo "/app" al directorio raíz. Se puede acceder a través del siguiente enlace: `http://tfg-phishing.ml/app`

El acceso a la interfaz gráfica de la API se lleva a cabo añadiendo "/ui" al directorio raíz. Se puede acceder a través del siguiente enlace: `http://tfg-phishing.ml/ui/`.

Se puede acceder al código implementado a través del siguiente enlace: `http://gitlab.fdi.ucm.es/tfg2020/phishing-tfg.git`.

Figura 4.3: Captura de pantalla de la aplicación

4.2. Obtención de los Datos

Durante los últimos meses se han investigado posibles conjuntos de datos que contengan tanto URLs de páginas web legítimas como URLs de *phishing*. Sin embargo no existen muchos conjuntos de datos con páginas web legítimas y que no sean de pago y menos aún que combinen ambos tipos de páginas. Por lo tanto se han construido conjuntos de datos propios con la información encontrada en diferentes sitios. El primer conjunto de datos utilizado está compuesto por 18,835 URLs, de las cuales 9,000 son legítimas y 9,835 de *phishing*. Las URLs legítimas fueron extraídas de *Alexa's Top Sites* [Sit20] y las de *phishing* de la base de datos de [Phi20]. Con este conjunto de datos se ha trabajado al principio, hasta la construcción de otro de mayor tamaño.

El segundo conjunto de datos [MFSE14] cuenta con 96,018 URLs, de las cuales 48,009 son legítimas y 48,009 de *phishing*. El conjunto se filtra, para eliminar aquellas URLs que contienen caracteres extraños que no permiten su análisis. El número total es de 95,911 URLs de las cuales 48,009 son legítimas y 47,902 son de *phishing*. El único inconveniente que presenta este conjunto de datos es que no está actualizado. Esto supone que las páginas web de *phishing* suelen dejar de funcionar una vez pasados unos días, al ser ya detectadas y categorizadas. Por lo tanto a este conjunto de datos, se le adjuntan otros dos, cuyas URLs de *phishing* se extraen el mismo día. Para que el máximo número de URLs de *phishing* respondan, media hora antes de extraer las características, se une el conjunto de datos de 95,911 URLs con los dos conjuntos de URLs de *phishing* extraídas ese día de *PhishTank* y de *OpenPhish*, respectivamente. Por último se le añade URLs benignas de [Ebu20] para tener el mismo número de URLs legítimas que de *phishing*. Este proceso de juntar los datos fue realizado varias veces durante las pruebas.

El conjunto de datos final con el que se genera el archivo de características definitivo se desarrolló juntando los siguientes conjuntos de datos:

1. 95,911 URLs de las cuales 47,902 de *phishing* y 48,009 legítimas [MFSE14].

2. Extracción de *PhishTank* el mismo día [Phi20]: 13,897 URLs de *phishing*.

3. Extracción de *OpenPhish* el mismo día [Ope20]: 2,836 URLs de *phishing*.

4. URLs legítimas para tener el mismo número de URLs legítimas que de *phishing* sacadas del repositorio de *GitHub* [Ebu20] creado por los autores del trabajo analizado [SBDD19]: 11,627 URLs legítimas

5. Total: 129,270 URLs de las cuales 64,635 son de *phishing* y 64,635 son legítimas.

En la Tabla 4.1 se muestra todos los conjuntos de datos utilizados:

Tabla 4.1: Conjunto de datos utilizados

Fuente	Tamaño Phishing	Tamaño Legítimas	Tamaño Total	Método de extracción
PhishTank [Phi20] y Alexa TopSites [Sit20]	9,835	9,000	18,835	.csv de Alexa's Top Sites y peticiones a la API de PhishTank
[MFSE14]	48,009	48,009	96,018	.csv que publicó este estudio
[MFSE14] + PhishTank [Phi20] + Openphish [Ope20] + [Ebu20]	64,635	64,635	129,270	.csv [MFSE14], API de PhishTank, .txt de Openphish. Este proceso se realizó 3 veces. El resultado es el de la última extracción.

Se ha invertido una cantidad de tiempo considerable en la extracción de un conjunto de datos ideal para poder tener el de mayor tamaño posible y así entrenar mejor el modelo. En comparación a todos los trabajos analizados, se ha obtenido el conjunto de datos más grande, casi duplicando al de mayor tamaño encontrado (73,575 URLs).

4.3. Características Desarrolladas

El proceso de extracción de características se divide en: la extracción de características de la URL, y la extracción de características del HTML del DOM. Hay un total de 40. Dado el tiempo excesivo que requiere la ejecución de la característica `broken_links_ratio` se ha decidido mencionar la implementación pero no se utilizará durante las pruebas. Eso deja un total de 39 de las cuales 12 pertenecen a la extracción de características de la URL y las 27 restantes corresponden al contenido HTML del DOM.

4.3.1. Características de la Dirección de la Página Web

Para extraer las 12 características de la URL, se utiliza únicamente la dirección URL del sitio web. Es común en las URLs de las páginas de *phishing* que se observen ciertos patrones como una mayor longitud o un mayor número de puntos en las direcciones URL. Estas 12 características extraen algunos de esos patrones. Los trabajos mencionados anteriormente en el capítulo de trabajos relacionados, han servido de referencia para extraer estas características. Todas estas características se encuentran en el archivo `url_feat_extract.py`.

La Figura 4.4 muestra las partes que componen una URL. En adelante se hará referencia al dominio de segundo nivel como dominio.

Figura 4.4: Componentes de una URL

La Tabla 4.2 muestra la relación existente entre las características de las URLs implementadas en este trabajo y las funciones asociadas. Las funciones reciben como parámetro la URL de la página web a analizar. Todas las funciones devuelven -1 en caso de error.

Tabla 4.2: Características de la URL y sus funciones asociadas

Código	Característica	Función
F1	Dominio incrustado	embedded_domain_extract(url)
F2	Dirección IP	ip_address_extract(url)
F3	Número de puntos en una URL	dots_in_url_extract(url)
F4	Número de puntos en el nombre de equipo	dots_in_hostname_extract(url)
F5	Longitud del nombre de equipo	hostname_length_extract(url)
F6	Número de veces que aparece el símbolo “@”:	at_symbol_extract(url)
F7	Número de veces que aparece el símbolo “-”	dash_in_url_extract(url)
F8	URL sospechosa	suspicious_url_extract(url)
F9	Número de símbolos sospechosos	suspicious_symbols_extract(url)
F10	Protocolo HTTPS	https_extract(url)
F11	Dominio de nivel superior mal posicionado	tld_out_position_extract(url)
F12	Vocabulario sensible	sensitive_words_extract(url)

A continuación se enumeran las características implementadas dentro del archivo `url_feat_extract.py`.

- **F1 - Dominio incrustado:** Esta característica busca la presencia de un dominio, nombre de equipo o ambos en la ruta (*path*) de la URL. El motivo es que en ocasiones los atacantes incluyen en la ruta el dominio o nombre de equipo de la página web legítima, para así engañar a los usuarios ocultando en la barra del navegador el verdadero nombre de equipo de la URL. Devuelve 1 si encuentra un dominio incrustado, 0 en caso contrario.

- **F2 - Dirección IP:** Dada una URL, esta característica comprueba si el nombre de dominio es una dirección IP. Por ejemplo, la URL `http://209.133.122.70/paypalix0/ebmdata.com`. Si el dominio es una dirección IP devuelve 1. En caso contrario devuelve 0.

- **F3 - Número de puntos en una URL:** Esta característica devuelve el número de puntos que aparece en la URL. Es común que las URLs de *phishing* tengan un mayor número de puntos que las legítimas.

- **F4 - Número de puntos en el nombre de equipo:** Esta característica es similar a la anterior, pero en este caso devuelve el número total de puntos sólo dentro del

nombre de equipo de la URL, no de la URL completa. El nombre de equipo está compuesto por el subdominio y el dominio de la URL.

- **F5 - Longitud del nombre de equipo:** Dada una URL, esta característica devuelve la longitud del nombre de equipo de la URL. Normalmente las páginas web de *phishing* suelen tener mayor longitud que las páginas web legítimas.

- **F6 - Número de veces que aparece el símbolo "@":** Dada una URL, esta característica devuelve el número de veces que aparece el símbolo `"@"` en la URL.

- **F7 - Número de veces que aparece el símbolo "-":** Dada una URL, esta característica devuelve el número de veces que aparece el símbolo del guión `"-"` en la URL.

- **F8 - URL sospechosa:** Dada una URL, esta característica devuelve 1 si encuentra algún símbolo de `"@"` o de guión `"-"` en la URL. En caso contrario devuelve 0.

- **F9 - Número de símbolos sospechosos:** Dada una URL, esta característica devuelve el número de símbolos sospechosos que aparecen en la URL, es decir, si incluye los símbolos `"@", "\", "-" y "~"`. Es común en las URLs de *phishing* que aparezcan caracteres poco comunes, al ser generadas muchas de ellas aleatoriamente.

- **F10 - Protocolo HTTPS:** Dada una URL, esta característica devuelve 1 si la URL utiliza el protocolo HTTPS y 0 en caso contrario. Cómo se ha visto en capítulos anteriores, cada vez más URLs de *phishing* utilizan el protocolo HTTPS, pero se considera una característica importante porque la mayoría de las URLs legítimas si lo utilizan, por tanto si una URL no lo utiliza probablemente sea de *phishing*.

- **F11 - Dominio de nivel superior mal posicionado:** Dada una URL, esta característica extrae el Top Level Domain (TLD) o dominio de nivel superior de la URL y devuelve 1 si está mal posicionado. En caso contrario devuelve 0.

- **F12 - Vocabulario sensible:** Dada una URL, esta característica extrae el número de veces que aparece en la URL vocabulario sensible. Algunas de las palabras que se han incluido en el vocabulario sensible son: *secure*, *login*, *signin*, *banking*.

La Tabla 4.3 muestra la relación existente entre las características de la URL implementadas en este trabajo y las implementadas en los trabajas mencionados anteriormente.

Tabla 4.3: Relación de características de la URL y trabajos asociados

Código	[ZHC07]	[XHRC11]	[HHF+11]	[RP19]	[LYC+19]	[BAS19]	Este trabajo
F1	-	X	-	-	-	-	X
F2	X	X	X	-	X	X	X
F3	X	X	-	-	-	X	X
F4	-	-	-	X	-	-	X
F5	-	-	-	X	-	-	X
F6	-	-	-	X	-	X	X
F7	X	X	-	-	-	-	X
F8	X	X	-	-	-	-	X
F9	-	-	-	-	X	-	X
F10	-	-	-	X	X	X	X
F11	-	X	-	-	-	-	X
F12	-	X	-	-	X	-	X

4.3.2. Características del Modelo de Objeto de Documento

Para extraer las características del HTML de la página web, se utiliza el DOM, una interfaz de programación de aplicaciones para documentos HTML y Extensible Markup Language (XML). Define la estructura lógica de los documentos y el modo en el que se acceden y se manipulan. *Python* tiene la librería *Beautiful Soup* que permite analizar (parsear) una página web creando un árbol con todos los elementos del documento. Esto permite extraer información de las etiquetas, atributos y elementos fácilmente.

Los trabajos mencionados anteriormente en el capítulo de trabajos relacionados, han servido de referencia para extraer estas características. Todas estas características se encuentran en el archivo `dom_feat_extract.py`.

A continuación se enumeran las características implementadas dentro del archivo `dom_feat_extract.py`. Las funciones reciben como parámetro el árbol basado en el HTML de la página web. Algunas funciones también reciben como parámetro la URL de la página. Todas las funciones devuelven -1 en caso de error.

- **F13 - Número de enlaces vacíos:** Esta característica extrae el número de enlaces vacíos dentro del documento HTML. Las páginas web de *phishing* normalmente utilizan enlaces vacíos para simular que la página contiene muchos hipervínculos. Un ejemplo de un enlace vacío sería el siguiente: and

- **F14 - Número de enlaces internos:** Esta característica extrae el número de enlaces internos dentro del documento HTML. Para esto se extraen los dominios de todos los enlaces del documento HTML y se comparan con el dominio de la URL dada. Las páginas web de *phishing* normalmente utilizan pocos enlaces internos.

- **F15 - Número de enlaces externos:** Esta característica extrae el número de enlaces externos dentro del documento HTML. Para esto se extraen los dominios de todos los enlaces del documento HTML y se comparan con el dominio de la URL dada. Las páginas web de *phishing* normalmente utilizan muchos enlaces externos para enriquecer sus contenidos.

- **F16 - Número de enlaces totales:** Esta característica extrae el número total de enlaces dentro del documento HTML. Primero se buscan todos los enlaces que

contengan la etiqueta `"a"` dentro del documento HTML y se devuelve el número total de aquellos que tengan el atributo `href`.

- **F17 - Longitud del contenido HTML:** Esta característica extrae la longitud total del contenido del HTML dentro de las etiquetas `style`, `script`, `link`, `comment` y `form`. Normalmente, las páginas legítimas tienen mayor cantidad de contenido en estas etiquetas que las páginas web de *phishing*.

- **F18 - Formularios:** Esta característica busca todos los formularios dentro del contenido HTML. Si encuentra algún formulario devuelve 1 y 0 en caso contrario. Esto permite descartar las páginas web que no tienen ningún formulario por lo que no extraen información del usuario.

- **F19 - Formularios de tipo login:** El objetivo de las páginas web de *phishing* es obtener los datos del usuario, y esto se suele hacer mediante un formulario de `login`. Esta característica comprueba para cada formulario, si es de `login` o no. Para detectar si es de `login`, se ha creado un diccionario con un total de 19 palabras que suelen aparecer en estos formularios. Si encuentra alguna de estas palabras devuelve 1. Por último comprueba si dentro de los formularios hay algún tipo de etiqueta `<input>`. En algunos casos, los atacantes sustituyen el contenido de los formularios por imágenes para que no puedan ser detectados fácilmente. Si no encuentra ninguna etiqueta, devuelve 1. En caso contrario devuelve 0. Algunas de las palabras que incluye el diccionario son: [**contraseña, contrasenia, password, signin, email, DNI, correo, credit card, card number, número de tarjeta, CVV**].

- **F20 - Pop-ups de alerta:** Esta característica devuelve 0 si no se encuentra ninguna ventana de alerta. En caso contrario devuelve 1.

- **F21 - Número de recursos internos:** Esta característica extrae los recursos internos del código HTML. Para ello compara el dominio del recurso con el de la página web. Si son iguales, es un recurso interno. La función devuelve la longitud de los recursos internos encontrados.

- **F22 - Número de recursos externos:** Esta característica extrae los recursos externos del código HTML. Para ello compara el dominio del recurso con el de la página web. Si no son iguales, es un recurso externo. La función devuelve la longitud de los recursos externos encontrados.

- **F23 - Número de recursos totales:** Esta característica extrae todos los recursos encontrados dentro del código HTML, tanto internos como externos. Devuelve la longitud total de los recursos encontrados.

- **F24 - Redireccionamiento:** Algunos sitios web de *phishing*, diseñan páginas web legítimas para que el usuario acceda, y posteriormente les redireccionan a las páginas web de *phishing*. Esta característica busca en el contenido HTML patrones que indiquen redireccionamiento. Por ejemplo, busca dentro de las etiquetas `<meta>` o `<script>`.

- **F25 - Existencia de la marca de la URL entre las marcas más conocidas:** Esta característica utiliza un listado de marcas comerciales conocidas. Compara la marca extraída de la URL con las marcas del listado. Si hay coincidencias devuelve 0. En caso contrario devuelve 1.

- **F26 - Consistencia entre la marca que aparece en el título y la marca de la URL:** Esta característica devuelve 0 si la marca que aparece en el título de la página web coincide con la marca que aparece en la URL. En caso contrario devuelve 1.

- **F27 - Consistencia entre la marca más frecuente y la marca de la URL:** Esta característica extrae los nombres de marca de todos los enlaces y selecciona el más frecuente. Después lo compara con el nombre de marca que aparece en la URL. Si son iguales devuelve 0. En caso contrario devuelve 1. En la mayoría de los casos, el nombre de la marca coincide con el dominio. Por ejemplo en: `https://twitter.com/`.

- **F28 - Formularios sospechosos:** Esta característica devuelve 1 si detecta que el formulario es sospechoso. Un formulario es sospechoso si: contiene etiquetas `<input>`, la URL utiliza el protocolo HTTP o el campo `acción` del formulario está vacío.

- **F29 - Campos de acción sospechosos:** El código de autenticación en las páginas web de *phishing* se suele encontrar en el directorio actual y el campo `acción` del formulario suele ser un simple nombre de archivo. Esta característica devuelve 1 si el campo de `acción` del formulario está vacío, si señala a un archivo dentro del directorio actual o si apunta a un dominio diferente al de la página web.

- **F30 - Nombre de la marca en posición errónea:** Esta característica primero encuentra el dominio y subdominio más frecuentes dentro del contenido HTML, y supone que es la marca de la página. Después extrae la parte de la derecha y lo compara con el dominio de la página web (extraído del parámetro URL). Si son iguales, significa que el nombre de marca está en una posición errónea dentro de la URL del sitio, por lo que devuelve 1. En caso contrario devuelve 0. Un ejemplo podría ser `http://asdtdgfd.paypal.com`. Al examinar el contenido HTML, `paypal` resulta ser la marca de la página. Si no fuese una página web de *phishing* la marca coincidiría con `asdtdgfd`. En este caso coincide con la parte de la derecha `paypal` por lo que el nombre de marca se encuentra en una posición errónea, algo muy común en las páginas web de *phishing*.

- **F31 - Contenido oculto o restringido:** Esta característica busca contenido escondido o restringido dentro de las etiquetas `<div>` `<input>` y `<button>`. Si el contenido en alguna de estas etiquetas tiene visibilidad `hidden`, el atributo `show` a `none` o algún botón está deshabilitado, esta función devuelve 1. En caso contrario devuelve 0.

