UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

DETECCIÓN DE ANOMALÍAS EN TRÁFICO DE RED CON MACHINE LEARNING

Alba Ramos Pedroviejo

Tutor: Manuel Antonio Sánchez-Montañés Isla

MAYO 2021

DETECCIÓN DE ANOMALÍAS EN TRÁFICO DE RED CON MACHINE LEARNING

AUTOR: Alba Ramos Pedroviejo
TUTOR: Manuel Antonio Sánchez-Montañés Isla

Grupo de Neurocomputación Biológica
Dpto. Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
mayo de 2021

Resumen (castellano)

Este Trabajo Fin de Grado se enfoca en el problema de detectar anomalías en tráfico de red mediante métodos de aprendizaje automático a partir de datos NetFlow. Si bien se han realizado numerosos esfuerzos en la comunidad para detectar estas anomalías mediante métodos supervisados, esta aproximación presenta el problema de que es necesario tener un conjunto de datos con numerosos ejemplos de ataques para que el algoritmo los aprenda. Esto, debido a que se trata precisamente de anomalías, no es posible en conjuntos de datos de tráfico real. En este sentido, surgen otros dos problemas: los ataques se tendrán que generar de manera artificial para compensar el desequilibrio y, además, los modelos no serán capaces de detectar de forma precisa ataques *zero-day* que no hayan ocurrido otras veces en el pasado, al no tener ejemplos con los que entrenar.

Este trabajo pretende analizar y demostrar de forma práctica esta problemática, investigando otras formas más precisas de resolverla mediante métodos de aprendizaje no supervisado, así como realizar una comparativa con el rendimiento de modelos supervisados cuando el conjunto de datos se equilibra de forma artificial. Se presenta un sistema que no solamente sea capaz de detectar ataques actuales y comunes, además de ataques *zero-day* sin necesidad de haberlos visto previamente, sino que también sea capaz de no alertar ante tráfico benigno que siga patrones distintos a los que se han conocido hasta el momento. El sistema permitirá recibir datos de fuentes heterogéneas mientras conserven el formato de entrada requerido.

Para resolver el problema se ha utilizado parte del dataset UGR'16, sobre el que se han entrenado y evaluado distintos modelos y obtenido los mejores resultados con un IsolationForest, seguido por un perceptrón multicapa. La arquitectura desarrollada permite construir un sistema capaz de detectar con precisión anomalías en tráfico de red de fuentes heterogéneas. Este desarrollo se puede trasladar, posteriormente, a un conjunto mayor de datos y los resultados seguirían siendo correctos.

Abstract (English)

This Bachelor Thesis focuses on the problem of anomaly detection in network traffic using machine learning methods by analyzing NetFlow data. Although there have been numerous efforts in the community to detect these anomalies using supervised methods, this approach presents the problem that it is necessary to have a dataset with numerous examples of attacks in order for the algorithm to learn them. Because attacks are, in fact, anomalies, this is not possible in real traffic datasets. In this sense, two other problems arise: the attacks will have to be generated in an artificial way to compensate for this imbalance and, furthermore, the models will not be able to precisely detect zero-day attacks that have not occurred previously, as they do not have examples to train with.

This thesis aims to analyze and demonstrate this problem in a practical way, investigating more precise ways of solving it using unsupervised machine learning methods, as well as comparing their performance with that of supervised methods when the dataset is artificially balanced. A system is presented that is not only capable of detecting up-to-date and common attacks, as well as zero-day attacks without the need to have previously seen them, but is also capable of not alerting to benign traffic that follows patterns different from those that are known. The system will allow receiving data from heterogeneous sources as long as they preserve the required input format.

To solve the problem, part of the UGR'16 dataset has been used, on which different models have been trained and evaluated and obtained the best results with an IsolationForest, followed by a multilayer perceptron. The developed architecture allows building a system capable of accurately detecting anomalies in network traffic from heterogeneous sources. This development can then be carried over to a larger dataset and the results would still be correct.

Palabras clave (castellano)

Anomalías, tráfico de red, aprendizaje automático, inteligencia artificial, benigno, ataque, supervisado, no supervisado, clasificación, conjunto de datos, ataques zero-day.

Keywords (inglés)

Anomaly, network traffic, machine learning, artificial intelligence, benign, attack, supervised, unsupervised, classification, dataset, zero-day attacks.

Agradecimientos

A la Universidad de Granada, en especial a Roberto Magán y a José Camacho, por sus aclaraciones y por proporcionar tanto el conjunto de datos como parte de las herramientas utilizadas en este proyecto.

A Idoia, por ser el faro que ilumina el mar de nuestra escuela.

A Fernando, por su apasionada forma de compartir sus conocimientos, por sus ganas de vivir tan contagiosas y por dejar un pedacito de su corazón en mí.