- **F32 - Frecuencia con la que aparece el dominio en los hipervínculos:** Esta característica examina todos los hipervínculos del contenido HTML y compara el dominio más frecuente con el dominio de la URL del sitio web. Si son iguales devuelve 0. En caso contrario devuelve 1.

- **F33 - Proporción de enlaces vacíos en la página:** Esta característica extrae el número de enlaces vacíos del contenido HTML y devuelve la proporción dividiendo el número de enlaces vacíos encontrados entre el número de enlaces totales. Las páginas web de *phishing* suelen incluir enlaces vacíos para intentar convencer al usuario de que es una página web legítima. Un ejemplo de enlace vacío sería el siguiente: `<a href="#"> Condiciones de uso </a>`

- **F34 - Proporción de enlaces vacíos en el pie de página:** Esta característica extrae el número de enlaces vacíos dentro del pie de página y devuelve la proporción dividiendo el número de enlaces vacíos encontrados entre el número de enlaces totales que hay en el pie de página.

- **F35 - Hipervínculos en el cuerpo del HTML:** Esta característica busca hipervínculos o *anchor links* dentro de la etiqueta `<body></body>` del contenido HTML. Los hipervínculos tienen la etiqueta `a` como en los siguientes ejemplos: `<a href=" "></a>`, `<a href="#"></a>`. Si encuentra alguno devuelve 1. En caso contrario devuelve 0.

- **F36 - Número de veces que aparece el enlace más común:** Esta característica extrae el enlace más común de todo el contenido HTML y cuenta cuántas veces aparece. Después calcula la proporción dividiéndolo por el número total de enlaces.

- **F37 - Número de veces que aparece el enlace más común en el pie de página:** Esta característica extrae el enlace más común del pie de página y cuenta cuántas veces aparece. Después calcula la proporción dividiéndolo por el número total de enlaces en el pie de página.

- **F38 - Número de veces que aparece el nombre de la marca:** Es común en las páginas web legítimas, que el nombre de la marca aparezca repetidas veces debido al uso frecuente de recursos internos. Esta característica devuelve el número de veces que aparece el nombre de la marca en los enlaces del HTML.

- **F39 - Número de enlaces rotos:** Esta característica extrae todos los enlaces y comprueba si estos existen o no haciendo un *request* de la cabecera del enlace. Si el enlace no existe o no responde, incrementa el contador de enlaces rotos. Después calcula la proporción de enlaces rotos dividiendo el número de enlaces rotos entre el total de enlaces de la página. Esta característica es la que más problemas ha dado a lo largo de todos los experimentos al tener que hacer por cada página web, un *request* a cada enlace que contiene.

- **F40 - Número de puntos en los enlaces:** Esta característica cuenta el número de puntos de cada enlace que aparece en el HTML. Después hace la media entre el número de puntos totales y el total de enlaces.

La Tabla 4.4 muestra la relación existente entre las características del DOM implementadas en este trabajo y las funciones asociadas.

Tabla 4.4: Características del DOM y sus funciones asociadas

Código	Característica	Función
F13	Número de enlaces vacíos	number_of_empty_links_extract(soup)
F14	Número de enlaces internos	number_of_internal_links_extract(soup,url)
F15	Número de enlaces externos	number_of_external_links_extract(soup,url)
F16	Número de enlaces totales	number_of_all_links_extract(soup)
F17	Longitud del contenido HTML	content_length_extract(soup)
F18	Formularios	form_extract(soup)
F19	Formularios de tipo login	is_login_form(soup)
F20	Pop-ups de alerta	alarm_window_extract(soup)
F21	Número de recursos internos	number_of_internal_resources_extract(soup,url)
F22	Número de recursos externos	number_of_external_resources_extract(soup,url)
F23	Número de recursos totales	number_of_resources(soup,url)
F24	Redireccionamiento	redirection_extended(soup)
F25	Existencia de la marca de la URL entre las marcas más conocidas	consistency_between_dataset_urlbrand(url,word_list)
F26	Consistencia entre la marca que aparece en el título y la marca de la URL	consistency_between_titlebrand_urlbrand(soup,url)
F27	Consistencia entre la marca más frecuente y la marca de la URL	consistency_between_most_frequentlinkbrand_urlbrand(soup,url)
F28	Formularios sospechosos	bad_forms_cant(soup,url)
F29	Campos de acción sospechosos	bad_action_fields_cant(soup,url)
F30	Nombre de marca en posición errónea	out_of_position_brand_name_cant(soup,url)
F31	Contenido oculto o restringido	hidden_or_restricted_information(soup)
F32	Frecuencia con la que aparece el dominio en los hipervínculos	frequency_of_domain_in_anchor_links(soup,url)
F33	Proporción de enlaces vacíos en la página	null_anchor_links_ratio_in_website(soup,url)
F34	Proporción de enlaces vacíos en el pie de página	null_links_ratio_footer(soup)
F35	Hipervínculos en el cuerpo del HTML	presence_of_anchor_links_in_html_body(soup)
F36	Número de veces que aparece el enlace más común	common_page_detection_ratio_in_website(soup)
F37	Número de veces que aparece el enlace más común en el pie de página	common_page_detection_ratio_in_footer(soup)
F38	Número de veces que aparece el nombre de la marca	number_of_urlbrandname_inhtmlcode(soup,url)
F39	Número de enlaces rotos	broken_links_ratio(soup)
F40	Número de puntos en los enlaces	number_of_dots_in_allurls(soup)

La Tabla 4.5 muestra la relación existente entre las características del DOM implementadas en este trabajo y las implementadas en los trabajos mencionados anteriormente.

Tabla 4.5: Relación de características del DOM y trabajos asociados

Código	[ZHC07]	[XHRC11]	[HHF+11]	[RP19]	[LYC+19]	[BAS19]	Este trabajo
F13	-	-	-	-	X	-	X
F14	-	-	-	-	X	-	X
F15	-	-	-	-	X	-	X
F16	-	-	-	-	X	-	X
F17	-	-	-	-	X	-	X
F18	X	X	-	-	-	-	X
F19	X	-	-	-	X	-	X
F20	-	-	-	-	X	X	X
F21	-	-	-	-	X	-	X
F22	-	-	X	-	X	-	X
F23	-	-	-	-	X	-	X
F24	-	-	-	-	X	X	X
F25	-	-	-	-	-	-	X
F26	-	-	-	-	X	-	X
F27	-	X	-	-	X	-	X
F28	-	X	-	-	-	-	X
F29	-	X	-	-	-	-	X
F30	-	X	-	-	-	-	X
F31	-	-	-	-	X	-	X
F32	-	-	-	X	-	-	X
F33	-	-	X	X	-	-	X
F34	-	-	X	-	-	-	X
F35	-	-	-	X	-	-	X
F36	-	-	-	X	-	-	X
F37	-	-	-	X	-	-	X
F38	-	-	-	-	X	-	X
F39	-	-	-	X	-	-	X
F40	-	-	X	-	-	-	X

4.4. Extracción de las Características

En esta sección se explica como ha sido el proceso de extracción de las características mencionadas. A lo largo de los últimos meses, se han extraído las características en repetidas ocasiones, corrigiendo errores y observando cómo influían sobre los diferentes conjuntos de datos explicados anteriormente. El archivo que contiene las funciones necesarias para el proceso de extracción se llama `features_extraction_test.py`

Al comenzar a utilizar el conjunto de datos de las 18,835 URLs, la extracción de las características era relativamente rápida. Pero se podía observar que muchas de las columnas del .csv generado no se estaban rellenando bien. En este momento, la característica `broken_links_ratio` estaba comentada. Conforme se hicieron diferentes pruebas, se pudo ver que la librería de *Python BeautifulSoup* que extrae el contenido HTML de la página, no conseguía acceder a muchas de ellas, y aparecían errores del tipo

`connection time out` o `max retries exceeded`. Esto se debía a varios factores. Por un lado, todos los conjuntos de datos, tanto los seleccionados como los descartados, contienen muchas URLs parecidas es decir, URLs con el mismo dominio pero diferente página, como por ejemplo serían `https://recursospython.com/category/guias-y-manuales/` y `https://recursospython.com`. Esto hacía que muchas veces se llamara consecutivamente al mismo dominio con la función de *BeautifulSoup*, por lo tanto la página no dejaba seguir intentándolo. Además se utilizaba un mismo agente de usuario o *User agent* para las sesiones por lo que la página web detectaba que las peticiones venían todas del mismo sitio. También se pudo ver que en la función que llama a *BeautifulSoup*, no estaba implementado ningún mecanismo para que reintentara una misma URL varias veces si daba error la primera vez. En esta primera etapa, se realizaban las pruebas en local, y cada extracción de características duraba una hora aproximadamente.

Cuando se consiguió el conjunto de datos de las 96,018 URLs se empezaron a implementar medidas para evitar que dieran los mismos errores con este conjunto de datos. Este conjunto de datos se tuvo que filtrar porque contenía caracteres ilegibles, y se quedó en 95,011 URLs. Lo primero que se hizo fue crear una lista de agentes de usuario o *User agents* para que la página no supiera que las peticiones venían todas del mismo sitio. Hay dos archivos, uno se llama `users.py` que contiene los agentes de usuario para ordenador y `user_agents.py` que contiene los agentes de usuario para dispositivos móviles. Después se puso en la función que llama a *BeautifulSoup* para extraer el contenido HTML, el código necesario para que utilizando diferentes adaptadores, agentes de usuario y tiempos de espera, consiguiese que el número de páginas web de las que se podía extraer el contenido fuese mayor. En esta segunda etapa, apareció otro problema. El conjunto de datos había aumentado considerablemente y la extracción de características (sin contar la de `broken_links_ratio` que aún seguía comentada) tardaba mucho. Para solucionar este problema, se utilizaron multihilos. El uso de multihilos permite crear más de un hilo de un mismo proceso. Esto permitió poder extraer las características más rápido. Las primeras pruebas con este conjunto de datos se hacían también en local y cada extracción de características duraba aproximadamente 4 horas y media.

Con este mismo conjunto de datos, se empezó a probar la característica `broken_links_ratio`. Esta característica había estado comentada en anteriores ocasiones porque cuando se intentó probarla, tardaba tanto tiempo que no se lograba terminar ninguna extracción con ella. La característica extrae el número de enlaces rotos para cada URL. Esto supone que para cada una de las 95,011 URLs extraía todos los enlaces y enviaba un *request* de la cabecera para ver el código de respuesta que devolvía la página. Según la característica que calcula el número de enlaces totales de cada página, algunas pueden llegar a tener más de 1,000 enlaces. Esto hacía que el coste de añadir esta función fuese muy grande. La intención era añadir esta característica puesto que parecía importante, porque las páginas legítimas en su mayoría tienen todos los enlaces funcionando correctamente, mientras que las de *phishing* es mucho más probable que no.

En este momento se dio la oportunidad de utilizar una máquina virtual para poder enviar allí la extracción de características y al tener mayor capacidad de cómputo, poder terminar antes la ejecución e incluso llegar a ejecutar la característica `broken_links_ratio` sin problemas. La máquina virtual virtual estaba alojada en *Google Cloud Platform* y contaba con 64 GigaByte (GB) de disco de arranque, 22 virtual Central Processing Unit (vCPU)s y 137,5 GB de memoria. Cuando se probaba la extracción de características sin la función `broken_links_ratio`, iba bastante mejor, llegando a durar apenas un par de horas. Pero todavía no se conseguía solucionar el problema totalmente. Otra mejora que se añadió fue cambiar el uso de multihilos por multiprocesamiento.

Al tener la máquina 22 vCPUs, podría ejecutar más de un proceso al mismo tiempo y así reducir aún más el tiempo. Aún así, aunque disminuía considerablemente el tiempo de extracción de las demás características, `broken_links_ratio` seguía sin acabar correctamente.

En esta última etapa de extracción, ya se utiliza el formato de conjunto de datos final, corrigiendo el conjunto de datos de 95,011 URLs y añadiéndole las que se extrajeron ese día de *phishing* y las correspondientes páginas web legítimas para que hubiese la misma cantidad de ambas. En total se contaba con aproximadamente 130,000 URLs. Este proceso de juntar los conjuntos de datos se hizo cada cierto tiempo por lo que a veces se obtenían más de 130,000 y a veces en torno a 129,000, dependiendo de las URLs de *phishing* que devolvieran *OpenPish* y *PhishTank*. Después de varios intentos con la característica `broken_links_ratio`, se decidió apartarla. Para estas últimas extracciones, se mezclaron aleatoriamente las URLs dentro del conjunto de datos antes de realizar la extracción, de tal forma que si existían URLs con el mismo dominio consecutivas, estuviesen dispersas a lo largo del conjunto de datos. Esto ayudó a poder extraer características de un mayor número de URLs.

Para continuar con el siguiente paso y analizar el archivo con todas las características extraídas, se aplicó preprocesamiento a los datos. Si una URL devolvía más de 15 valores a "-1" (es decir, que la característica no ha podido ser extraída por un error) esa URL se eliminaba. La última extracción realizada fue con la que se entrenaron los algoritmos de aprendizaje automático y con un total de 129,270 URLs. Después de aplicar este preprocesamiento quedaron 68,729 URLs, aproximadamente la mitad del conjunto de datos inicial.

4.5. Configuración y Análisis de los Algoritmos Utilizados

Una vez extraídas y preprocesadas las características, se generaba el archivo `extraction_result.csv`. La versión final de este archivo está formada por 68,729 filas (sin contar la cabecera) y 40 columnas. Las 39 primeras corresponden a las características, y la última es la columna que indica si las características extraídas de esa fila corresponden a una página web de *phishing* (el valor es 1) o no (el valor es 0).

El primer paso consistía en dividir las características en dos conjuntos, conjunto de entrenamiento y conjunto de prueba. El conjunto de entrenamiento es el que se ha utilizado para afinar y entrenar el modelo. El 70 % del conjunto total corresponde al conjunto de entrenamiento, y el 30 % restante corresponde al conjunto de prueba. Una vez se habían separado ambos conjuntos, lo siguiente fue aplicar una escala. Se hicieron pruebas aplicando en algunos casos estandarización y en otros casos normalización. En el contexto del aprendizaje automático, normalizar significa escalar los valores para que se encuentren en el rango [0,1]. Por otro lado, estandarizar significa escalar los valores para que tengan una media de 0 y una desviación típica de 1. En la mayoría de los casos se ha aplicado normalización, porque algunos de los algoritmos que han sido analizados sólo aceptaban datos entre 0 y 1.

Después de haber aplicado la escala, el siguiente paso fue seleccionar los algoritmos de clasificación a probar. Estos fueron: `BernoulliNB`, `LinearSVC`, `GradientBoosting`, KNN, RF, `AdaBoost`, `Árboles de decisión`, MLP, Stochastic Gradient Descent (SGD), Support Vector Classification (SVC), `GaussianNB` y `Análisis Discriminante Lineal`. También se han hecho pruebas con `Redes neuronales`.

4.5.1. Configuración de los Hiperparámetros

Para encontrar los hiperparámetros adecuados para cada algoritmo, se han utilizado las funciones `GridSearchCV` y `RandomizedSearchCV` que pertenecen a la biblioteca `Scikit-learn`. La diferencia entre ambas es que `GridSearchCV`, dados unos rangos para unos parámetros, prueba todas las combinaciones existentes y devuelve la mejor combinación. En cambio `RandomizedSearchCV` dados unos rangos para unos parámetros y un número de iteraciones, realiza tantas combinaciones como número de iteraciones y devuelve la mejor combinación encontrada. Ambos hacen este proceso mediante validación cruzada o `cross-validation`. Se ha intentado aplicar `GridSearchCV` a todos los algoritmos elegidos, pero por el gran tamaño del conjunto de entrenamiento en muchas ocasiones el `GridSearchCV` no terminaba en un tiempo razonable. Con ese grupo de algoritmos se utilizó `RandomizedSearchCV`. El archivo en el que se hace la configuración de hiperparámetros se llama `find_best_hyperparams.py`.

Los algoritmos en los que se ha usado `GridSearchCV` son `BernoulliNB`, `AdaBoost`, `Árboles de decisión`, `LinearSVC`, `GaussianNB`, `Análisis Discriminante Lineal` y `Redes neuronales`.

Por otro lado, los algoritmos en los que no se ha podido usar `GridSearchCV` y se ha optado por usar `RandomizedSearchCV` son: KNN, `GradientBoosting`, RF, MLP, SGD y SVC. Normalmente, el número de iteraciones (combinaciones entre los parámetros seleccionados) estaba a 1,000. Pero para algunos algoritmos como el KNN se tuvo que bajar considerablemente.

En todas las pruebas se fueron calibrando los hiperparámetros para cada algoritmo hasta encontrar los óptimos. Además ambas pruebas se han realizado con los valores de validación cruzada de 5 y 10 respectivamente.

Algunas de estas pruebas (en particular cuando eran muchas combinaciones) se han hecho en la máquina virtual. Para optimizar al máximo los recursos, en las funciones de `GridSearchCV` y `RandomizedSearchCV` se ha establecido el parámetro `n_jobs` a "-1" para que utilice todos los procesadores disponibles. Como estas pruebas tenían tantas combinaciones, a veces la sesión establecida con la máquina virtual caducaba o se desconectaba, por lo que la ejecución se terminaba al estar ligada a la sesión. Para solucionar este problema, se utiliza el comando `nohup` que permitía seguir ejecutando un proceso aunque la sesión que lo había iniciado se cerrara.

4.6. Tecnologías Utilizadas

Durante el desarrollo de este trabajo se han utilizado diferentes herramientas para la implementación del código, la comunicación, las pruebas y la realización de la memoria. Para la comunicación se ha elegido la herramienta *Google Meet*, que es una aplicación de videoconferencias. Se ha elegido esta herramienta por su interfaz rápida y ligera, pero sobre todo porque ofrece reuniones en línea seguras y de fácil acceso. Además en la comunicación, se ha utilizado *TeamViewer* que es un software que permite conectarse remotamente a otro equipo, que ha permitido compartir y controlar escritorios, transferencia de archivos entre ordenadores y reuniones en línea. Para la implementación de código y el control de versiones se ha optado por la herramienta *GitLab* que ha permitido el desarrollo de software colaborativo y ha servido como repositorio al mismo tiempo. Una vez terminado el código y hechas las pruebas se ha pasado a la realización de la memoria, que ha requerido de una nueva herramienta. *LaTex* ha sido la herramienta elegida para llevar a cabo esta tarea. *LaTex* permite la fácil generación de todo tipo de índices y listados de bibliografía

citada y genera un texto escrito según el estilo de una revista científica.

4.6.1. Desarrollo del Aprendizaje Automático

Una de las herramientas más importantes en el desarrollo de un SaaS es el uso de un buen Integrated Development Environment (IDE), o Entorno de Desarrollo Integrado, que permita implementar de forma eficiente un programa en el lenguaje *Python*. Es por ello que se decidió utilizar la herramienta *Pycharm*, uno de los IDEs más potentes que existe ya que cuenta con asistencia, análisis y completitud de código. También cuenta con *plug-ins* que permiten gestionar los proyectos con herramientas de control de versiones como *Gitlab*.

Para el desarrollo del aprendizaje automático del proyecto se ha utilizado la librería *Scikit-learn*, que incluye varios algoritmos de clasificación y regresión como SVM, RF o KNN. Esta biblioteca también permite validar los modelos y sus mejores parámetros mediante clases como `GridSearchCV` o `RandomizedSearchCV`. También permite aplicar si es necesario preprocesamiento a los datos que así lo requieran con clases como `MinMaxScaler` o `StandardScaler`. Para la obtención de las características de forma concurrente se han utilizado las clases `ThreadPoolExecutor` para una ejecución multihilos y `ProcessPoolExecutor` para multiprocesos. Esto permitió reducir notablemente el tiempo de extracción de las características.

Muchas pruebas han sido llevadas a cabo en una instancia virtual en *Google Cloud Platform*. Esta instancia contaba con 64 GB de disco de arranque, 22 vCPUs y 137,5 GB de memoria. En esta máquina virtual se instaló el correspondiente archivo `requirements.txt` para tener todas las librerías y poder llevar a cabo las pruebas correctamente.

4.6.2. Desarrollo de la API y de la Aplicación Web

Para la API REST se ha creado una base de datos local con *SQLite*, que es un sistema de gestión de bases de datos relacional muy sencillo de usar y que ocupa poco espacio. Para poder establecer la relación de la base de datos con el resto del código, se ha utilizado *SQLAlchemy*, un kit de herramientas Structured Query Language (SQL) que permite conectar la interfaz de comunicación con la base de datos. Además se ha utilizado *Flask*, un "micro" *framework* que permite facilitar la creación de Aplicaciones Web y *Marshmallow* para la serialización y deserialización de objetos. Para manejar las peticiones HTTP y mapear los *endpoints* a las funciones escritas en *Python* se ha utilizado *Connexion* y *Swagger*. La conexión con la API de *Google Safe Browsing* se ha hecho mediante una API *key* que permitía poder hacer las peticiones. Para comprobar que las peticiones se hacían correctamente se ha utilizado *Postman*, una herramienta que permite el envío de peticiones HTTP sin necesidad de desarrollar un cliente.

Para la Aplicación Web sencilla, se ha utilizado el lenguaje de programación *JavaScript*, el lenguaje de diseño gráfico Cascade Style Sheets (CSS) y el lenguaje de marcado HTML. Para depurar en *JavaScript*, hemos utilizado *DevTools* de *Google Chrome* que permite poner *breakpoints* y ejecutar sólo una parte del código.

Tanto la API cómo la Aplicación Web sencilla, han sido desplegadas en un *Docker* y subidas a una instancia virtual. Un *Docker* es un contenedor que permite levantar máquinas independientes con sistemas operativos ligeros en poco tiempo.

Capítulo 5

Experimentos y Resultados

Una vez obtenidos los mejores hiperparámetros para cada clasificador, se llevan a cabo los experimentos. Estos han sido realizados con el conjunto de características resultante tras filtrar el conjunto de datos de 129,270 URLs. Quedando un total de 68,729 URLs con sus correspondientes características extraídas. Para cada uno de los 12 algoritmos seleccionados, se han realizado diferentes experimentos con distintos enfoques de los que se han extraído dos resultados. También se han llevado a cabo diversas pruebas con diferentes conjuntos de características. Posteriormente estos resultados han sido analizados y comparados entre sí para determinar la mejor configuración y el mejor resultado para cada algoritmo. En la etapa final de los experimentos, se ha llevado a cabo un análisis del modelo elegido utilizando la curva de validación.

5.1. Métricas Utilizadas

Se define **P** como páginas web de *phishing* y **N** como páginas web legítimas.

Se define **TP** o verdadero positivo como el número de páginas web de *phishing* que han sido correctamente etiquetadas como *phishing*.

Se define **FP** o falso positivo como el número de páginas web legítimas que han sido incorrectamente etiquetadas como *phishing*.

Se define **TN** o verdadero negativo como el número de páginas web legítimas que han sido correctamente etiquetadas como legítimas.

Se define **FN** o falso negativo como el número de páginas web de *phishing* que han sido incorrectamente etiquetadas como legítimas.

Las métricas que se utilizan son las siguientes:

- **Sensibilidad *(sensitivity)***: También conocida como *recall* o tasa de verdaderos positivos es la proporción de páginas web de *phishing* etiquetadas correctamente por el algoritmo.

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

- **Especificidad *(specificity)***: También conocida como tasa de verdaderos negativos es la proporción de páginas web legítimas etiquetadas correctamente por el algoritmo.

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

- **Precisión *(accuracy)***: Proporción de páginas web legítimas y de *phishing* que han sido correctamente etiquetadas por el algoritmo.

$$Precisión = \frac{TP + TN}{P + N}$$

- **Exactitud *(precision)***: Proporción entre los positivos reales predichos por el algoritmo y todos los casos positivos.

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

- **Tasa de falsos positivos *(false positive rate)***: Proporción de páginas web legítimas clasificadas incorrectamente respecto al número total de páginas web legítimas.

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

- **Tasa de falsos negativos *(false negative rate)***: Proporción de páginas web de phishing clasificadas incorrectamente respecto al número total de páginas web de phishing.

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

- **Tasa de error *(error rate)***: Proporción de páginas web tanto legítimas como de *phishing* etiquetadas incorrectamente respecto al número total de páginas web.

$$ERR = \frac{FP + FN}{P + N}$$

.

- **Valor-F1 *(F1-score)***: Media harmónica de la exactitud y la sensibilidad.

$$F1 - score = 2 \cdot \frac{exactitud \cdot sensibilidad}{exactitud + sensibilidad}$$

5.2. Experimentos

Para cada experimento, se han evaluado dos resultados. El primer resultado ha sido calculado de la siguiente manera:

- Se divide el conjunto de características en dos: conjunto de entrenamiento (70 % del conjunto total) y conjunto de prueba (30 % restante) mediante la función `train_test_split()`.
- Se normalizan los datos.
- Se entrena el modelo seleccionado con el conjunto de entrenamiento.
- Se predice el resultado del conjunto de prueba sobre el modelo entrenado.
- Se extraen las métricas.

El segundo resultado ha sido calculado de la siguiente manera:

- Se normaliza el conjunto de características completo.
- Se aplica muestreo estratificado o `stratified sampling`, y validación cruzada al conjunto de características. Esto se hace mediante la función `cross_val_score()` que devuelve la puntuación recibida de cada iteración que realiza. La media de estas puntuaciones corresponde a la precisión o `accuracy` del modelo.
- Se predice el resultado del conjunto utilizando la función `cross_val_predict()` que aplica también validación cruzada para extraer la predicción.
- Se extraen las métricas.

El número de particiones de la validación cruzada viene determinado por el parámetro `cv`. Si este parámetro es igual a 10, la función realizará el procedimiento de ajuste 10 veces por lo que seleccionará cada vez un 10 % como conjunto de prueba y el 90 % resultante como conjunto de entrenamiento.

En aprendizaje automático, hay un fenómeno llamado sobreajuste o *overfitting* que tiene lugar cuando un modelo ajusta los datos de entrenamiento hasta un punto que no predice bien los datos nuevos. Una solución a este problema es aplicar validación cruzada. Comparando estos dos resultados de cada experimento, se quiere analizar cómo afecta a la precisión de los modelos el aplicar validación cruzada (probando diferentes valores de `cv`) en vez de una única división del conjunto de datos (cómo sería en el primer resultado) y si este primer resultado podría estar sobre ajustando el modelo excesivamente.

A continuación se explican los experimentos mas relevantes para cada modelo:

1. **Experimentos Realizados con el Algoritmo KNN**

   El algoritmo KNN o k-vecinos más próximos, estima la similitud entre el dato a clasificar y los datos utilizados en la etapa de entrenamiento. Clasifica el dato con la misma etiqueta que tenga el dato de entrenamiento más cercano. Si tiene más de un dato de entrenamiento cercano, se le asigna la etiqueta que sea más común entre sus vecinos más cercanos. El valor que determina el número de vecinos más cercanos es `k`.

   En la Tabla 5.1 se puede observar que cuando se comparan los resultados a los que se les aplica validación cruzada, se obtienen mejores resultados cuándo el número de veces que se realiza la validación cruzada es 10. Este resultado es bastante interesante, puesto que cuanto mayor es el valor de `cv`, menor probabilidad hay de que se produzca sobreajuste en el modelo y aún así, los datos son mejores.

Tabla 5.1: KNN con validación cruzada

CV	Precisión	Sensibilidad	Exactitud	Valor-F1	ERR	TN	FP	FN	TP	Conf.
5	0.886	0.800	0.824	0.811	0.113	44,093	3,582	4,193	16,861	c1
	0.878	0.763	0.826	0.793	0.121	44,312	3,363	4,984	16,070	c2
	0.896	0.789	0.862	0.823	0.103	45,017	2,658	4,425	16,629	c3
10	0.896	0.824	0.836	0.829	0.103	44,268	3,407	3,686	17,368	c1
	0.893	0.788	0.852	0.818	0.106	44,813	2,862	4,456	16,598	c2
	0.908	0.814	0.878	0.858	0.091	45,311	2,364	3,903	17,151	c3

```
c1: metric="manhattan", n_neighbors=1, p=1 weights='uniform'
c2: metric="minkowski", n_neighbors=5, p=2 weights='uniform'
c3: metric='manhattan', n_neighbors=7, p=2 weights='distance'
```

En las pruebas que se han hecho con este algoritmo, el mejor resultado se obtiene con la configuración de parámetros `c3`, dónde el parámetro `k` toma el mayor valor de los que se han probado `n_neighbors=7`. Además el parámetro que determina el cálculo de la importancia o peso de los diferentes vecinos, está puesto para que puntúen más alto los vecinos cuánto mas cercanos estén.

Cuando se comparan los resultados generados a los que se les han aplicado validación cruzada con los resultados generados utilizando `train_test_split` como se observa en la Tabla 5.2 con este último método se obtienen mejores resultados. Esto es debido a que al realizar una sola partición, hay más posibilidades de que el modelo se sobreajuste a los datos y dé un mejor resultado, al no hacer la media de los `k` resultados obtenidos como se hace en el método de validación cruzada. El mejor resultado obtenido con este algoritmo ha sido con el método `train_test_split` y da una precisión de 93,4 % utilizando la configuración `c3`.

El resultado más ajustado obtenido con este algoritmo ha sido con el método de validación cruzada con `cv=10`, y da una precisión del 90,8 %. Esto es un 3,8 % menos que utilizando el otro método.

Tabla 5.2: KNN con train_test_split

División	Precisión	Sensibilidad	Exactitud	Valor-F1	ERR	TN	FP	FN	TP	Conf.
70 % 30 %	0.927	0.889	0.874	0.881	0.072	13,526	803	698	5,592	c1
	0.918	0.843	0.884	0.863	0.081	13,637	692	985	5,305	c2
	0.934	0.879	0.903	0.890	0.065	13,737	592	759	5,531	c3

2. Experimentos Realizados con el Algoritmo BernoulliNB

BernoulliNB es un algoritmo de clasificador probabilístico del tipo bayesiano ingenuo. Este algoritmo asume que las características toman valores binarios, es decir 0 o 1 y aplica el teorema de *Bayes* con la suposición "ingenua" de independencia condicional entre cada par de características. Se han hecho pocas pruebas con este algoritmo dado que los resultados obtenidos han sido bastante bajos en comparación con otros algoritmos.

Las pruebas hechas con validación cruzada que se pueden observar en la Tabla 5.3, dan una precisión de un 79,8 %, que es un valor muy bajo comparado con otros algoritmos que se han probado. En este algoritmo, los datos obtenidos con el parámetro `cv=5` y los datos obtenidos con el parámetro `cv=10` son muy similares. Incluso es interesante observar que a diferencia de otros algoritmos, aunque la precisión es mayor con `cv=10`, la sensibilidad del modelo es mayor con `cv=5`.

Tabla 5.3: BernoulliNB con validación cruzada

CV	Precisión	Sensibilidad	Exactitud	valor-F1	ERR	TN	FP	FN	TP	Conf.
5	0.797	0.665	0.670	0.667	0.202	40,779	6,896	7,050	14,004	c1
	0.797	0.665	0,670	0.667	0.202	40,779	6,896	7,052	14,002	c2
10	0.798	0.659	0.675	0.669	0.201	41,013	6,662	7,171	13,883	c1
	0.798	0.658	0.675	0.666	0.201	41,013	6,662	7,183	13,871	c2

```
c1: alpha=0.21842105263157896, binarize=0.0, fit_prior=True
c2: alpha=0.1, binarize=0.0, fit_prior=True
```

En este caso también se observa la diferencia de precisión entre los valores a los que se les había aplicado validación cruzada y los valores calculados usando la función `train_test_split` aunque la sensibilidad obtenida en ambos casos es muy similar.

Como se puede observar en la Tabla 5.4, el mejor resultado obtenido para el algoritmo BernoulliNB ha sido con el método `train_test_split` y da una precisión de 80,1 % utilizando tanto la configuración `c1` como la `c2`. La sensibilidad y la exactitud también son iguales en ambas configuraciones, la única diferencia apreciable es el número de FP, que toma un valor más con la configuración `c2`.

El resultado más ajustado obtenido con este algoritmo es con el método de validación cruzada con `cv=10`, y da una precisión de 79,8 %. Esto supone un 0,3 % menos que utilizando el otro método. En este algoritmo el uso de un método de entrenamiento u otro no supone una gran diferencia.

Tabla 5.4: BernoulliNB con train_test_split

División	Precisión	Sensibilidad	Exactitud	Valor-F1	ERR	TN	FP	FN	TP	Conf.
70 % 30 %	0.801	0.665	0.677	0.670	0.198	12,335	1,994	2,101	4,189	c1
	0.801	0.665	0.677	0.670	0.198	12,334	1,995	2,101	4,189	c2

3. Experimentos Realizados con el Algoritmo Árboles de decisión

Los Árboles de decisión o *Decision Trees* son algoritmos de clasificación que utilizan aprendizaje inductivo. Cada nodo del árbol representa una característica y cada rama representa un posible valor que puede tomar dicha característica. Mediante este proceso y una serie de reglas de decisión que infiere de los datos, predice la clase a la que pertenecen. Cuanto más profundo sea el árbol, más complejas serán las reglas y más ajustado estará el modelo.

Como se puede observar en la Tabla 5.5 las pruebas hechas con validación cruzada vuelven a dar mejor resultado tanto para la precisión como para la sensibilidad y la exactitud con el parámetro `cv=10`.

Tabla 5.5: Árboles de decisión con validación cruzada

CV	Precisión	Sensibilidad	Exactitud	Valor-F1	ERR	TN	FP	FN	TP	Conf.
5	0.901	0.816	0.865	0.839	0.098	44,999	2,676	3,862	17,192	c1
	0.900	0.818	0.847	0.832	0.099	44,583	3,092	3,814	17,240	c2
	0.900	0.797	0.865	0.829	0.099	45,065	2,610	4,263	16,791	c3
10	0.911	0.832	0.869	0.850	0.088	45,005	2,620	3,522	17,532	c1
	0.908	0.837	0.857	0.846	0.091	44,750	2,925	3,424	17,630	c2
	0.908	0.815	0.876	0.844	0.091	45,261	2,414	3,874	17,180	c3

```
c1: max_depth=22, min_samples_leaf=1, min_samples_split=2
c2: max_depth=none, min_samples_leaf=1, min_samples_split=2
c3: max_depth=20, min_samples_leaf=1, min_samples_split=15
```

Como se puede observar en la Tabla 5.6 el cambiar el valor de los parámetros `min_samples_split` que corresponde al mínimo valor de muestras necesarias para dividir un nodo y `max_depth` que corresponde a la profundidad máxima que puede tomar el árbol, no supone un gran cambio en la precisión del algoritmo, como máximo de un 0.002 %.

El mejor resultado obtenido para el algoritmo Árboles de decisión ha sido con el método `train_test_split` y da una precisión de 93 % utilizando tanto la configuración `c1` cómo la c2. El valor de la sensibilidad es mejor con la configuración `c2` mientras que la exactitud toma mejor valor en la configuración `c1`.

El resultado más ajustado obtenido con este algoritmo es con el método de validación cruzada con `cv=10`, y da una precisión de 91,1 %. Esto supone un 1,9 % menos que utilizando el otro método. En este algoritmo el uso de un método de entrenamiento u otro no supone una gran diferencia.