A mi tutor, Manuel, por lograr transmitir su pasión por la ciencia de datos a sus alumnos y, particularmente, a mí, para realizar este trabajo.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1 Introducción

1.1 Motivación

Los ataques de red representan una amenaza global, creciente y cambiante que afecta a un gran número de empresas. Situaciones límite como la pandemia por coronavirus que estamos viviendo han supuesto, además, un aumento notable de este tipo de ataques. Es más, la forma de atacar cambia con el tiempo y aparecen amenazas novedosas que suponen un gran peligro, al no haber sido estudiadas previamente y no saber cómo se comportan. En vista de esta situación, es conveniente desarrollar un sistema capaz de proteger frente a todos estos tipos y formas de amenazas.

La detección de anomalías en el tráfico de red es una tarea que ha sido ampliamente estudiada por la comunidad científica en los últimos años, pero que sigue presentando limitaciones debido a la naturaleza cambiante del tráfico y a la posibilidad de aparición de ataques nunca antes vistos por el sistema (denominados ataques *zero-day*). Por este motivo, y por la gran utilidad que podría tener un sistema capaz de detectar con precisión dichos eventos, se ha desarrollado este Trabajo Fin de Grado. El sistema que aquí se propone es capaz de detectar eventos anómalos maliciosos y ataques *zero-day* de forma precisa a partir de flujo de red preprocesado de tal forma que sea posible que provenga de fuentes heterogéneas. Para ello, se utilizan modelos de aprendizaje automático supervisados y no supervisados y se demuestra de forma práctica cómo los modelos supervisados no son capaces de resolver el problema si el conjunto de datos no está equilibrado. El sistema ha sido probado exitosamente con tráfico de red real a partir de datos extraídos del dataset UGR'16.

1.2 Objetivos

Los objetivos del presente Trabajo Fin de Grado son:

1. Analizar el estado del arte: por un lado, sobre el panorama actual de la ciberseguridad, con el fin de comprender la necesidad y el problema de detectar ataques de red. Por otro lado, sobre diferentes aproximaciones, modelos de machine learning y contribuciones recientes de la comunidad para resolverlo.
2. Estudiar diferentes conjuntos de datos relacionados con el problema, con el fin de valorar sus diferencias, ventajas y desventajas y elegir el más adecuado sobre el que desarrollar un modelo que resuelva el problema.
3. Entrenar diferentes modelos de aprendizaje automático supervisado y no supervisado, con el fin de evaluar sus diferencias y rendimiento a la hora de resolver el problema.
4. Proponer un modelo que resuelva adecuadamente el problema de acuerdo a distintas métricas de rendimiento.

1.3 Organización del documento

El documento se compone de los siguientes apartados:

- *Estado del arte*: resumen de los distintos tipos de sistemas de detección de intrusiones y análisis tanto de técnicas de aprendizaje automático como de conjuntos de datos relacionados con la detección de ataques en tráfico de red. Revisión literaria de documentos y trabajos relacionados con esta temática.
- *Diseño e implementación*: descripción de la metodología a aplicar durante el desarrollo del proyecto, las herramientas utilizadas y las pruebas ejecutadas.
- *Análisis de resultados*: estudio y comparativa de los resultados de las pruebas ejecutadas en el proyecto.
- *Conclusiones y trabajo futuro*: resumen de los resultados obtenidos en el proyecto y planteamiento de posibles líneas de investigación futuras a partir de los mismos.

2 Estado del arte

2.1 Ciberataques: panorama actual

A lo largo de los años ha habido un esfuerzo notable por desarrollar sistemas capaces de detectar los ataques de red más novedosos utilizando diferentes herramientas proporcionadas por el machine learning. No es de extrañar, ya que el tráfico consistente en ataques de red cada vez aumenta más: de hecho, este tipo de ataques representaron el 32% de ataques sufridos por compañías estadounidenses en el año 2019, como se puede observar en la figura 2-1 extraída de Statista [1].