Tabla 5.6: Árboles de decisión con train_test_split

División	Precisión	Sensibilidad	Exactitud	Valor-F1	ERR	TN	FP	FN	TP	Conf.
70 % 30 %	0.930	0.884	0.888	0.886	0.069	13,629	700	724	5,566	c1
	0.930	0.893	0.879	0.885	0.069	13,560	769	670	5,620	c2
	0.928	0.864	0.896	0.879	0.071	13,699	630	853	5,437	c3

4. Experimentos Realizados con el Algoritmo RF

El algoritmo RF o Bosques aleatorios utiliza una combinación de árboles de decisión que ajusta utilizando diferentes submuestras extraídas del conjunto de datos. Utiliza la media para mejorar la capacidad de precisión y evitar el sobreajuste.

Como se ha comentado en pruebas anteriores y se vuelve a ver en la Tabla 5.7 el mejor resultado obtenido es con el parámetro `cv=10`, tanto para la precisión, como para la sensibilidad y la exactitud.

Tabla 5.7: Bosques aleatorios con validación cruzada

CV	Precisión	Sensibilidad	Exactitud	Valor-F1	ERR	TN	FP	FN	TP	Conf.
5	0.929	0.835	0.929	0.879	0.070	46,342	1,333	3,472	17,582	c1
	0.928	0.834	0.925	0.877	0.071	46,263	1,412	3,479	17,575	c2
	0.928	0.833	0.920	0.874	0.071	46,166	1,509	3,498	17,556	c3
10	0.937	0.854	0.936	0.893	0.062	46,457	1,218	3,062	17,992	c1
	0.936	0.855	0.934	0.892	0.063	46,407	1,268	3,042	18,012	c2
	0.934	0.851	0.930	0.888	0.065	46,339	1,336	3,128	17,976	c3

```
c1: bootstrap=False, max_depth=100, max_features='sqrt',min_samples_split=5
 n_estimators=800
c2: bootstrap=True, max_depth=None, max_features='auto',min_samples_split=2,
n_estimators=100
c3: bootstrap=True, max_depth=20, max_features='auto',min_samples_split=2,
n_estimators=200
```

Durante las pruebas con este algoritmo se ha podido observar que el parámetro `max_depth` que indica la profundidad máxima que puede tener un árbol, sigue sin ser un parámetro que influya drásticamente en la precisión puesto que los resultados obtenidos con la configuraciones `c1` y `c2` son muy similares, y el valor toma valores muy diferentes. En la configuración `c1` `max_depth=100` mientras que en la configuración `c2` `max_depth=None`. También ocurre una situación parecida con el parámetro `n_estimators` que corresponde al número de arboles de decisión utilizados.

En la Tabla 5.8 se puede observar que el mejor resultado obtenido para el algoritmo de RF ha sido con el método `train_test_split` y da una precisión de 95,5 % utilizando tanto la configuración `c1` como la `c2`. El valor de la sensibilidad es mejor con la configuración `c2` mientras que la exactitud toma mejor valor en la configuración `c1`.

El resultado más ajustado obtenido con este algoritmo es con el método de validación cruzada y `cv=10`, y da una precisión de 93,7 %. Esto supone un 1,8 % menos que utilizando el otro método.

Tabla 5.8: Bosques aleatorios con train_test_split

División	Precisión	Sensibilidad	Exactitud	Valor-F1	ERR	TN	FP	FN	TP	Conf.
70 % 30 %	0.955	0.906	0.944	0.924	0.044	13,992	337	587	5,703	c1
	0.955	0.908	0.942	0.924	0.044	13,981	348	577	5,713	c2
	0.953	0.899	0.943	0.920	0.046	13,990	339	630	5,660	c3

5. Experimentos Realizados con el Algoritmo AdaBoost

AdaBoost es un algoritmo de clasificación que comienza ajustando un clasificador al conjunto de datos original y posteriormente ajusta copias adicionales del clasificador (clasificadores débiles) al mismo conjunto de datos pero variando los pesos de las instancias clasificadas incorrectamente para que los siguientes clasificadores den

más importancia a los casos clasificados incorrectamente. Este proceso acaba con la generación de un clasificador robusto.

```
c1: learning_rate=1.0, n_estimators=50
c2: learning_rate=0.1, n_estimators=800
```

Después de hacer algunas pruebas con este algoritmo, se observa que la ejecución es especialmente rápida comparada con otros algoritmos. La configuración con mejores valores ha sido `c2`, en la cuál el parámetro `learning_rate` que reduce la tasa de contribución de cada clasificador, es especialmente bajo, con un valor de 0.1. El número de clasificadores débiles utilizado en la segunda configuración tiene un valor de 800, bastante más grande que el de la configuración `c1`.

En la Tabla 5.10 se puede comprobar que el mejor resultado obtenido para el algoritmo de AdaBoost ha sido con el método `train_test_split` y da una precisión de 89,4 % utilizando tanto la configuración `c1` como la `c2`. El valor de la sensibilidad es mejor con la configuración `c1` mientras que la exactitud toma mejor valor en la configuración `c2`.

Como se puede ver en la Tabla 5.9 el resultado más ajustado obtenido con este algoritmo es con el método de validación cruzada con `cv=10`, y da una precisión de 88,6 %. Esto supone un 0,8 % menos que utilizando el otro método.

Tabla 5.9: AdaBoost con validación cruzada

CV	Precisión	Sensibilidad	Exactitud	Valor-F1	ERR	TN	FP	FN	TP	Conf.
5	0.865	0.722	0.816	0.766	0.134	44,262	3,413	5,848	15,206	c1
	0.869	0.720	0.829	0.770	0.130	44,559	3,116	5,887	15,167	c2
10	0.883	0.764	0.838	0.799	0.116	44,583	3,092	4,949	16,105	c1
	0.886	0.764	0.848	0.803	0.113	44,807	2,868	4,963	16,091	c2

Tabla 5.10: AdaBoost con train_test_split

División	Precisión	Sensibilidad	Exactitud	Valor-F1	ERR	TN	FP	FN	TP	Conf.
70 % 30 %	0.894	0.785	0.855	0.818	0.105	13,494	835	1,350	4,940	c1
	0.894	0.773	0.866	0.816	0.105	13,578	751	1,422	4,868	c2

6. Experimentos Realizados con el Algoritmo MLP

MLP es un algoritmo de clasificación basado en perceptrones multicapa que le permite resolver problemas que no son linealmente separables. Los perceptrones multicapa son redes neuronales artificiales.

```
c1: alpha=1, hidden_layer_sizes=(18,)
c2: alpha=0.00011, hidden_layer_sizes=(100,)
```

En las pruebas realizadas, se ha observado que la configuración `c2` obtiene mejores resultados. Esto puede deberse a que el parámetro `hidden_layer_sizes` que

contiene el número de neuronas en cada capa oculta, es mucho mayor en esta configuración.

Como se puede observar en la Tabla 5.12 el mejor resultado obtenido para el algoritmo MLP ha sido con el método `train_test_split` y da una precisión del 91,8 % utilizando la configuración `c2`.

En la Tabla 5.11 se puede ver que el resultado más ajustado obtenido con este algoritmo es con el método de validación cruzada con `cv=10`, y da una precisión del 90,3 %. Esto supone un 1,5 % menos que utilizando el otro método.

Tabla 5.11: MLP con validación cruzada

CV	Precisión	Sensibilidad	Exactitud	Valor-F1	ERR	TN	FP	FN	TP	Conf.
5	0.824	0.599	0.777	0.676	0.175	44,065	3,610	8,433	12,621	c1
	0.887	0.744	0.872	0.802	0.112	45,375	2,300	5,380	15,674	c2
10	0.828	0.633	0.767	0.693	0.171	43,630	4,045	7,710	13,344	c1
	0.903	0.811	0.867	0.838	0.096	45,068	2,607	3,971	17,083	c2

Tabla 5.12: MLP con train_test_split

División	Precisión	Sensibilidad	Exactitud	Valor-F1	ERR	TN	FP	FN	TP	Conf.
70 % 30 %	0.846	0.622	0.831	0.711	0.153	13,535	794	2,376	3,914	c1
	0.918	0.840	0.886	0.862	0.081	13,649	680	1,002	5,288	c2

7. Experimentos Realizados con el Algoritmo GradientBoosting

GradientBoosting es similar al algoritmo AdaBoost mencionado anteriormente, pero en este caso, no varían los pesos de las instancias clasificadas incorrectamente sino que propone una función de error cuyo gradiente hay que intentar minimizar. Este algoritmo ha dado resultados mucho mejores que el algoritmo AdaBoost.

Como se ha comentado en algoritmos anteriores, en la Tabla 5.13 se puede apreciar que tanto la precisión como la sensibilidad y la exactitud que se obtienen mediante la validación cruzada es mayor cuando el parámetro `cv=10`.

Tabla 5.13: GradientBoosting con validación cruzada

CV	Precisión	Sensibilidad	Exactitud	Valor-F1	ERR	TN	FP	FN	TP	Conf.
5	0.887	0.741	0.870	0.800	0.202	40,779	6,896	7,050	14,004	c1
	0.924	0.840	0.908	0.872	0.075	45,888	1,787	3,356	17,698	c2
10	0.902	0.789	0.880	0.832	0.097	45,413	2,262	4,440	16,614	c1
	0.931	0.853	0.915	0.882	0.068	46,021	1,654	3,093	17,961	c2

```
c1: max_depth=3, min_samples_leaf=1, min_samples_split=2, n_estimators=100
c2: max_depth=5, min_samples_leaf=2, min_samples_split=15, n_estimators=400
```

A lo largo de todas las pruebas, la configuración `c2` ha obtenido mejores resultados, probablemente porque utiliza un número mayor de clasificadores y porque el parámetro `max_depth` es mayor, por lo que permite que el árbol tenga mas profundidad. También el parámetro `min_samples_split`, que es el número de muestras requeridas para poder dividir un nodo interno, toma un valor mayor en la configuración `c2`.

Según se puede ver en la Tabla 5.14 el mejor resultado obtenido para el algoritmo de GradientBoosting ha sido con el método `train_test_split` y da una precisión de 94,5 % utilizando la configuración `c2`.

El resultado más ajustado obtenido con este algoritmo es con el método de validación cruzada con `cv=10`, y da una precisión de 93,1 %. Esto supone un 1,4 % menos que utilizando el otro método.

Tabla 5.14: GradientBoosting con train_test_split

División	Precisión	Sensibilidad	Exactitud	Valor-F1	ERR	TN	FP	FN	TP	Conf.
70 % 30 %	0.914	0.814	0.897	0.853	0.085	13,741	588	1,169	5,121	c1
	0.945	0.890	0.926	0.907	0.054	13,882	447	686	5,604	c2

8. Experimentos Realizados con el Algoritmo LinearSVC

El algoritmo LinearSVC es similar al algoritmo SVC pero con el parámetro `kernel` puesto como lineal, para que tenga mayor flexibilidad a la hora de elegir las penalizaciones y funciones de pérdida. En la documentación de la librería `sklearn` que implementa este clasificador mencionan que este algoritmo funciona mejor que el SVC cuando hay un gran número de muestras como sería este caso. El algoritmo SVC pertenece al conjunto de algoritmos llamado máquinas de soporte vectorial.

```
c1: C=100, max_iter=2000, penalty="l1", dual=False
c2: C=1.0, max_iter=1000, penalty="l2", dual=True
c3: C=0.001, max_iter=1000, penalty='l2', dual=True
```

La configuración que mejor resultado ha dado en general ha sido la `c1`. Esto puede deberse a que el número de iteraciones máximas es mayor que en las otras dos configuraciones. Además el parámetro de regularización `C`, también es bastante mayor en la configuración `c1`.

Según se puede ver en la Tabla 5.16 el mejor resultado obtenido para el algoritmo de LinearSVC ha sido con el método `train_test_split` y da una precisión de 87,4 % utilizando la configuración `c1`.

En la Tabla 5.15 se puede observar que el resultado más ajustado obtenido con este algoritmo es con el método de validación cruzada con `cv=10`, y da una precisión de 85,1 %. Esto supone un 2,3 % menos que utilizando el otro método.

Tabla 5.15: LinearSVC con validación cruzada

CV	Precisión	Sensibilidad	Exactitud	Valor-F1	ERR	TN	FP	FN	TP	Conf.
5	0.844	0.648	0.806	0.718	0.155	44,408	3,267	7,401	13,653	c1
	0.843	0.642	0.808	0.715	0.156	44,478	3,197	7,534	13,520	c2
	0.808	0.591	0.733	0.654	0.191	43,142	4,533	8,605	12,449	c3
10	0.850	0.651	0.825	0.727	0.149	44,774	2,901	7,340	13,714	c1
	0.851	0.647	0.829	0.726	0.148	44,871	2,804	7,418	13,636	c2
	0.815	0.598	0.747	0.664	0.184	43,427	4,248	8,451	12,603	c3

Tabla 5.16: LinearSVC con train_test_split

División	Precisión	Sensibilidad	Exactitud	Valor-F1	ERR	TN	FP	FN	TP	Conf.
70 % 30 %	0.874	0.700	0.860	0.771	0.125	13,614	715	1,881	4,409	c1
	0.860	0.656	0.853	0.741	0.139	13,619	710	2,161	4,129	c2
	0.823	0.606	0.766	0.676	0.176	13,168	1,161	2,473	3,817	c3

9. Experimentos Realizados con el Algoritmo SGDClassifier

El algoritmo SGD resuelve la regresión lógica a través del gradiente estocástico descendente. Para ello, hace pases secuenciales sobre los datos de aprendizaje y, durante cada pase, actualiza las ponderaciones a la vez con el fin de tratar aquellas que son óptimas. Para cada dato de aprendizaje, el sistema calcula el error cometido en la predicción usando una función de pérdida.

```
c1: alpha=0.0001, penalty="l2", max_iter=5, tol=0.001
c2: alpha=1e-05, penalty="l1", max_iter=1000, tol=0.0001
c3: alpha=0.0001, penalty="l2", max_iter=1000, tol=0.001
```

Este algoritmo no se ha analizado en detalle, puesto que ya se tenían algunos resultados con precisiones superiores o iguales al 90 %. Se ha probado con diferentes configuraciones para ver si podía superar a los que ya se tenían pero no ha sido el caso. La mejor configuración obtenida para este algoritmo fue la `c2`.

En la Tabla 5.18 se puede ver que el mejor resultado obtenido para el algoritmo de SGD fue con el método `train_test_split` y da una precisión de 87,8 % utilizando la configuración `c2`.

Como se puede observar en la Tabla 5.17 el resultado más ajustado obtenido con este algoritmo es con el método de validación cruzada con `cv=10`, y da una precisión de 86 %. Esto supone un 1,8 % menos que utilizando el otro método.

Tabla 5.17: SGD con validación cruzada

CV	Precisión	Sensibilidad	Exactitud	Valor-F1	ERR	TN	FP	FN	TP	Conf.
5	0.817	0.623	0.768	0.687	0.182	43,714	3,961	7,933	13,121	c1
	0.846	0.649	0.818	0.723	0.153	44,635	3,040	7,389	13,665	c2
	0.823	0.616	0.763	0.681	0.176	43,652	4,023	4,023	12,987	c3
10	0.837	0.605	0.794	0.686	0.162	44,377	3,298	8,316	12,738	c1
	0.860	0.685	0.835	0.752	0.139	44,842	2,833	6,629	14,425	c2
	0.836	0.604	0.821	0.695	0.163	44,912	2,763	8,324	12,730	c3

Tabla 5.18: SGD con train_test_split

División	Precisión	Sensibilidad	Exactitud	Valor-F1	ERR	TN	FP	FN	TP	Conf.
70 % 30 %	0.843	0.752	0.740	0.746	0.156	12,669	1,660	1,557	4,733	c1
	0.878	0.772	0.820	0.795	0.121	13,264	1,065	1,432	4,858	c2
	0.855	0.698	0.802	0.746	0.144	13,249	1,080	1,897	4,393	c3

10. Experimentos Realizados con el Algoritmo GaussianNB

El algoritmo GaussianNB es un clasificador probabilístico de tipo bayesiano ingenuo, como también lo es BernoulliNB que se ha visto anteriormente. El algoritmo GaussianNB se suele utilizar cuando las características toman valores continuos siguiendo una distribución de Gauss.

`c1: priors=None, var_smoothing=1e-09`

Con este algoritmo sólo se han hecho 3 pruebas de las cuales se ha documentado sólo una porque de las otras dos se han obtenido resultados casi idénticos. Puesto que ya se tenían algoritmos con precisiones más altas que las que daba este algoritmo, no se han realizado más pruebas. Quizás por la falta de pruebas, es curioso observar que la precisión más alta no se ha conseguido mediante el método `train_test_split` como se puede observar en la Tabla 5.20, sino mediante validación cruzada con `cv=10` como se puede ver en la Tabla 5.19. Es el único algoritmo que se ha analizado y en el que pasa esto.

El resultado más ajustado que coincide con el mejor resultado ha sido extraído con el método de validación cruzada con `cv=10`. Da una precisión de 80,9 %.

Tabla 5.19: GaussianNB con validación cruzada

CV	Precisión	Sensibilidad	Exactitud	Valor-F1	ERR	TN	FP	FN	TP	Conf.
5	0.807	0.560	0.748	0.640	0.192	43,703	3,972	9,247	11,807	c1
10	0.816	0.535	0.797	0.640	0.183	44,808	2,867	9,778	11,276	c1

Tabla 5.20: GaussianNB con train_test_split

División	Precisión	Sensibilidad	Exactitud	Valor-F1	ERR	TN	FP	FN	TP	Conf.
70 % 30 %	0.809	0.425	0.896	0.576	0.190	14,021	308	3,616	2,674	c1

11. Experimentos Realizados con el Algoritmo Análisis Discriminante Lineal

El algoritmo Análisis Discriminante Lineal estima la probabilidad de que una observación, dado un determinado valor de los predictores, pertenezca a cada una de las clases de la variable cualitativa. Finalmente se asigna la observación a la clase para la que la probabilidad predicha sea mayor.

Con este algoritmo apenas se han hecho pruebas puesto que ya se tenían algoritmos con precisiones más altas. `c1: priors=None, var_smoothing=1e-09`

Se puede observar en la Tabla 5.22 que el mejor resultado obtenido para el algoritmo Análisis Discriminante Lineal ha sido con el método `train_test_split` y da una precisión del 85,5 %.

En la Tabla 5.21 se puede ver que el resultado más ajustado ha sido extraído con el método de validación cruzada con `cv=10`. Da una precisión del 84,6 %. Esto es un 1,1 % menos que utilizando el otro método.

Tabla 5.21: Análisis Discriminante Lineal con validación cruzada

CV	Precisión	Sensibilidad	Exactitud	Valor-F1	ERR	TN	FP	FN	TP	Conf.
5	0.839	0.627	0.804	0.704	0.160	44,453	3,222	7,835	13,219	c1
10	0.846	0.632	0.826	0.716	0.153	44,976	2,799	7,744	13,310	c1

Tabla 5.22: Análisis Discriminante Lineal con train_test_split

División	Precisión	Sensibilidad	Exactitud	Valor-F1	ERR	TN	FP	FN	TP	Conf.
70 % 30 %	0.855	0.644	0.846	0.731	0.144	13,593	736	2,238	4,052	c1

12. Experimentos Realizados con el Algoritmo Redes Neuronales

Las redes neuronales obtienen el resultado mediante la asignación de pesos a cada neurona y el uso de funciones de activación que tiene lugar entre la capa de entrada (neuronas de entrada) y las capas ocultas. El correcto ajuste de los pesos se consigue entrenando la red neuronal correctamente. Se han conseguido algunos de los mejores parámetros mediante la función `GridSearchCV` y se han hecho distintas pruebas variando el número de capas y el número de `epochs`. El valor `epochs` corresponde al número de iteraciones que se llevan a cabo en los datos de entrenamiento. Con dos capas, los resultados no eran nada óptimos y por eso no están incluidos en la tabla. Por otro lado, la configuración más óptima se ha encontrado con la configuración `c1` que utiliza tres capas. El uso de una cuarta capa no supone una mejora apreciable en los resultados pero si afecta notablemente al tiempo de ejecución. Sobre el parámetro

`epochs` se ha observado que aumenta la precisión cuando supera el valor de 70. Con un valor de 130 se ha encontrado el mejor resultado. Cuando se ha aumentado el valor a 150 no ha supuesto ninguna mejoría.

```
c1:layers=(50,30,15), activation='relu', batch_size=256, epochs=130
c2:layers=(70,50,30,15), activation='relu', batch_size=256, epochs=150
```

En la Tabla 5.23 se puede observar igual que en los algoritmos anteriores, que las redes neuronales también dan mejor resultado con `cv = 10`. En concreto con la configuración `c2`, que aumenta en un 0.002 % el resultado que se obtiene con la configuración `c1`.

Tabla 5.23: Redes Neuronales con validación cruzada

CV	Precisión	Sensibilidad	Exactitud	Valor-F1	ERR	TN	FP	FN	TP	Conf.
5	0.704	0.277	0.504	0.357	0.295	41,953	5,722	15,219	5,835	c1
	0.696	0.300	0.545	0.386	0.303	42,390	5,285	14,719	6,335	c2
10	0.866	0.676	0.855	0.755	0.133	45,277	2,398	6,815	14,239	c1
	0.868	0.663	0.857	0.747	0.131	45,345	2,330	7,078	13,976	c2

Como en otros modelos, se sigue observando la diferencia de precisión entre los valores calculados con validación cruzada y los valores calculados usando la función `train_test_split`.

Como se puede observar en la Tabla 5.24 el mejor resultado obtenido para las redes neuronales ha sido con el método `train_test_split` y da una precisión de 92,2 % utilizando tanto la configuración `c1` como la `c2`. Se puede observar que aunque las configuraciones den la misma precisión, la sensibilidad es mayor en la configuración `c1`.

El resultado más ajustado obtenido con este algoritmo es con el método de validación cruzada con `cv=10`, y da una precisión de 86,8 %. Esto supone un 5,4 % menos que utilizando el otro método.

Tabla 5.24: Redes Neuronales con train_test_split

División	Precisión	Sensibilidad	Exactitud	Valor-F1	ERR	TN	FP	FN	TP	Conf.
70 % 30 %	0.922	0.838	0.900	0.867	0.077	13,749	580	1,014	5,276	c1
	0.922	0.827	0.911	0.077	0.867	13,824	505	1,086	5,204	c2

5.2.1. Pruebas con Diferentes Conjuntos de Características

Durante esta etapa de experimentación, se ha llevado a cabo una serie de pruebas con diferentes conjuntos de características. En concreto, se han separado las características extraídas de la URL de las extraídas del DOM y se han evaluado los resultados individualmente. Los resultados obtenidos por separado no han sido tan buenos como los obtenidos combinando ambas características. En esta sección se incluye la precisión obtenida para cada uno de los 12 algoritmos probados, tanto aplicando validación cruzada como aplicando la división mediante `train_test_split`. Para el resultado de validación

cruzada, se toma únicamente `cv=10` puesto que se ha observado en las pruebas anteriores con todas las características, que da mejor resultado que `cv=5`.

La Tabla 5.25 hace referencia a los resultados obtenidos exclusivamente de la extracción de características de la URL aplicando validación cruzada. Se puede observar que la precisión no llega a superar el 80 %, lo que supone un porcentaje bastante bajo en comparación con el obtenido con ambas características. El algoritmo KNN es el que peor precisión obtiene, incluso siendo el tercer algoritmo que mejor resultado da cuando se juntan ambas extracciones de características.

Tabla 5.25: Resultados de los algoritmos con características de la URL con validación cruzada

Algoritmo	Precisión con validación cruzada
Gradient Boosting	0.795
RF	0.794
Árboles de decisión	0.791
Adaboost	0.788
MLP	0.786
Linear SVC	0.732
Análisis discriminante lineal	0.729
GaussianNB	0.713
SGD	0.708
BernoulliNB	0.698
Redes neuronales	0.691
KNN	0.567

La Tabla 5.26 hace referencia a los resultados obtenidos exclusivamente de la extracción de características de la URL aplicando la división mediante `train_test_split`. Como en casos anteriores, la precisión obtenida mediante este método es superior a la obtenida aplicando validación cruzada. En estas pruebas, el algoritmo KNN ha obtenido un resultado notablemente mayor que el obtenido mediante validación cruzada.

Tabla 5.26: Resultados de los algoritmos con características de la URL con train_test_split

Algoritmo	Precisión con train_test_split
RF	0.804
Árboles de decisión	0.803
Gradient Boosting	0.802
Redes neuronales	0.789
MLP	0.788
KNN	0.774
Adaboost	0.767
Linear SVC	0.735
Análisis discriminante lineal	0.732
SGD	0.715
BernoulliNB	0.711
GaussianNB	0.708

La Tabla 5.27 hace referencia a los resultados obtenidos exclusivamente de la extracción

de características del DOM aplicando validación cruzada. Se puede observar que la precisión llega a superar el 90 % para los algoritmos RF y Gradient Boosting. RF es el algoritmo que mejor resultado da al utilizar solo las características del DOM, con una diferencia aproximadamente de 2 % menos que el resultado obtenido con validación cruzada para ambas características. El algoritmo GaussianNB es el que peor precisión obtiene, con un porcentaje muy bajo y aunque al usar ambas características no obtenga un porcentaje muy alto sigue existiendo una inmensa diferencia entre los dos porcentajes.

Tabla 5.27: Resultados de los algoritmos con características del DOM con validación cruzada

Algoritmo	Precisión con validación cruzada
RF	0.914
Gradient Boosting	0.901
KNN	0.895
Árboles de decisión	0.892
MLP	0.840
AdaBoost	0.839
SGD	0.813
LinearSVC	0.807
Análisis discriminante lineal	0.792
BernoulliNB	0.769
Redes Neuronales	0.697
GaussianNB	0.467

La Tabla 5.28 hace referencia a los resultados obtenidos exclusivamente de la extracción de características del DOM aplicando la división mediante `train_test_split`. Como en casos anteriores, la precisión obtenida mediante este método es superior a la obtenida aplicando validación cruzada. En estas pruebas, el algoritmo RF ha obtenido un resultado muy parecido al obtenido usando ambas características. Como en el caso anterior el algoritmo GaussianNB ha obtenido un porcentaje muy bajo siendo el último en la lista.

Tabla 5.28: Resultados de los algoritmos con características de la DOM con train_test_split

Algoritmo	Precisión con train_test_split
RF	0.936
Gradient Boosting	0.922
KNN	0.918
Árboles de decisión	0.914
Redes neuronales	0.865
MLP	0.858
AdaBoost	0.848
Linear SVC	0.808
SGD	0.805
Linear Discriminant Analysis	0.802
BernoulliNB	0.777
GaussianNB	0.493

5.2.2. Validation Curve

Como se ha indicado en apartados anteriores, el uso de `GridSearchCV` permite ajustar los mejores hiperparámetros para un modelo y de esta forma lograr maximizar su precisión. Sin embargo, esta maximización puede provocar que la puntuación obtenida esté sesgada y no obtener el resultado esperado al tratar de conseguir la misma precisión para otro conjunto de datos.

Para evaluar el comportamiento de un parámetro en la puntuación de entrenamiento y prueba es muy útil hacerlo mediante la curva de validación o `validation curve`, que permite comprobar cómo se comporta el modelo con cada parámetro y si se produce sobreajuste (*overfitting*) o subajuste (*underfitting*) en los valores obtenidos para cada parámetro.

Puesto que la mejor estimación se obtuvo con el modelo RF el análisis de la curva de validación se realizará sobre este modelo y cada parámetro suyo ajustado.

La Figura 5.1 muestra el comportamiento del modelo para el parámetro `n_estimators`, que permanece casi constante para el rango de valores mostrado. Esto justifica que en las pruebas realizadas las configuraciones `c1` y `c2`, aunque se tomaran valores de `n_estimators` muy diferentes, (entre 100 y 800) el resultado fuera muy similar. Este parámetro representa el número total de árboles y cuanto mayor sea su valor mejor aprende el modelo, sin embargo, esto supone también una ralentización en el proceso de aprendizaje del modelo.

El parámetro `max_depth` indica la profundidad máxima que tiene cada árbol de decisión en el modelo. En la Figura 5.2 se puede observar que para un valor mayor que 30 y menor que 40 la puntuación de precisión apenas se incrementa, esto quiere decir que no es necesario un valor de profundidad tan alto para lograr una mayor precisión en el modelo. Esto se ve reflejado en las pruebas ya que la configuración `c3` que tiene `max_depth=20` obtenía un peor resultado en comparación con las otras.

El parámetro `min_samples_split` representa el número mínimo de características que debe haber en un nodo interno en cada árbol de decisión para poder hacer la partición. Tal y como se muestra en la Figura 5.3, incrementar el valor de este parámetro apenas mejora la precisión del modelo, incluso disminuye ligeramente la puntuación para el conjunto de entrenamiento. Todas las configuraciones probadas utilizan valores bajos (entre 2 y 5) lo cual confirma que con estos valores se logra una alta precisión.

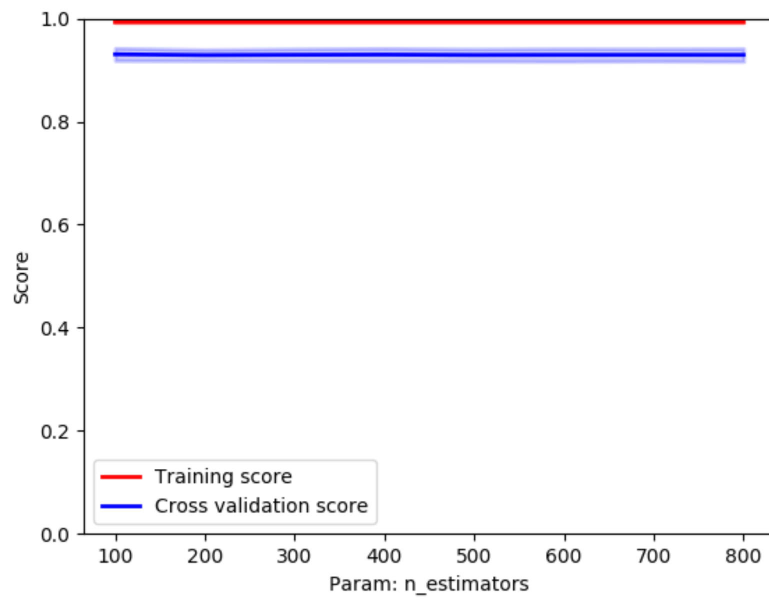


Figura 5.1: Curva de Validación para el parámetro “n_estimators”

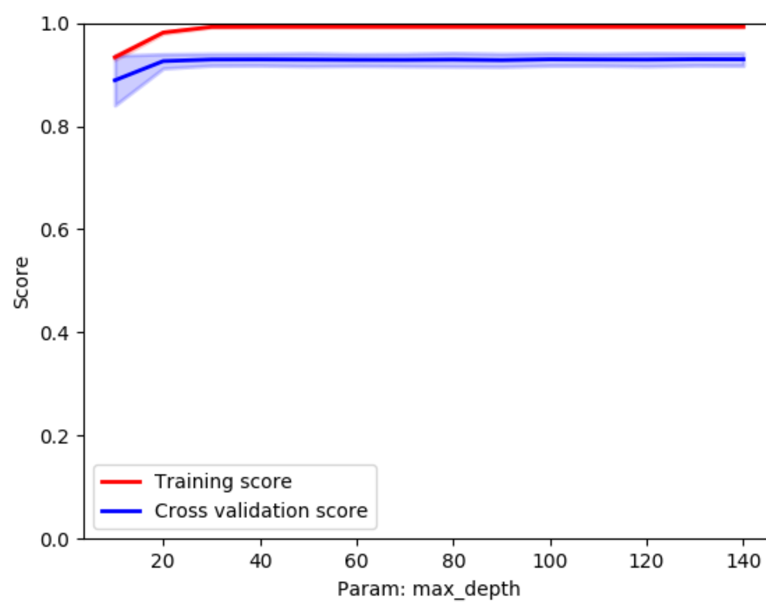


Figura 5.2: Curva de Validación para el parámetro “max_depth”

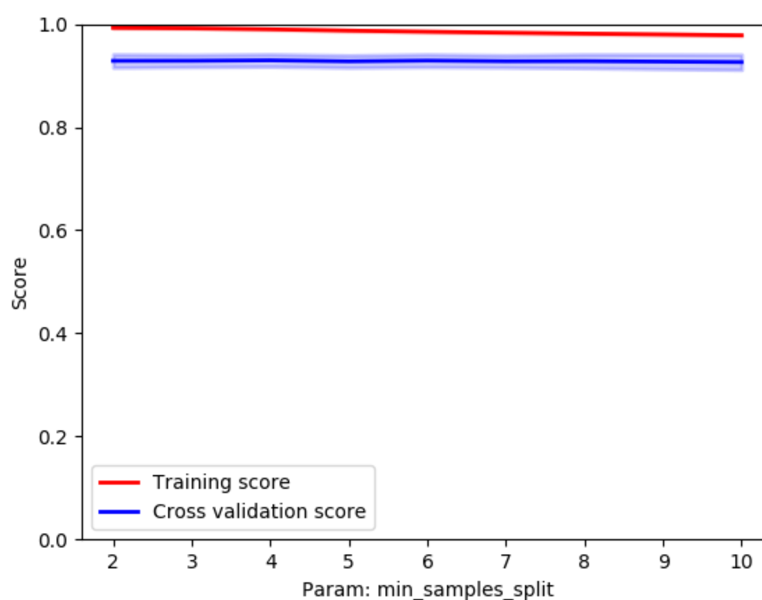


Figura 5.3: Curva de Validación para el parámetro “min_samples_split”

5.3. Evaluación de las Mejores Características

La evaluación de características es una parte importante de cualquier proceso de aprendizaje automático. Permite encontrar las características que aportan mayor valor al modelo al generar un alto impacto en la variable de salida. Si se agregan características irrelevantes al modelo, pueden generar ruido o aumentar innecesariamente el tiempo de ejecución.

En esta sección se han evaluado las 39 características implementadas. Como el algoritmo que ha obtenido mejor resultado ha sido el RF se ha utilizado su modelo para extraer las mejores características con el método `feature_importances_`. Este método calcula la correlación entre la variable de salida que en este caso sería *phishing* y cada una de las características implementadas. Las que tienen el coeficiente de correlación mayor son las que más inciden en la variable de salida.

Las 5 características más importantes han sido `sensitive_words`, `number_of_all_links_extract`, `number_of_urlbrandname_inhtmlcode`, `null_anchor_links_ratio_in_website` y `dots_in_url`.

La Figura 5.4 muestra las 20 mejores características ordenadas de mayor a menor importancia. Durante la etapa de experimentos, se ha evaluado el algoritmo RF con estas primeras 20 características para comprobar si al usar únicamente estas, se podía obtener un resultado mejor o parecido puesto que se habrían descartado características que aportasen ruido al modelo. Después de las pruebas los resultados que han salido con estas 20 características han sido algo peores que los obtenidos con las 39 características. Por lo tanto, las características que no están entre las 20 mejores, aunque no sean tan importantes, tampoco aportan ruido excesivo al modelo.