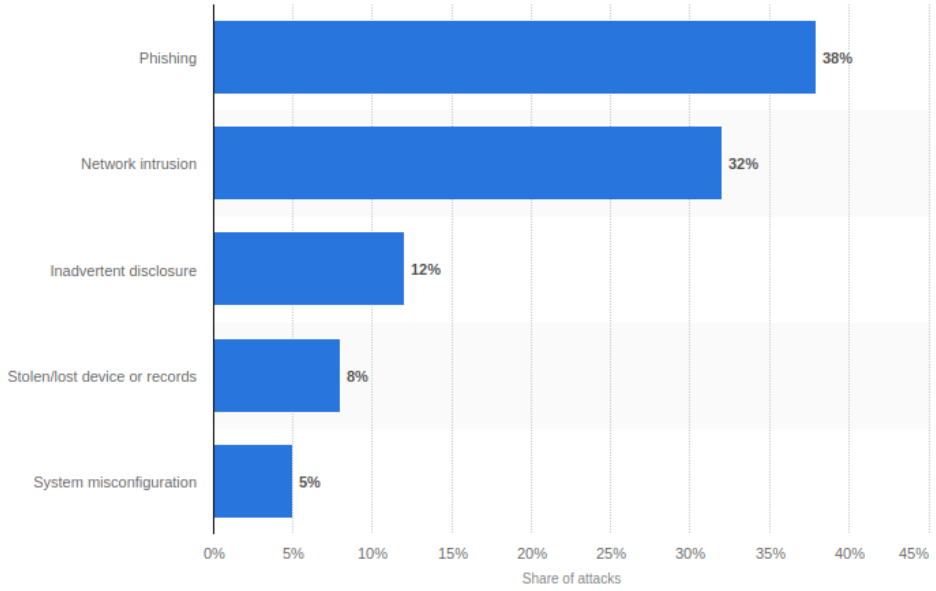

Figura 2-1: ataques más comunes sufridos por compañías estadounidenses (2019).
Extraído de [1]

Asimismo, el CNI también alerta de un creciente número de ciberataques en España cada año, según el periódico ABC [2]. De hecho, recientemente, un ataque botnet tuvo paralizado el SEPE durante más de siete días [3]. Es más, con la aparición de la pandemia por la COVID-19, los ciberataques aumentaron, según afirma el informe publicado por la Interpol en Agosto de 2020 [4]. Dichos ataques consistieron principalmente en campañas de spam relacionadas con el coronavirus, según la figura 2-2 extraída del propio informe, además de ataques de DoS y ransomware.

En definitiva, nos encontramos en una época en la que este tipo de ataques está más presente que nunca. Además, con el auge del teletrabajo y el Internet of Things (IoT) será aún más necesario desarrollar sistemas de detección de intrusiones, en adelante IDS, para proteger tanto a usuarios como a compañías y garantizar el cumplimiento de los principios de la seguridad informática.

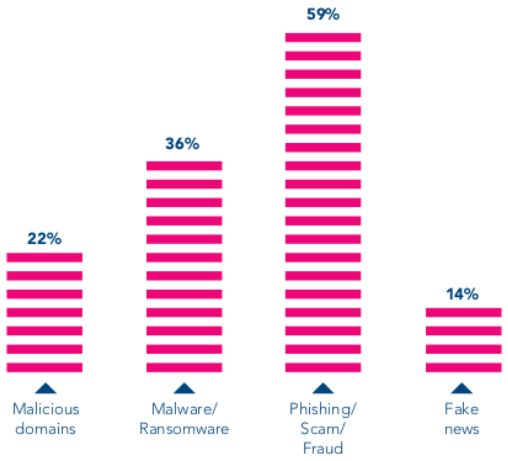

Figura 2-2: distribución de ataques relacionados con la COVID-19 en España.
Extraído de [4]

2.2 Sistemas de detección de intrusiones

Según [5], un IDS es una herramienta capaz de detectar comportamientos no autorizados en un ordenador o una red. A diferencia de los firewalls, que utilizan un conjunto de reglas predefinidas para detectar ataques, los IDS van analizando el tráfico y detectando patrones de comportamiento.

Los IDS se clasifican de acuerdo a dos factores, según [6]:

- Según el lugar donde se realiza la detección de ataques:
    - *Host-based IDS*, en adelante HIDS: se coloca el sistema en un dispositivo y se analiza su información cada cierto tiempo, comparándola con la que había anteriormente para detectar cambios inapropiados. Esta información puede venir del sistema de archivos, de la RAM, de los logs o de cualquier otro lugar. Asimismo, se analizan también los recursos a los que accede cada programa, con el objetivo de detectar accesos indebidos.
    - *Network IDS*, en adelante NIDS: el sistema analiza el tráfico de una red en busca de comportamientos anómalos o ataques conocidos y almacenados en una base de datos.
- Según la forma de detectar los ataques:
    - *Detección basada en firmas*: el tráfico de red se compara con patrones de ataques ya conocidos y preconfigurados en el sistema, denominados firmas.
    - *Detección basada en anomalías*: el tráfico de red se compara con un modelo de normalidad ya configurado que determina el comportamiento aceptable en dicha red. Este modelo se construye usando técnicas de machine learning. La principal ventaja de esta categoría frente a la anterior es la capacidad para detectar ataques *zero-day* desconocidos hasta el momento, pero esto también puede dar lugar a la aparición de falsos positivos: la

aparición de actividad legítima no vista hasta el momento y que, por tanto, se queda fuera del modelo de normalidad preestablecido, notificándose como un ataque [7]. Notificar parte del tráfico benigno como maligno no supone un problema tan grave comparado con permitir el paso de ataques *zero-day*.

2.3 Machine learning

El aprendizaje automático o *machine learning* es una rama de la inteligencia artificial que busca que las máquinas aprendan a detectar patrones en los datos y generalizarlos a otros conjuntos de datos de manera autónoma [8]. Este proyecto se centra en algoritmos de clasificación, es decir, aquellos que arrojan como resultado una clase (tráfico normal o anómalo), frente a algoritmos de regresión (que arrojarían, como resultado, un número en un intervalo continuo). Asimismo, se centra tanto en aprendizaje supervisado, donde los datos estarán etiquetados previamente, como en aprendizaje no supervisado, donde no se conocen las etiquetas de antemano y se utilizan técnicas como *clustering* [9], que clasifica los datos sin disponer de las etiquetas de antemano.