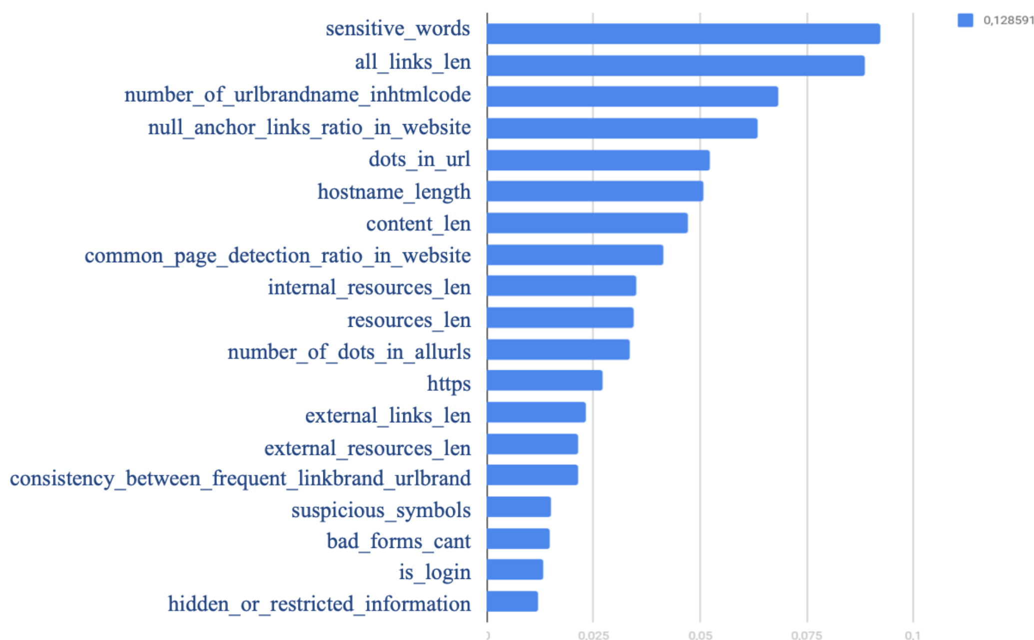


Figura 5.4: 20 mejores características por orden de importancia

5.4. Resultados Obtenidos

Una vez realizados todos los experimentos explicados en la sección anterior, se ha observado que las métricas cambian considerablemente dependiendo de si han sido extraídas utilizando el método convencional de dividir el conjunto de datos en un 70 % de datos de entrenamiento y un 30 % de datos de prueba o si han sido extraídas aplicando validación cruzada con `cv=5` y `cv=10`. Por lo tanto para los resultados, se han aportado las métricas obtenidas por ambos métodos dado que muchos de los trabajos con los que se comparan los resultados, no especifican cómo han entrenado el modelo.

La Tabla 5.29 muestra los mejores resultados obtenidos aplicando `train_text_split`.

Tabla 5.29: Mejores resultados obtenidos con train_text_split

Algoritmo	Precisión	Sensibilidad	Exactitud	Valor-F1	ERR
Random Forest	0.955	0.908	0.942	0.924	0.044
GradiantBoost	0.945	0.890	0.926	0.907	0.054
KNN	0.934	0.843	0.884	0.863	0.081
Árboles de decisión	0.930	0.893	0.879	0.885	0.069
Redes Neuronales	0.922	0.838	0.900	0.867	0.077

La Tabla 5.30 muestra los mejores resultados obtenidos aplicando validación cruzada, en concreto con `cv=10`.

Tabla 5.30: Mejores resultados obtenidos con validación cruzada

Algoritmo	Precisión	Sensibilidad	Exactitud	Valor-F1	ERR
Random Forest	0.937	0.854	0.936	0.893	0.062
GradiantBoost	0.931	0.853	0.915	0.882	0.068
Árboles de decisión	0.911	0.832	0.869	0.850	0.088
KNN	0.908	0.814	0.878	0.858	0.091
Redes Neuronales	0.868	0.663	0.857	0.747	0.131

Por lo tanto se puede concluir que el mejor resultado se ha obtenido con el algoritmo RF. Por lo que se ha utilizado RF para generar el modelo final.

La Tabla 5.31 muestra la comparación de los resultados obtenidos en este trabajo con los obtenidos en los estudios mencionados en el Capítulo 3.

Tabla 5.31: Comparación de resultados con otros trabajos

Trabajo	Precisión	Sensibilidad	Exactitud	Valor-F1	ERR	TPR	FPR
CANTINA [ZHC07]	-	-	-	-	-	89 %	1 %
CANTINA +[XHRC11]	-	-	-	-	-	92 %	0.4 %
[SBDD19]	97.98 %	97.0 %	99.0 %	97.9 %	-	-	-
[RP19]	-	-	-	-	-	99.55 %	0.45 %
[LYC+19]	97.30 %	-	-	-	-	-	-
[HHF+11]	-	-	-	-	-	97.33 %	1.45 %
[BAS19]	92.80 %	-	-	-	-	-	-
Este trabajo	95.5 %	90.8 %	94.2 %	92.4 %	4.4 %	90.6 %	2.35 %

Capítulo 6

Contribuciones Individuales

La idea de realizar este Trabajo de Fin de Grado, surgió por parte de nuestros tutores académicos Luis Javier García Villalba y Esteban Alejandro Armas Vega. Los tres alumnos que les habíamos elegido como directores de nuestro trabajo, queríamos aprender e investigar sobre el mismo tema, la detección de amenazas en la red. El poder realizar este trabajo entre tres personas nos ha permitido abordar una propuesta mucho más ambiciosa, y poder repartirnos el trabajo equitativamente para llevar a cabo una gran cantidad de pruebas y experimentos. Dentro de la detección de amenazas, habían muchos campos en los que poder investigar. En concreto estuvimos leyendo e investigando sobre los ataques de secuestro de datos o *ransomware* y los ataques de *phishing*. Finalmente juntos con nuestros tutores decidimos enfocarnos en los ataques de *phishing* puesto que al haber ocurrido recientemente el ataque *ransomware* conocido como *WannaCry* que tuvo un gran impacto, y otros ataques similares, habían muchas mas investigaciones sobre este tema que sobre *phishing*. Nuestra intención era aportar un nuevo enfoque a la detección de ataques de *phishing* y en la medida de lo posible incrementar el resultado obtenido en algunos de los trabajos que habíamos estado estudiando.

Los trabajos en grupo exigen constante comunicación entre los miembros que lo forman. En nuestro caso, teníamos reuniones con nuestros tutores semanalmente y la comunicación entre nosotros tres era constante y fluida. Esto nos ha permitido poder aprovechar el tiempo y ayudarnos entre nosotros constantemente. A continuación se explica con más detalle la implicación de cada miembro del grupo en las tareas que se han llevado a cabo para realizar este trabajo y los conocimientos adquiridos a lo largo de todo el proceso.

6.1. Héctor Hugo Coronado Huamán

A la hora de comenzar este trabajo, el primer paso fue buscar toda la información relacionada con la detección de amenazas en la red. Puesto que se necesitaba encontrar la información más veraz y completa posible utilicé el buscador de *Google Scholar* y la biblioteca CISNE de la Universidad Complutense de Madrid para buscar artículos de revistas de investigación y trabajos relacionados con la detección de amenazas en la red. Leyendo estos artículos aprendí varios conceptos y definiciones sobre los distintos tipos de ataques que se utilizan actualmente así como las técnicas implementadas para su prevención y detección. Juntos con nuestros tutores, decidimos centrarnos en los ataques de *phishing*, y puesto que no existían muchos trabajos relacionados, comencé a investigar a fondo este tipo de amenaza.

Posteriormente, decidimos que la detección de amenazas la llevaríamos a cabo utilizando algoritmos de aprendizaje automático. Muchos de los trabajos que había visto, implementaban estos algoritmos utilizando el lenguaje de programación *Python*. Por recomendación de nuestros tutores y para poder implementar el código de forma más óptima utilizamos el entorno de desarrollo *Pycharm* en conjunción con el servicio web de control de versiones *Gitlab* para poder trabajar de forma más cómoda y eficiente a la hora de diseñar el código y poder repartirnos las tareas con mis compañeras. En la carrera había visto múltiples lenguajes de programación pero no había utilizado *Python* en ningún momento. Por tanto, durante las primeras semanas de desarrollo del trabajo estuve leyendo información acerca de su funcionamiento (conceptos como la indentación, diferencias con los demás lenguajes), listas intencionales, etc. También aprendí acerca de algunas de las herramientas más utilizadas en el campo del aprendizaje automático y del *Web Scraping* como son *Scikit-learn*, *Matplotlib*, *Pandas*, *BeautifulSoup*, entre otras. Al mismo tiempo, nuestros tutores nos recomendaron tomar algunas clases sobre los algoritmos de aprendizaje automático más conocidos. Esto nos permitió conocer los fundamentos y características más importantes de cada algoritmo así como sus ventajas e inconvenientes.

Durante las reuniones semanales que teníamos nuestros tutores también nos guiaron en el diseño que debería tener el sistema de detección y cómo organizar cada una de sus fases. Una vez tuvimos un concepto claro del trabajo que íbamos a desarrollar nos repartimos el trabajo. Para poder empezar con el trabajo necesitábamos extraer las características que habíamos leído en algunos de los artículos. Para ello volví a leer todos los artículos y trabajos encontrados y seleccioné los que me parecían más interesantes. En total fueron siete trabajos de los cuales seleccioné las características de la URL y del DOM más relevantes. Más adelante, mientras mis compañeras investigaban la parte del desarrollo de la API, empecé a implementar la extracción de características de la URL que había encontrado. Para ello utilicé las bibliotecas de *Python* `re` y `urllib3`. La primera me sirvió para poder comprobar si aparecía algún patrón en la dirección web utilizando las expresiones regulares, mientras que la segunda permitía obtener las diferentes partes de una URL. En total fueron 12 características. Una vez habíamos implementado todas las características tanto de la URL como del DOM, seleccionamos los algoritmos que queríamos probar.

Para cada algoritmo que habíamos seleccionado, busqué los rangos de los hiperparámetros para introducirlos posteriormente en la función `GridSearchCV` y obtener los hiperparámetros más óptimos para nuestro conjunto de datos de entrenamiento. Paralelamente, mientras mi compañera preprocesaba el conjunto de datos, corregíamos errores de las características extraídas que no se podían ver hasta no tener el archivo con todas las características extraídas. Una vez tuvimos todas las características corregidas,

empezamos a investigar cómo funcionaban las APIs. Mediante algunos artículos y tutoriales aprendimos cómo hacer una API REST sencilla utilizando *SQLAlchemy*, *Connexion* y *Flask*.

Una vez teníamos la base de la API hecha, y las características corregidas, empezamos a extraer las características de diferentes conjuntos de datos. Estuve haciendo pruebas con el conjunto de datos de 96,000 URLs. Una vez tuvimos todas las pruebas, extrajimos las características del conjunto de datos final. Tuvimos que añadir una configuración para la cabecera de la petición *GET* hecha para cada URL puesto que, de lo contrario, muchas de las peticiones podrían ser rechazadas. Para ello utilizamos un archivo con una lista de *User Agents*, configuramos el número de intentos para reintentar una conexión (*max_retries*) y el tiempo para cancelarla (*timeout*). Asimismo implementé la optimización de la extracción mediante el uso de las librerías *ThreadPoolExecutor* y *ProcessPoolExecutor*. Estas nos permitieron ejecutar de forma concurrente la obtención de características y a la vez habían conseguido reducir el tiempo de ejecución considerablemente. Posteriormente, procedimos conjuntamente a probar los algoritmos que habíamos seleccionado en un principio, con los hiperparámetros óptimos encontrados. De cada algoritmo, conseguimos las métricas y seleccionamos aquellos que habían obtenido los mejores resultados utilizando el módulo `sklearn.metrics`.

Para evaluar los resultados del algoritmo que mejor puntuación ha obtenido, busqué toda la información posible acerca de la curva de validación para después aplicarla a nuestro proyecto puesto que permite comprobar que los parámetros obtenidos mediante `GridSearchCV` son los mas óptimos posibles, es decir, conseguir con el valor menos costoso computacionalmente, el mejor resultado. También en esta etapa de experimentación, investigué acerca de las características que más aportaban al modelo utilizando el algoritmo RF y comprobando gráficamente los resultados. Una vez obtenida y organizada toda la información de las diferentes pruebas y experimentos, realizamos la comparación de los resultados finales y procedimos a revisar y limpiar el código de nuestra implementación incluyendo comentarios y facilitando su lectura.

Seguidamente decidimos desarrollar una aplicación sencilla que pudiera ser accesible desde una ruta diferente a la API utilizando las anotaciones de *Connexion* para construir el *routing* para cada una de ellas. La idea de diseñar una aplicación fue que a través de ella se pudiera acceder a la API de forma más intuitiva para el usuario. Utilizando las librerías *SQLAlchemy*, *Connexion* y *Flask* implementamos su funcionamiento y también preparamos un archivo con la configuración básica necesaria para su posterior despliegue a un contenedor de *Docker*. En esta última parte utilicé los conocimientos que había adquirido en la asignatura Aplicaciones Web así como también la experiencia laboral adquirida con *Javascript* y HTML.

Con el proyecto casi finalizado y, tras reunir y evaluar toda la información obtenida de los experimentos, comenzamos a escribir la memoria. Para ello utilizamos *LaTeX*, una herramienta para preparar documentos muy utilizada en la redacción de artículos y en el ámbito de la investigación. Había escuchado hablar de este sistema pero hasta el momento de redactar esta memoria no lo había utilizado antes. Es por ello que nuestros tutores nos ofrecieron la posibilidad de aprender los conceptos más importantes sobre *LaTeX*, como crear tablas, añadir figuras o citar los documentos, artículos y páginas web. Además, nos enseñaron cuál debe ser la estructura general que debe tener un documento de investigación. Con los conocimientos adquiridos mediante esas clases y los múltiples recursos en línea que hay al respecto, mis compañeras y yo desarrollamos la memoria con relativa facilidad.

6.2. Adina Han

A lo largo del desarrollo de este trabajo, hemos decidido trabajar tanto de forma individual como de forma conjunta. La primera tarea individual ha sido la de investigar y buscar artículos académicos relacionados con las amenazas en la red, la manera de detectarlas y sus posibles contra-medidas. Leí distintos artículos relacionados con ataques de *phishing* y de *ransomware*, artículos encontrados utilizando el buscador de *Google Scholar*. La biblioteca de la Universidad Complutense ha sido otra de las fuentes utilizadas para acceder a diversas revistas y artículos, ya que permite el acceso gratuito a estos artículos. *Google Scholar* ha sido un recurso muy valioso que utilizaré con certeza en el futuro. Otro recurso al que hemos sacado provecho para la investigación han sido las clases de fundamentos de Redes Neuronales y Aprendizaje Profundo, que nos recomendaron nuestros tutores, clases impartidas en el recinto de nuestra facultad. Durante la investigación hemos decidido centrarnos en los ataques de *phishing*.

Uno de los métodos que más se utilizan para detectar ataques de *phishing*, es la generación de características heurísticas que mediante algoritmos de aprendizaje automático puedan detectar si una página web es *phishing* o no. Al cursar en segundo de carrera la asignatura optativa Minería de datos y en tercero de carrera, la asignatura de Inteligencia Artificial, adquirí los fundamentos que me han permitido entender e implementar con mayor facilidad, todo lo que he estado leyendo en los diferentes estudios sobre aprendizaje automático. Mediante diversos vídeos y tutoriales profundicé en poco tiempo el lenguaje *Python* que al ser similar a otros lenguajes de programación que ya he visto en la carrera, no fue muy difícil de dominar. Este lenguaje me ha permitido trabajar con todo tipo de estructuras de datos sin ninguna dificultad. Además, he utilizado múltiples librerías que incluye este lenguaje, como serían *Matplotlib*, *Pathlib*, *Scikit-learn*, *Numpy*, *Pandas*, *TensorFlow* o *Keras*, librerías que ofrecen multitud de facilidades y funciones predefinidas que aligeraba mi trabajo.

El objetivo de nuestro sistema de detección era poder hacer un sistema híbrido que combinase algoritmos de aprendizaje automático con dos listas negras, una almacenada localmente y la otra accesible mediante una API externa. Para ello mi compañera creó una base de datos local que correspondería a nuestra lista negra y yo evalué diversas opciones gratuitas que hay en el mercado para una API externa de lista negra. La mejor opción por la que me he decidido ha sido *LookupAPI* llevada a cabo por *Google Safe Browsing* porque es la API que contiene el mayor número de páginas web fraudulentas en la actualidad.

Al terminar la tarea de selección de características que estaba asignada a mi compañero, he podido proceder a la implementación de las características del DOM que me correspondían. De un total de 28 características del DOM que mi compañero había encontrado he implementado 14 y mi compañera las 14 restantes. Una vez terminamos de implementar todas las características, seleccionamos los 12 algoritmos de clasificación con los que llevaríamos a cabo las pruebas. Mientras mi compañera hacía algunas pruebas con el conjunto de datos que generaban las características extraídas, para cada algoritmo que habíamos seleccionado, evalué los distintos valores y combinaciones que podían tomar los hiperparámetros para introducirlos después en la función `GridSearchCV` y obtener los hiperparámetros más óptimos para nuestro conjunto de datos de entrenamiento.

Posteriormente al analizar en profundidad los datos resultantes y repasar la implementación de las características se observaban algunos valores incorrectos para algunas de ellas y por lo tanto se hicieron las cambios pertinentes para solventar esto. Una vez comprobado el correcto funcionamiento de la implementación de las características, desarrollamos parte de la API. Mediante algunos artículos y tutoriales aprendimos cómo

hacer una API REST utilizando *SQLAlchemy*, *Connexion* y *Flask*.

Con las características ya corregidas, empezamos a extraer las características de diferentes conjuntos de datos. Extraje las características del tercer conjunto de datos, que contenía 130,000 URLs y que sería el conjunto de datos con el que nos quedaríamos finalmente. También realicé distintas pruebas con diversos conjuntos de características. La extracción de características y las pruebas realizadas, dado el conjunto de datos muy amplio requerían un amplio plazo de tiempo. Esta ha sido la causa por la que hemos buscado maneras de extraer las características eficientemente y el uso de las librerías *ThreadPoolExecutor* y *ProcessPoolExecutor* nos ha facilitado esta tarea.

El siguiente paso fue aplicar los mejores hiperparámetros que se han conseguido a los algoritmos seleccionados con anterioridad. De cada algoritmo, sacamos las métricas y seleccionamos los que mejores resultados habían obtenido. También llevé a cabo experimentos para cada uno de los 12 algoritmos seleccionados, con las características extraídas únicamente de la URL y he documentado estos resultados. Todas las pruebas que he realizado se han unificado con las hechas por mis compañeros para afinar lo mejor posible el modelo y poder tener un modelo fiable con la mayor tasa de aciertos posible y la menor tasa de fallos.

Una vez obtenida y organizada toda la información de las diferentes pruebas y experimentos, realizamos la comparación de los resultados finales para poder extraer las conclusiones sobre el modelo desarrollado. Además extrajimos las mejores características que generaba el modelo. Las mejores características las obtuvimos en base al coeficiente de correlación quedándonos con las que tuvieran mayor impacto en la variable de salida, en este caso, el campo *phishing*. También procedimos a revisar y limpiar el código de nuestra implementación incluyendo comentarios facilitando así su lectura. Además, revisé todos los experimentos que habíamos realizado aumentando el contenido de algunos de ellos incluyendo pruebas realizadas con otros parámetros y métricas que faltaban por incluir. El último paso que hemos tenido que dar para poder completar este trabajo ha sido el desarrollo de la memoria, que se ha realizado en conjunto.

Para que se pudiese acceder a la API de forma mas intuitiva, desarrollamos una Aplicación Web sencilla. Esta aplicación se llevó a cabo utilizando *Javascript* y HTML. Cómo pertenezco a la rama de tecnología de la computación, no tuve la ocasión de cursar la asignaturas de Aplicaciones Web durante la carrera. Por lo tanto, me informé acerca de la creación de páginas web mediante diversos cursos que encontré en línea y apuntes que me dejaron mis compañeros. *LaTeX* ha sido la herramienta que mis compañeros y yo hemos utilizado para escribir la memoria. No la había utilizado con anterioridad pero ha sido una oportunidad de adquirir nuevos conocimientos y con la ayuda de nuestros tutores que nos han proporcionado algunas clases de *LaTeX* y con la información encontrada en linea he llevado a cabo esta tarea con éxito.

Durante estos últimos meses de desarrollo de este trabajo, he aprendido la importancia del trabajo en equipo y de la planificación de la tareas. Al trabajar en equipo mis compañeros y yo hemos logrado generar grandes ideas para la finalización de este trabajo y hemos encontrado soluciones creativas a los problemas a los que nos hemos enfrentado durante la realización de este trabajo. La fecha limite que teníamos para cada tarea me ha ayudado a centrarme en lo importante en cada momento. Además, he adquirido conocimientos muy valiosos y he afianzado los que ya tenía y esto me servirá a la hora de empezar en el mundo laboral.

6.3. Laura Sanz García

Para llevar a cabo este trabajo, desde un principio nos íbamos dividiendo las tareas de tal forma que pudiésemos aprender un poco de todo a lo largo del trabajo. La primera tarea que tuve que hacer nada mas empezar, fue investigar acerca de los diferentes tipos de amenazas que se utilizan en la actualidad y que métodos se implementan para detectarlas. En particular, leí diversos artículos relacionados con ataques de *phishing* y de *ransomware*. Aprendí a buscar artículos y estudios utilizando el buscador de *Google Scholar* y diversas bases de datos. Además, la Universidad Complutense permite el acceso gratuito a diversas revistas de artículos. Esto me pareció bastante útil y lo utilizaré en un futuro cada vez que quiera buscar información en profundidad sobre cualquier tema. Una vez terminada esta etapa de lectura y documentación, como se menciona en la introducción de este capítulo, decidimos centrar nuestro trabajo en la detección de ataques de *phishing*.

Para detectar ataques de *phishing*, los métodos más comunes utilizan listas blancas, listas negras o algoritmos de aprendizaje automático. Había oído hablar del aprendizaje automático pero no lo había estudiado en ninguna asignatura. Además la mayoría de las implementaciones se hacían utilizando *Python* y tampoco lo había visto con anterioridad. Afortunadamente, en la carrera he estudiado diversos lenguajes de programación, como *Java* o *C++*, que han hecho que aprender este nuevo lenguaje fuese mucho mas sencillo. Lo que me pareció mas interesante de este lenguaje es que permite llevar a cabo implementaciones relativamente difíciles con poco código, puesto que tiene una gran cantidad de librerías. Aprendí en particular acerca de las librerías *NumPy*, para manejar vectores y matrices con facilidad, *Scikit-learn* para implementar el aprendizaje automático, *Pandas* para manipular y analizar datos y *BeautifulSoup* para extraer el contenido del HTML entre muchas otras. Además durante las primeras semanas, se nos ofreció la oportunidad de asistir a unas clases de introducción a las Redes Neuronales y al Aprendizaje Profundo o *Deep Learning*. Esto incitó a que uno de los 12 algoritmos que probamos en la fase de experimentación, fuera el de Redes Neuronales. Combinando estas clases con diversos tutoriales y documentación oficial que encontré en Internet, adquirí los conocimientos necesarios para poder empezar a utilizar con destreza los algoritmos de aprendizaje automático. Nuestra idea era poder hacer un sistema híbrido que combinase algoritmos de aprendizaje automático con dos listas negras, una almacenada localmente y la otra accesible mediante una API externa. Para ello mi compañera buscó una API externa que pudiésemos utilizar y yo creé una base de datos local que correspondería a nuestra lista negra. Conocía perfectamente cómo funcionaban las bases de datos porque había cursado a lo largo de la carrera asignaturas como Bases de Datos o Ampliación de Bases de Datos. En esta base de datos local introduje un conjunto de URLs de *phishing* que habíamos extraído recientemente. La base de datos la hice con *SQLite*. Una vez mi compañero hubo seleccionado las características que íbamos a implementar, tanto de la URL como del DOM, nos dividimos las características que había que hacer. Implementé 14 características del DOM y mi compañera implementó las otras 14. Una vez terminadas todas las características, seleccionamos los 12 algoritmos con los que queríamos hacer las pruebas. Mientras mis compañeros extraían los hiperparámetros para cada algoritmo hice unas primeras pruebas con el conjunto de datos que generaban las características una vez extraídas. Me di cuenta de que algunas características no se extraían correctamente, y el conjunto de datos resultante tenía muchas características a "-1". Esto se debía a que muchas URLs daban errores del tipo *"connection time out"* o *"max retries exceeded"* porque se intentaba acceder a ellas múltiples veces en un corto periodo de tiempo. Por tanto arreglé los errores utilizando distintos agentes de usuario y filtré el conjunto de datos para que

cogiera exclusivamente las características que tuvieran menos de 15 valores a “-1”.Después entre todos, corregimos algunas de las características que aunque parecía que funcionaban correctamente, una vez analizados los datos se veía que daban valores erróneos.

Una vez hecho este paso, empezamos a desarrollar la API juntando las listas negras que habíamos encontrado y montando la estructura base utilizando *SQLAlchemy*, *Connexion* y *Flask*. Una vez teníamos la primera parte de la API hecha, y las características corregidas, empezamos a extraer las características de diferentes conjuntos de datos. Extraje las características del primer conjunto de datos, que contenía 18,000 URLs. También estuve probando en la máquina virtual de la que disponíamos, con el conjunto de 96,000 URLs añadiendo la característica `broken_links_ratio` puesto que no conseguíamos que acabase la ejecución en ninguno de nuestros ordenadores. Para acelerar este proceso, implementé funciones de las librerías *ThreadPoolExecutor* para ejecutar multihilos y *ProcessPoolExecutor* para ejecutar los procesos de manera concurrente aprovechando las capacidades de la máquina virtual que teníamos. Durante este proceso, aprendí a utilizar comandos como `nohup` para poder mantener un proceso ejecutándose incluso si la sesión con la máquina virtual se cerraba. Además me sirvió para ver cómo funcionaban las instancias en la nube, en concreto en *Google Cloud Platform*. Después de que mis compañeros hicieran pruebas de entrenamiento con distintos tipos de conjuntos de características, sacamos las métricas para cada algoritmo y seleccionamos los que mejor resultado habían obtenido. A parte de estos experimentos, probé cada uno de los 12 algoritmos seleccionados, con las características extraídas únicamente del DOM para ver si los resultados que se obtenían eran similares a los obtenidos al juntar las características extraídas tanto del DOM como de la URL. Además mediante el algoritmo RF, evaluamos todas las características para encontrar las que más aportaban al modelo. Una vez hechos y organizados todos los experimentos, realizamos la comparación de los resultados finales y procedimos a revisar todo el código que habíamos escrito para incluir aclaraciones que facilitasen su entendimiento. Por último decidimos hacer una aplicación sencilla para el acceso a la API que fuera más intuitiva para el usuario. En esta última parte utilicé los conocimientos que había adquirido en asignaturas cómo Aplicaciones Web o la asignatura optativa de Programación en Aplicaciones Móviles de *Javascript* y HTML. Para comprobar que las peticiones a la API se hacían correctamente, utilicé la herramienta *Postman*. Una vez el proyecto estaba casi acabado, empezamos a escribir la memoria. Para ello utilizamos *LaTeX*. *LaTeX* es un sistema de composición de textos que permite que los documentos presenten una alta calidad tipográfica. Había escuchado hablar de este sistema pero no lo había llegado a aplicar a ningún documento. Nuestros tutores del Trabajo de Fin de Grado, nos ofrecieron la posibilidad de recibir alguna clase sobre *LaTeX*. Con los conocimientos que adquirimos mediante esas clases y los múltiples recursos en línea que hay al respecto, desarrollamos la memoria con relativa facilidad.

Durante el desarrollo de este trabajo, he adquirido multitud de nuevos conocimientos. Desde principio de curso tenía reuniones semanales y estaba en constante contacto tanto con mis compañeros como con mis tutores para desarrollar las tareas siguiendo los plazos establecidos. Además me gustaría destacar la cantidad de tiempo que he dedicado a la investigación, que me ha permitido aprender a evaluar conocimiento por mi cuenta y extraerlo de fuentes fiables. Para mi ha supuesto un ensayo de lo que supondría llevar a cabo un proyecto en una empresa y ha sido una grata experiencia.

Capítulo 7

Conclusiones y Trabajo Futuro

7.1. Conclusiones

En este trabajo se propone un sistema de detección híbrido de páginas web fraudulentas que combina listas negras con aprendizaje automático. Mediante este sistema de detección, primero se comprueba que la dirección de la página web a analizar no se encuentra en las listas negras. En caso de no encontrarse se extraen 39 características de la dirección de la página web y del modelo de objeto del documento. Una vez obtenidas las características se pasan al modelo generado que ha sido entrenado y probado con anterioridad y que finalmente clasifica la página web como legítima o como *phishing*.

La mayor parte de este trabajo se centra en el desarrollo y el entrenamiento de este modelo para obtener la mejor precisión posible. Para la fase de entrenamiento se ha utilizado un conjunto de 68,729 páginas web de las cuales se han extraído las 39 características. Tras evaluar el modelo utilizando 12 algoritmos diferentes de aprendizaje automático se ha realizado un análisis de cada uno de ellos variando los distintos parámetros para conseguir un mayor porcentaje en la tasa de aciertos.

Para cada algoritmo probado se buscan los mejores hiperparámetros y se genera el modelo. Posteriormente se realiza una comparación entre todos ellos y se elige el modelo que ha obtenido mejores resultados. Para cada algoritmo se han obtenido dos resultados, uno aplicando validación cruzada y el otro aplicando la división del conjunto de datos en un 70 % que corresponde al conjunto de entrenamiento y un 30 % que corresponde al conjunto de prueba. Tras observar los resultados se ha llegado a la conclusión de que el mejor algoritmo ha sido el de Bosques Aleatorios o *Random Forest* con un porcentaje de verdaderos positivos de 90.6 %, un porcentaje de falsos positivos de 2.35 % y una precisión de 95,50 %.

Como se pudo comprobar en la Tabla 5.31 el sistema propuesto supera en precisión a alguno de los trabajos seleccionados y obtiene una tasa de error bastante baja. Muchos de estos trabajos no especifican concretamente algunos de los parámetros, por lo que en algunos casos no ha sido posible calcular todas las métricas para realizar una comparación más detallada.

Si bien es cierto que existen sistemas de detección actualmente que tienen una precisión algo mayor, este trabajo cuenta con el conjunto de datos más grande de todos los estudios relacionados que se han analizado. Cuenta con un conjunto de datos inicial de 129,270 direcciones web el cuál será publicado en Internet por si pudiese ser útil para futuros trabajos o experimentos. Además se aporta un total de 39 características extraídas del modelo de objeto de documento y de la dirección de la página web que responden perfectamente a las características que poseen las páginas web de *phishing* actualmente.

Al contar con varias fases que permiten la detección de páginas web fraudulentas este sistema estaría preparado para detectar páginas de *phishing* de *zero-day* puesto que tanto la lista negra externa como la base de datos del sistema se actualizan con cada detección que realizan. La precisión se ha calculado teniendo en cuenta solo la parte de aprendizaje automático. Esta precisión podría aumentar su porcentaje si se evaluara la parte de aprendizaje automático junto con la búsqueda de las direcciones de las páginas web en las listas negras.

7.2. Trabajo Futuro

Como posibles trabajos futuros se podrían mejorar y añadir los siguientes aspectos:

- Extraer características de motores de búsqueda y de servicios de terceros. Esto no se ha llevado a cabo en este proyecto porque se trabajaba con un conjunto de datos muy grande y el tiempo de ejecución era muy elevado. Añadir este tipo de características supondría aumentar aún mas el tiempo de ejecución, pero es cierto que podría suponer una mejora en la tasa de aciertos.
- Extraer características de Procesamiento Natural del Lenguaje como se hace en el trabajo [SBDD19] que podría incrementar el porcentaje de precisión.
- Añadir un módulo de detección de imágenes que permita comparar los iconos de las páginas legítimas con los de las páginas que se están analizando. Algunas páginas de *phishing* no implementan el icono de página o emplean uno diferente al de la página legítima.
- Analizar el contenido de las páginas que se cargan dinámicamente así como el contenido de las páginas redirigidas y adaptar las características si fuera necesario. Con esto se lograría examinar el contenido generado por *JavaScript*, como por ejemplo formularios, que no pueden ser detectados solo leyendo el contenido de la página.
- Generar conjuntos de datos cada cierto tiempo y entrenar el modelo para poder adaptarlo a nuevos patrones de *phishing* que puedan surgir. Esto permitiría mantener actualizado el sistema de detección de nuevas páginas web de *phishing*.
- Crear un *plug-in* para el navegador en el que introducir el sistema de detección para que el usuario pueda tener acceso a él fácilmente.

Resumen del Trabajo en Inglés

Capítulo 8

Introduction

8.1. Introduction and Current Context

Today's world is fully digitized. It is estimated that by the end of 2019 there were a total of 4,574,150,134 [NO20] users connected. Most people communicate with each other using the Internet through a mobile device or a computer. In addition, the number of people banking online or using e-commerce has increased considerably due to the convenience and the assistance many of these services offer.

Unfortunately, this expansion has also been reflected in an increase in the number of attacks, specifically phishing attacks. In this type of attacks, social engineering is used to try to steal personal or confidential information from the user, pretending to be a trusted company or person. For example, sensitive company information, passwords, bank account data or credit card data. The first contact with the user usually takes place through emails, instant messaging systems, social networks or even phone calls.

To steal information, the most used technique is to redirect the users to a fraudulent web page, very similar to the trusted web page so that the users enter sensitive information thinking they are doing it on the trusted website. In this work we will use the term phishing web page to refer to a fraudulent or malicious web page.

Figure 8.1 shows the graph of websites with malicious content detected by Google Safe Browsing in January of the last few years. The red line represents the number of fraudulent web pages detected.

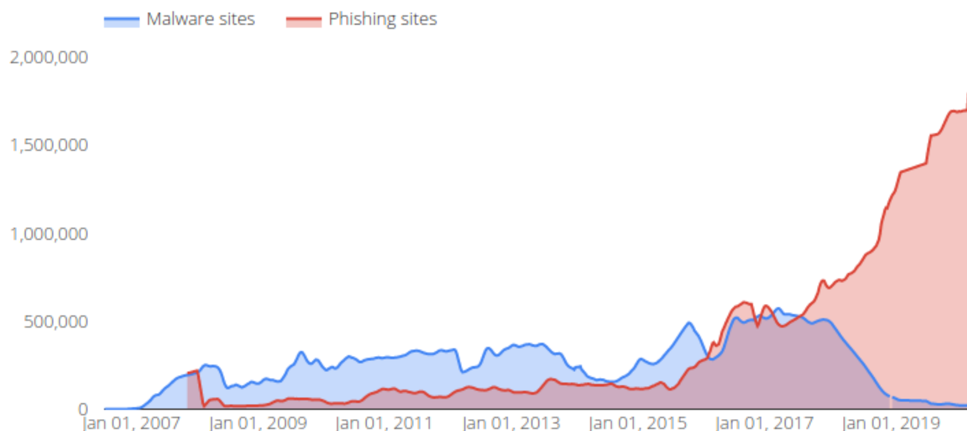


Figure 8.1: Google Safe Browsing Statistics Graph [Bro20]

Since the early 2020s, the pandemic declared by the COVID19 outbreak has had millions of people around the world confined to their homes, with the purpose of controlling its spread. In order to continue with economic activity as much as possible, many companies and their corresponding workers have had to appeal to telecommuting, and electronic commerce has increased considerably.