En el proceso de aprendizaje automático, los algoritmos son entrenados con un conjunto de datos, de los que aprenden, y, a partir de ahí, crean un modelo que puede utilizarse para predecir la etiqueta de otros datos que no hayan sido vistos previamente. En la figura 2-3 se puede ver cómo será este procedimiento para nuestro proyecto. Los datos consistirán en tráfico de red del pasado, recogido y preprocesado mediante métodos que se irán explicando en secciones posteriores, que engloba comportamientos legítimos (figuras verdes, que representan el grueso del tráfico) y también comportamientos fraudulentos (figuras rojas, que representan anomalías sobre el tráfico esperado). Estos datos se utilizarán para entrenar diferentes modelos y, una vez entrenados, se puede poner en producción el mejor de ellos para que monitorice y etiquete datos no vistos hasta entonces, habiéndolos preprocesado como se hizo anteriormente. En el proceso de entrenamiento, donde se conoce de antemano la clasificación si se trata de aprendizaje supervisado, es común dividir los datos en dos subconjuntos: conjunto de entrenamiento o *training*, para que el algoritmo aprenda, y conjunto de pruebas o *test*, para comprobar que el aprendizaje es correcto y el algoritmo clasifica correctamente.

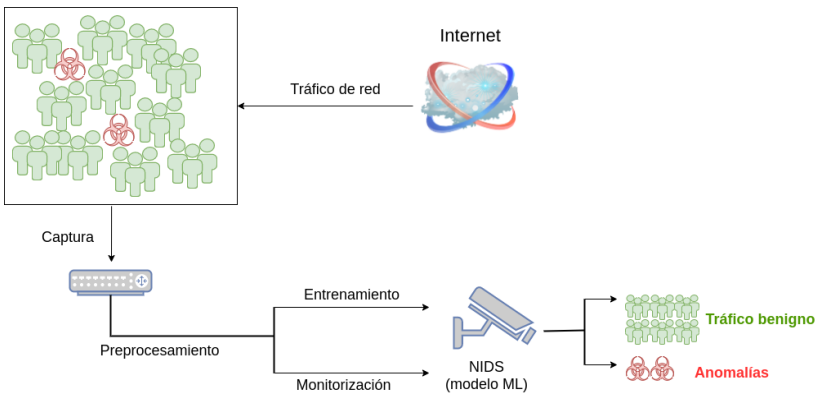

Figura 2-3: proceso de aprendizaje y monitorización del sistema

Se pueden clasificar los algoritmos de machine learning para la detección de anomalías de acuerdo a la forma en la que realizan la predicción. Esta clasificación se resume en la tabla 2-1 y se especifica a continuación.

- *Basados en proximidad*: predicen la etiqueta de un dato en función de cómo sean los datos cercanos. Por tanto, un dato será detectado como anómalo si cerca de él no hay otros datos [10]. Un ejemplo típico de este tipo de algoritmos es K-Nearest Neighbours (KNN), donde se mide la distancia del dato a sus *k* datos "vecinos" más próximos, de forma que si esta distancia es grande el dato será anómalo. Otro ejemplo es LocalOutlierFactor (LOF), que realiza una métrica similar a KNN pero teniendo en cuenta las desviaciones locales de los datos respecto a las de sus vecinos [11].

- *Modelos lineales*: separan los datos utilizando un hiperplano, por lo que éstos deben ser linealmente separables [12]. Un ejemplo sería OneClassSVM (OCSVM), que trata de aprender solamente de los datos considerados normales, creando una frontera de decisión sobre ellos y clasificando como anomalías aquellos datos fuera de la misma [13]. Otros ejemplos incluyen la Regresión Logística o C-Support Vector Classification (SVC).

- *Probabilísticos*: para un dato de entrada predicen una distribución de probabilidades de pertenecer a cada etiqueta, en vez de la más probable. Un ejemplo interesante es Copula-Based Outlier Detection (COPOD) [14], un novedoso algoritmo basado en funciones cópula capaz de detectar anomalías sin necesidad de parámetros y computacionalmente eficiente.

- *Ensembles*: basan su predicción en lo que un grupo de modelos prediga, de forma que obtiene mayor precisión que la que obtendrían los algoritmos individuales por separado. Esta predicción se puede obtener, por ejemplo, con un voto por mayoría de lo que predigan los modelos individuales, o con un voto ponderado si se quiere dar más importancia a algunos de ellos [15]. Un ejemplo es IsolationForest, que realiza predicciones utilizando un conjunto de árboles de decisión.