According to the article published on May 11, 2020 by the research group APWG [APWG20], in the first period of 2020, the number of phishing reports they have, has risen 5.38 % and the sector with the most phishing attacks is the one that includes SaaS and emails, with 33.5 % of the total number of phishing attacks.

Figure 8.2 shows the sectors that have received the most phishing attacks in the first quarter of 2020.

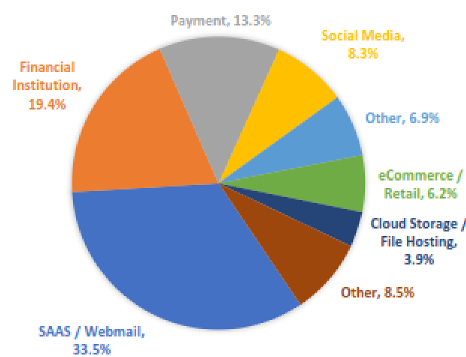


Figure 8.2: Chart of the most attacked sectors [APWG20]

Figure 8.3 shows a real example of a malicious email that was circulating in March 2020.

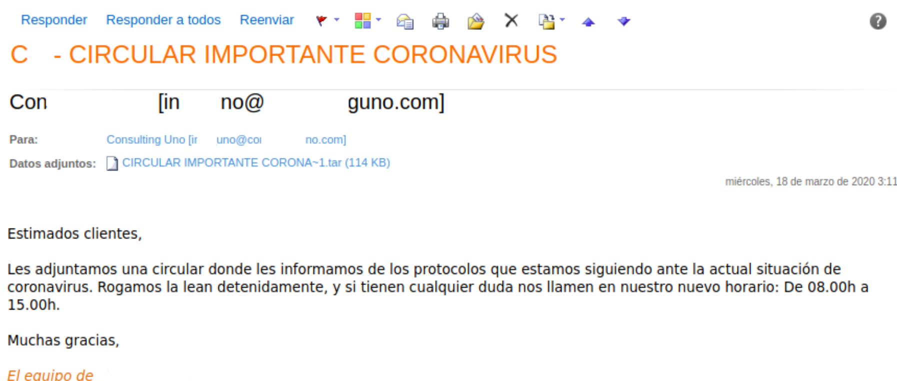


Figure 8.3: Malicious Email Screenshot [INC20]

Figure 8.4 shows the graph of websites with malicious content detected in the last three years (May 2018 to May 2020). The red line represents the number of fraudulent web pages detected. We can see two large peaks during the COVID19 pandemic, with a maximum of 59,525 fraudulent websites identified in one day, specifically April 26.

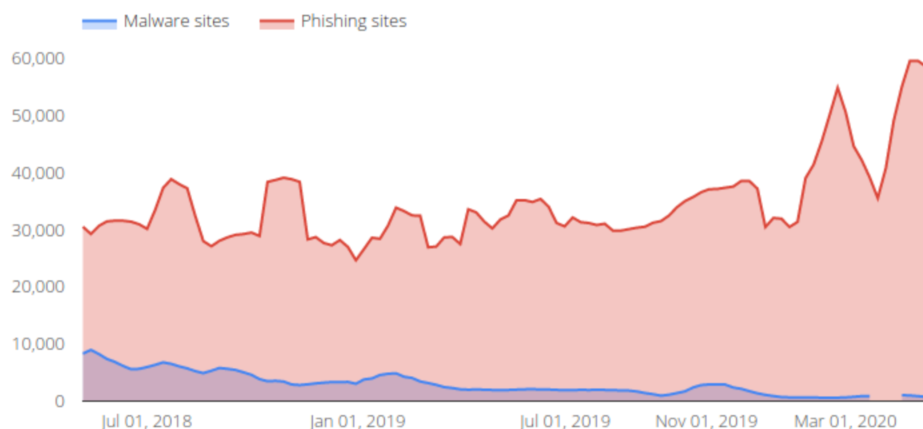


Figure 8.4: Malicious content websites [Bro20]

We observe that as time goes by, phishing attacks are refined. Currently, 74 % of phishing attacks already use HTTPS.

HTTPS is an application protocol based on the HTTP that uses encryption in SSL/TLS. In the Webroot [APWG20] threat report they emphasize that although the S stands for "secure", it really is related to privacy. When we see a green lock icon in the browser, it only means that the information that will be transmitted to that site is encrypted and delivered securely, but it does not guarantee that the destination is trustworthy.

As we can see in the Figure 8.5, the percentage of phishing web pages that use HTTPS currently reaches 74 %. That is why it is very important to develop a phishing detection system that not only considers the protocol used, but also takes into account all the features that phishing web pages present that make them distinguishable from legitimate ones. In addition, many of the current solutions use blacklists to categorize web pages because they have a low percentage of FP. A 2009 study showed that between 47 % and 83 % of phishing takes 12 hours to be blacklisted from the first time it was detected [SWW+18]. This has likely improved over time, but is still a problem today. Therefore, it would be a determining factor to be able to detect phishing web pages in real time.

This paper proposes a real-time detection system for phishing web pages based on the extraction of features from the DOM and the URL of the web page and their further analysis using machine learning algorithms.

The final goal of this project is to compare our detection system with other existing works and try, as far as possible, to equal or exceed its percentage of TP and to minimize the percentage of FP.

A TP is a result in which the model correctly predicts the positive data. On the other hand, a FP is a result in which the model incorrectly predicts the negative data. Therefore, applied to this case, a TP would occur when analyzing a phishing page, the model detects that it is phishing, and a FP would be when analyzing a legitimate page, the model detects that it is phishing when it is not. The higher the percentage of TP and the lower the percentage of FP, the better the model.

Figure 8.5: Malicious content websites [APWG20]

To carry out the experiments, several data sets have been used. A first set with about 18,000 URLs and a second set with 96,000 URLs. These sets have been refined throughout the different experiments carried out before applying machine learning algorithms to reduce noise and avoid errors.

In addition, to complete the proposed detection system, an API REST has been developed with a simple web application. This application not only integrates the machine learning model, but also has two blacklist databases, one of the blacklists is provided by the API Google Safe Browsing, and the other one was made with some phishing URLs that we found.

8.2. Context

This Final Degree Project has been carried out within the Analysis, Security and Systems Group (GASS Group, `https://gass.ucm.es/`, Group 910,623 from the catalog of groups recognized by the Complutense University of Madrid) as part of the research project activities THEIA (Techniques for Integrity and Authentication of Multimedia Files of Mobile Devices) with reference FEI-EU-19-04.

8.3. Workplan

This section explains the phases that have taken place throughout the whole project.

1. **Investigation**: This first phase was accomplished over a period of three months approximately and it was carried out on the theme of the work: detection of fraudulent websites. The initial investigation was based on both detection of phishing and ransomware threats. So far works on phishing are less than ransomware works, this was one of the main reasons why, together with our tutors, we decided to delve into the detection of phishing threats. Once the research was done, works that had been carried out so far and their results were identified and progress was made from this point on. The numerous articles in scientific journals, conferences and some books on the subject helped to expand our knowledge. Using the search engine Google Scholar, specialized in academic documents, along with the library of the Complutense University of Madrid, the necessary information was obtained. We

searched mainly for publications on detecting phishing attacks as well as measures and tools to prevent these attacks. Many of the works used in the research phase were found thanks to references from other publications already studied. The programming language Python and the library Scikit Learn, that would be used during development, were studied in depth.

2. **Development**: In the second phase, that is the development of the proposal, we have started once the research phase was sufficiently advanced and we acquired all the necessary knowledge. At this moment we have started the implementation of the proposed system: the development of a method that allows identifying if a web page is categorized as phishing or not phishing. The investigation has continued in the background while we developed and codified the proposal, we only looked for and consulted specific concepts or at times when implementation could be difficult. In this phase we carry out the search and creation of the necessary data sets to train and evaluate the proposed system. We have also created the local database and searched for an API that could provide us a blacklist as completed as possible.

3. **Tests**: In the third phase, 12 algorithms have been selected previously for which we performed all kinds of tests with different features and parameters to obtain the best combination for each one. We developed, coded, tested, and redesigned the system numerous times to achieve the final design. Once we obtained the metrics of the different algorithms, we selected the algorithm which had the best result. This result has been verified using validation curve.

4. **Results**: In the last phase, we analyzed all the results obtained during the tests and we draw conclusions from those results. According to the data obtained, we made small changes to the code from the previous phase and we perform new tests that could increase the results.

Capítulo 9

Conclusion

9.1. Conclusions

In this work we propose a fraudulent web page hybrid detection system that combines blacklists with machine learning. Through this detection system first it verifies if the address of the web page being analyzed is in the blacklists. If not, 39 features are extracted from the web page address and the document object model. Once this features are obtained they are passed to the generated model, that has been previously trained and tested, and that finally classifies the website as legitimate or phishing.

Most of this work focuses on the development and training of this model to obtain the best possible accuracy. For the training phase, we have used a set of 68,729 web pages, from which 39 features have been extracted. After evaluating the model using 12 different machine learning algorithms, we perform an analysis of each of them, varying the different parameters to achieve a higher percentage of true positive rate.

For each algorithm tested, we searched the best hyperparameters and generate the model. Subsequently, we made a comparison between all of them and the model that obtained the best results is chosen. For each algorithm, two results were obtained, one applying cross validation and the other applying the division of the data set by 70 % for the training set and 30 % for the test set. After observing the results, we have concluded that the best algorithm has been the Random Forest with a percentage of true positives of 90.6 %, a percentage of false positives of 2.35 % and an accuracy of 95.50 %.

As it could be verified on Table 5.31 the proposed system surpasses some of the other works in precision, as well as obtaining a fairly low error rate. Many of these do not clearly specify some parameters, so in certain cases it has not been possible to calculate all the metrics for a more detailed comparison.

Although there are currently detection systems that have a slightly higher precision, this work has the largest data set of all related studies that has been analyzed. We have a data set of 129,270 web addresses that will be published on the Internet in case it could be useful for future work or experiments. In addition, we provide a total of 39 features extracted from the document object model and the address of the web page, which perfectly responds to the features that phishing web pages currently own.

By having several phases that allow the detection of fraudulent web pages, this system would be prepared to detect phishing pages of zero-day since both, external blacklist and system database are updated with every detection they make. Accuracy has been calculated taking only into account the machine learning part. This precision could increase its percentage, if the machine learning part were evaluated together with the search of the addresses of the web pages in the blacklists.

9.2. Future Work

This work allows to add or improve the following aspects:

- Extract features from search engines and third-party services. This was not done in this project since we worked with a very large data set and the execution time was very high. Adding this type of features would mean increasing the execution time even more, but it is true that it might result in an improvement in the accuracy.
- Add an image detection module that allows to compare the icons of the legitimate pages with those of the pages that are being analyzed. Some phishing pages do not implement the page icon or use a different one than the legitimate page.
- Analyze the content of dynamically loaded pages as well as the content of redirected pages and adapt the features if necessary. With this it would be possible to examine the content generated by JavaScript, such as the forms that cannot be detected just by reading the page content.
- Extract features using Natural Language Processing methods like the study [SBDD19] that could increment the percentage of accuracy.
- Generate data sets from time to time and train the model to adapt it to new phishing patterns that may arise. This would keep the detection system up to date for new phishing web pages that may appear.
- Create a plug-in for the browser in which to introduce our detection system to improve the user experience and accessibility.

Bibliografía

[AKF19] Sahar Abdelnabi, Katharina Krombholz, and Mario Fritz. Whitenet: Phishing website detection by visual whitelists. *arXiv preprint arXiv:1909.00300*, 2019.

[APWG20] Inc Anti-Phishing Working Group. Phishing Activity Trends Reports. https://apwg.org/trendsreports/, March 2020.

[BAS19] Mehdi Babagoli, Mohammad Pourmahmood Aghababa, and Vahid Solouk. Heuristic nonlinear regression strategy for detecting phishing websites. *Soft Computing*, 23(12):4315–4327, 2019.

[BK20] Simon Bell and Peter Komisarczuk. An analysis of phishing blacklists: Google safe browsing, openphish, and phishtank. In *Proceedings of the Australasian Computer Science Week Multiconference*, pages 1–11, 2020.

[Bro20] Google Safe Browsing. Google Safe Browsing – Google Transparency Report. https://transparencyreport.google.com/safe-browsing/overview?hl=en, January 2020.

[Cia14] Robert B Cialdini. *Influencia: ciencia y práctica de la persuasión*. Ilustrae, 2014.

[Ebu20] EbubekirBbr. Phishing Detection. https://github.com/ebubekirbbr/pdd, 2020.

[FC10] Jeremy Feigelson and Camille Calman. Liability for the costs of phishing and information theft. *J. INTERNET L.*, 13:10–16, 2010.

[GAP18] Brij B Gupta, Nalin AG Arachchilage, and Kostas E Psannis. Defending against phishing attacks: taxonomy of methods, current issues and future directions. *Telecommunication Systems*, 67(2):247–267, 2018.

[Gar07] Tom McCall Gartner. Gartner Survey Shows Phishing Attacks Escalated in 2007; More than $3 Billion Lost to These Attacks. https://web.archive.org/web/20090602192527/http://www.gartner.com/it/page.jsp?id=565125, Dicember 2007.

[HF08] Cormac Herley and Dinei Florêncio. A profitless endeavor: phishing as tragedy of the commons. In *Proceedings of the 2008 New Security Paradigms Workshop*, pages 59–70, 2008.

[HHF+11] Mingxing He, Shi-Jinn Horng, Pingzhi Fan, Muhammad Khurram Khan, Ray-Shine Run, Jui-Lin Lai, Rong-Jian Chen, and Adi Sutanto. An efficient phishing webpage detector. *Expert systems with applications*, 38(10):12018–12027, 2011.

[INC20] INCIBE. Distribución de malware vinculado a Covid-19 suplantando varias empresas — INCIBE. https://www.incibe.es/protege-tu-empresa/avisos-seguridad/distribucion-malware-vinculado-covid-19-suplantando-varias, March 2020.

[Ins15] Ponemon Institute. The cost of phishing & value of employee training., 2015.

[JG16] Ankit Kumar Jain and Brij B Gupta. A novel approach to protect against phishing attacks at client side using auto-updated white-list. *EURASIP Journal on Information Security*, 2016(1):9, 2016.

[JM06] Markus Jakobsson and Steven Myers. *Phishing and countermeasures: understanding the increasing problem of electronic identity theft*. John Wiley & Sons, 2006.

[KPRK19] G. J. W. Kathrine, P. M. Praise, A. A. Rose, and E. C. Kalaivani. Variants of phishing attacks and their detection techniques. In *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*, pages 255–259, 2019.

[LYC+19] Yukun Li, Zhenguo Yang, Xu Chen, Huaping Yuan, and Wenyin Liu. A stacking model using url and html features for phishing webpage detection. *Future Generation Computer Systems*, 94:27–39, 2019.

[MC07] Tyler Moore and Richard Clayton. Examining the impact of website take-down on phishing. In *Proceedings of the Anti-Phishing Working Groups 2nd Annual ECrime Researchers Summit*, eCrime '07, page 1–13, New York, NY, USA, 2007. Association for Computing Machinery.

[MFSE14] Samuel Marchal, Jérôme François, Radu State, and Thomas Engel. Phishstorm: Detecting phishing with streaming analytics. *IEEE Transactions on Network and Service Management*, 11(4):458–471, 2014.

[NO20] Gfk local ICT Regulators other reliable sources Nielsen Online, International Telecommunications Union. World Internet Users Statistics and 2020 World Population Stats. https://internetworldstats.com/stats.htm, May 2020.

[Ope20] OpenPhish. OpenPhish - Phishing Feeds. https://openphish.com/phishing_feeds.html, 2020.

[OS11] Ugiomo S Odaro and Benjamin G Sanders. Social engineering: phishing for a solution. *Advances in Communications, Computing, Networks and Security Volume 8*, page 88, 2011.

[Phi20] PhishTank. PhishTank: Developer Information. https://www.phishtank.com/developer_info.php, 2020.

[RFF+20] Muhammad Hafiz Ramli, Ahmad Izatul Hisham Faugi, Nur Mawaddah Mohd Faizal, NorHaziqah Mohd Khadri, and Jasni Mohamad Zain. Anti-phishing with google extension 3h1m using blacklist algorithm. *MALAYSIAN JOURNAL OF COMPUTING*, 5(1):362–373, 2020.

[RP19] Routhu Srinivasa Rao and Alwyn Roshan Pais. Detection of phishing websites using an efficient feature-based machine learning framework. *Neural Computing and Applications*, 31(8):3851–3873, 2019.

[SBDD19] Ozgur Koray Sahingoz, Ebubekir Buber, Onder Demir, and Banu Diri. Machine learning based phishing detection from urls. *Expert Systems with Applications*, 117:345–357, 2019.

[Sit20] Alexa Top Sites. Alexa - Keyword Research, Competitive Analysis & Website Ranking. https://www.alexa.com/, 2020.

[Str] Javelin Strategy. Research (2005b) phishing: consumer behavior and awareness. *Syndicated Report Brochure*, 294.

[SWW+18] Steve Sheng, Brad Wardman, Gary Warner, Lorrie Cranor, Jason Hong, and Chengshan Zhang. An empirical analysis of phishing blacklists, Jun 2018.

[XHRC11] Guang Xiang, Jason Hong, Carolyn P Rose, and Lorrie Cranor. Cantina+ a feature-rich machine learning framework for detecting phishing web sites. *ACM Transactions on Information and System Security (TISSEC)*, 14(2):1–28, 2011.

[ZHC07] Yue Zhang, Jason I Hong, and Lorrie F Cranor. Cantina: a content-based approach to detecting phishing web sites. In *Proceedings of the 16th international conference on World Wide Web*, pages 639–648, 2007.