- *Redes neuronales*: algoritmos formados por conjuntos de unidades organizadas en diversas capas que se transmiten información entre sí, emulando el funcionamiento del cerebro humano [16]. Si hay una diferencia en el número de unidades (neuronas) entre dos capas, variará la dimensionalidad del problema entre esas capas, lo que permite que cada capa se especialice en una característica de los datos y se construya un sistema muy robusto a la hora de resolver problemas. Un ejemplo de este tipo de red es el perceptrón multicapa (MLP).

- *Otros*: se utilizarán también otros modelos que no encajan en las categorías anteriores:

    - *Árboles de decisión*: predice utilizando estructuras en forma de árbol, donde cada rama representa una decisión sobre un atributo y cada hoja representa

la clase predicha tras el conjunto de decisiones tomadas hasta llegar a la misma [17].

- *Elliptic Envelope:* asume que los datos normales siguen una distribución gaussiana y, por tanto, los anómalos serán aquellos que se alejen de la distribución asumida [18]. Para este tipo de distribución, el algoritmo forma una elipse alrededor de los datos como frontera.

Categoría	Algoritmos	Tipo de aprendizaje
Basados en proximidad	KNN, LOF	No supervisado
Modelos lineales	OCSVM	No supervisado
	Reg. logística	Supervisado
	SVC	
Probabilísticos	COPOD	No supervisado
Conjuntos	Isolation Forest	No supervisado
Redes neuronales	MLPClassifier	Supervisado
Otros	Árbol de decisión	Supervisado
	EllipticEnvelope	No supervisado

Tabla 2-1: clasificación de algoritmos utilizados

2.4 Comparativa de datasets

El problema principal a la hora de diseñar un IDS basado en aprendizaje automático es encontrar un dataset adecuado con el que entrenar al sistema y que aprenda a detectar ataques del mundo real [19]. A lo largo de los años, el esfuerzo realizado para generar estos datasets ha ido en aumento, y es que no solo es necesario ir corrigiendo errores que otros datasets anteriores puedan contener, sino ir reflejando la evolución de los ataques a lo largo del tiempo y adaptarse al comportamiento cambiante de los usuarios en la red (por ejemplo, el uso de unos protocolos frente a otros). No solo eso, sino que también hay que modelar con precisión el tráfico benigno que representaría a los usuarios que utilizan la red de forma lícita: según [19], el esfuerzo de quienes generan los datasets se ha centrado en mejorar el tipo de ataques generados, y no tanto en la simulación de un tráfico benigno realista. Asimismo, otro factor limitante es que un gran número de datasets utilizados a lo largo de los años no contienen tráfico realista, ya que ha sido generado íntegramente de forma artificial, en un entorno controlado utilizado únicamente para el desarrollo del mismo.

A continuación se realiza una comparativa entre algunos de los datasets más modernos y que han sido utilizados en la literatura para el diseño de IDSs que se resume en la tabla 2-2:

- *KDD-99* [20]: este dataset, a pesar de su antigüedad, ha sido ampliamente utilizado en la construcción de IDS debido a que es fácil de obtener y bastante completo, tanto en lo que se refiere al número de atributos que describen cada dato (41 atributos) como al total de datos recogidos (4.9 millones de registros). Sin embargo, el tráfico que contiene es generado de forma artificial y existen errores, tales como registros duplicados [21]. Si bien otros datasets posteriores como NSL-KDD buscan corregir estos errores, son tan antiguos que ya no representan ni el tráfico de red ni los ataques de la actualidad. Es por eso que muchos trabajos que utilizan este dataset obtienen buenos resultados de entrenamiento, pero esos mismos algoritmos aplicados a un dataset más reciente fallan debido a las características obsoletas de los datos en KDD-99, según [22]. A pesar de ello, la revisión de la literatura entre los años 2007-2017 realizada por Salo et al en [23] muestra que la gran mayoría de estudios se han realizado con este dataset, a pesar de su antigüedad, lo que sugiere que estos IDS están desactualizados y no están preparados para detectar ataques en el mundo de hoy.

- *UNSW-NB15* [24]: este dataset fue generado utilizando la herramienta IXIA PerfectStorm para generar ataques a servidores de una red construida para la ocasión. Debido a esto, el tráfico generado no es realista, sino que, más bien, sigue comportamientos teóricos, pues no hay nada de tráfico real detrás. Además, contiene menos tráfico que el dataset anterior (2.5 millones de registros), pero detecta ataques más variados (9 ataques frente a los 4 que detecta KDD-99).

- *UGR-16* [19]: se trata de un dataset con más de 16 millones de datos recogidos de un Proveedor de Servicios de Internet (ISP) español durante un período superior a 4 meses. Debido a que se construyó en el año 2016 proporciona datos más actualizados, con protocolos ampliamente utilizados y hasta 9 ataques actuales. Asimismo, al tratarse de información recogida de un ISP, los datos reflejan el comportamiento de diversidad de usuarios reales. El tráfico recogido es en su mayoría tráfico real de ese ISP, pero además se han ido introduciendo ataques generados que se entremezclan con el flujo normal del ISP, dentro del cual también hubo ataques reales. Debido a la naturaleza de los datos y al tiempo durante el cual han sido recogidos, permite realizar un análisis temporal del flujo de red y estudiar su periodicidad.

- *CICIDS-2017* [25]: este dataset recoge tráfico de red generado durante 5 días por ordenadores con sistemas operativos actuales y diversos, tales como Windows 10 o Kali. Si bien el tráfico generado es realista, se trata de una arquitectura diseñada específicamente para la generación de esta información. Es más completo en ataques que los anteriores datasets (detecta hasta 12 ataques), pues se basa en un reporte de McAfee de 2016, y además proporciona un mayor número de atributos

(83 en total) para cada dato recogido. Sin embargo, incluye registros sin clase y otros con la información incompleta, por lo que sería necesario corregirlo antes de trabajar con él.

- *CSE-CICIDS-18* [26]: la novedad que aporta este dataset frente a otros anteriores es la forma en la que ha sido construido, puesto que se han creado perfiles de usuarios atendiendo a un análisis previo de los comportamientos de la red con el objetivo de reproducir estos comportamientos en el sistema y así generar tráfico benigno realista. Como consecuencia, se consigue representar una gran y actualizada variedad de ataques (6 en total), así como reflejar un comportamiento realista debido al análisis previo de la red.
- *MAWI*: este dataset viene de tráfico real que lleva recogiéndose desde el 2001 y clasificándose mediante detectores de anomalías. Al ser tan extenso permitiría realizar un análisis durante largos períodos de tiempo. Sin embargo, al abarcar tanto tiempo hay fracciones del dataset que tienen problemas. Por ejemplo, en el año 2015 hubo paquetes duplicados debido a una mala configuración de uno de los routers del sistema. Asimismo, al clasificarse los datos mediante detectores de anomalías del equipo desarrollador del dataset, podría contener falsos positivos [19].

Dataset	Registros	Atributos	Clases
KDD-99	4.9 M	41	normal o ataque (4 categorías)
UNSW-NB15	2.5 M	49	normal o ataque (9 categorías)
UGR'16	16.9 M	12	background o ataque (9 categorías)
CIC-IDS 2017	3.1 M	83	normal o ataque (12 categorías)
CSE-CIC-IDS 2018	15.4 M	80	normal o ataque (6 categorías)
MAWI	Creciente	9	anomalía, sospechoso, aviso, normal

Tabla 2-2: comparativa de propiedades cuantitativas de diferentes datasets

En vista de las características de cada dataset, el más adecuado para este proyecto es UGR'16, puesto que proporciona tráfico realista, actual y representativo de una gran diversidad de usuarios de la red.

2.5 Trabajos relacionados

A lo largo de los años ha habido distintas propuestas interesantes para el desarrollo de IDS que se han basado en el dataset KDD-99, por lo que estas no se tendrán en cuenta debido a la antigüedad del mismo y a su obsolescencia, ya que no es capaz de representar el espacio de ataques que existe en la actualidad. En este sentido, se revisará la literatura del año 2018 en adelante con el objetivo de mostrar el panorama más actualizado posible.

En el estudio realizado en 2019 por Yilmaz y Masum [27] se ha resuelto el problema de desequilibrio en el dataset UGR'16 mediante el uso de redes neuronales GAN para generar ejemplos de tráfico de ataque que incluir en el dataset, mejorando así la precisión de los modelos de aprendizaje supervisado. Dicho método es perfectamente aplicable a otros datasets desequilibrados. Cabe destacar que este desequilibrio es lo normal en tráfico de red, ya que precisamente la anomalía a detectar son los ataques que se mezclan con el tráfico benigno. Sobre este mismo dataset, Magán-Carrión y Díaz-Cano [28] resolvieron exitosamente el problema del desequilibrio mediante la técnica SMOTE, además de que su estudio arroja resultados prometedores al utilizar un conjunto de datos actualizado. Kostas, en el año 2018 [29], también obtuvo buenos resultados al aplicar algoritmos de machine learning sobre otro dataset actualizado: CICIDS2017.

La técnica SMOTE también fue usada por Lin et al [30] en el desarrollo de un sistema capaz de resolver dinámicamente el problema del desequilibrio, utilizando además mecanismos de deep learning para afrontar el problema de la clasificación. Este sistema se probó con el dataset CSE-CIC-IDS2018, que consiste en tráfico de dispositivos del IoT, obteniendo resultados satisfactorios también en este campo.

Tres características destacan en estos trabajos: el esfuerzo por distinguir entre distintos tipos de ataques (métodos supervisados), la importancia de utilizar conjuntos de datos actualizados, equilibrados y representativos y el uso creciente de mecanismos de deep learning para la clasificación. Sin embargo, Phong et al [31] proponen en su estudio el uso de métodos no supervisados que permitan detectar ataques *zero-day*, que el modelo no ha visto previamente y que no están en el dataset, de forma precisa. En este sentido, han desarrollado el modelo GEE, compuesto por un Autoencoder que detecte las anomalías y una técnica capaz de proporcionar una descripción sobre los ataques encontrados, con el objetivo de facilitar su clasificación. Este es un avance importante respecto a otros estudios, ya que aporta la ventaja de poder detectar ataques no vistos anteriormente con precisión y sobre un dataset desequilibrado.

Finalmente, cabe destacar el artículo escrito por Magán-Carrión et al en el año 2020 [22] en el que proponen una metodología a seguir para desarrollar IDS comparables entre sí. Dicha metodología sugiere dividir el desarrollo en 6 pasos: extracción y selección de atributos, preprocesamiento de datos, selección de hiperparámetros, selección del modelo y métricas del rendimiento. De este modo, IDS de distintos autores podrán ser fácilmente comparables entre sí. Por tanto, en este trabajo se seguirá dicha metodología con el objetivo de facilitar a la comunidad el análisis del sistema propuesto.

3 Diseño e implementación

3.1 Dataset UGR'16

Las ventajas que aporta este dataset [19] respecto a los demás hace que sea el más adecuado para desarrollar los objetivos de este trabajo. La característica más relevante del mismo es que refleja la evolución ciclo-estacionaria del tráfico, pues los datos fueron recogidos a lo largo de 4 meses, a diferencia de otros datasets que fueron generados en períodos más cortos. Esto permite conocer la evolución del tráfico a lo largo de los meses, así como del día y la noche.

Además, al recogerse los datos de un ISP español que provee servicios en la nube, el tráfico abarca a una gran y diversa cantidad de usuarios legítimos con comportamientos muy diferentes. Esto es una gran ventaja frente a otros datasets que solo recogen tráfico de entidades concretas, como universidades. También es una ventaja frente a otros datasets más recientes, pero que han sido generados de forma artificial, a la hora de reflejar el comportamiento de los usuarios de Internet.

También es el dataset que más datos recogidos tiene. Si bien MAWI también contiene tráfico realista y está en constante crecimiento, también contiene fallos en los datos, cosa que UGR'16 no tiene. Finalmente, cabe destacar que, si bien tiene menos atributos en comparación con otros datasets, esto es debido a que sus creadores ya realizaron un proceso de selección de los mismos antes de publicarlo, por lo que los atributos que tiene son los más representativos para esos datos.

El dataset ha sido construido recogiendo tráfico NetFlow entrante y saliente de los routers del ISP, anonimizando las direcciones IP para respetar la privacidad. El tráfico, tras procesarse, ha sido etiquetado y se proporciona dividido por semanas y también separando los ataques de cada semana a ficheros aparte según su clase. Estos ficheros se proporcionan tanto en formato *csv* como *nfcapd*. El conjunto de datos está dividido en dos partes:

- *Calibración*: capturas de tráfico normal del ISP, sin ataques generados. Sin embargo, tras realizar un análisis del tráfico, los autores detectaron un comportamiento anómalo en el período comprendido entre el 11 de abril y el 25 de abril de 2016, que resultó ser un ataque de escaneo SSH. Dicha anomalía puede observarse en la figura 3-1, en la que claramente se observa una concentración de puntos rojos en el período indicado, lo que implica un aumento del tráfico SSH.
- *Test*: captura de tráfico normal del ISP junto con ataques generados sintéticamente. Aparte de estos, también hubo ataques reales tanto de escaneo UDP como de campañas de spam en este periodo.

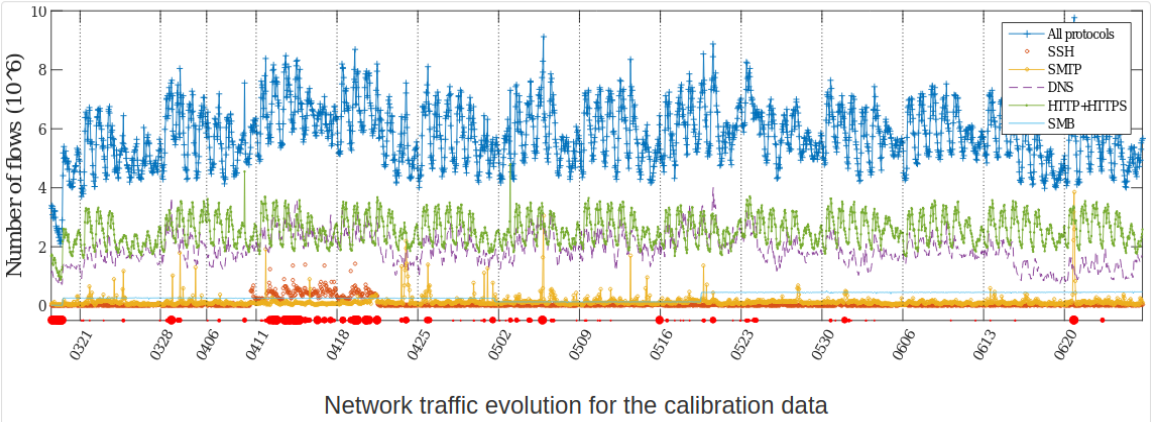

Figura 3-1: distribución del tráfico en el conjunto de calibración de UGR'16.
Extraído de [19]

UGR'16 clasifica el tráfico de acuerdo a las siguientes clases: background (no se sabe si es un ataque o no), blacklist (la IP de ese registro estaba en la lista negra del ISP), anomaly-spam (campaña de spam real detectada en el tráfico recogido), anomaly-udpscan (escaneo de puertos UDP real detectado en el tráfico recogido), ssh scanning (escaneo SSH real detectado en el tráfico recogido), dos (ataque de denegación de servicio), scan11 (ataque de escaneo de puertos uno a uno), scan44 (ataque de escaneo de puertos varios a varios), exf1KB (ataque de botnet en el que máquinas infectadas envían un fragmento de 1KB) y exf1MB (ataque de botnet en el que máquinas infectadas envían un fragmento de 1MB).

3.2 Subconjuntos de datos

El dataset UGR'16, como ya se ha explicado, se encuentra etiquetado y dividido semanalmente. Además, dentro de esta división se pueden encontrar los ficheros de tráfico correspondiente a ataques divididos por tipo. Gracias a esta estructura, es posible crear datasets más pequeños partiendo de estos ficheros. Para este proyecto se han creado ficheros que contengan exclusivamente tráfico benigno y anómalo, estos últimos por tipo de ataque. En la tabla 3-1 se muestran los ficheros que se han creado, la cantidad de filas que serán utilizadas de cada uno y el lapso temporal que abarcan. La diferencia en las fechas se debe a que el modelo de machine learning deberá entrenarse con datos anteriores en el tiempo a aquellos con los que se pruebe, ya que solo de esta manera se puede determinar que el algoritmo predice correctamente de cara al futuro. Concretamente, se tomará el 13 de Junio de 2016 como fecha límite para datos de entrenamiento. Se explicarán todas estas decisiones en los siguientes subapartados.

Fichero	Filas utilizadas	Fechas utilizadas
benign.csv	1029	06/06/2016 - 29/08/2016
attacksSSH.csv	7	11/04/2016 - 12/04/2016
attacks_ssh_august.csv	25	01/08/2016 - 23/08/2016
attacks_spam.csv	25	08/08/2016 - 09/08/2016
attacks_dos_august.csv	25	08/08/2016 - 09/08/2016
attacks_botnet_august.csv	25	01/8/2016

Tabla 3-1: ficheros generados para el experimento

3.3 Herramientas utilizadas

El proyecto ha sido desarrollado en Python, uno de los lenguajes por excelencia en ciencia de datos. Python es un lenguaje de programación que facilita la creación de código legible y cuenta con una curva de aprendizaje más baja que otros, al ser un lenguaje interpretado (sin necesidad de compilación), dinámico y que soporta la orientación a objetos. Cuenta con multitud de librerías auxiliares que es posible utilizar para el desarrollo de proyectos de ciencia de datos. En los siguientes subapartados se describen las que han sido utilizadas para este.

3.3.1 Google Colaboratory

Esta plataforma permite ejecutar código en la GPU de Google, lo que ofrece toda la potencia de estas máquinas a la hora de desarrollar proyectos de machine learning. Este tipo de proyectos suele implicar el manejo de miles de datos en el mejor de los casos, siendo la escala mayor en los proyectos reales. En máquinas orientadas al usuario medio, esto implicaría fácilmente su saturación. Con esta herramienta se ha desarrollado el Jupyter notebook del proyecto, pues permite implementarlos y tenerlos almacenados en Google Drive, contando con hasta 12.72GB de RAM y 107.77GB de disco.

3.3.2 NumPy y Pandas

Librerías que facilitan el trabajo con conjuntos de datos. Pandas facilita la lectura de datos en formato *csv*, mientras que NumPy optimiza las operaciones con vectores y matrices de datos. Se utilizarán en el proyecto para la carga y preprocesamiento de los datos.

3.3.3 Scikit-learn

Scikit-learn [32] es una librería ampliamente utilizada en el ámbito del aprendizaje automático que proporciona implementaciones de algoritmos de aprendizaje supervisado y no supervisado, así como de clasificación y regresión. También proporciona herramientas para el preprocesamiento de los datos y métricas de rendimiento. Se utilizará a lo largo del proyecto para preprocesar los datos y entrenar modelos.

3.3.4 FaaC

Feature as a Counter (FaaC) es una herramienta de código abierto [33] que permite obtener nuevos atributos a partir de contadores de los originales. De este modo, permite representar datos de fuentes heterogéneas en un formato común y comprensible para los modelos de machine learning, así como realizar un análisis temporal del tráfico. Se utilizará para preprocesar el dataset y extraer atributos derivados de los originales.

3.3.5 PyOD

Python Outlier Detection (PyOD) [34] es una librería que implementa diferentes algoritmos para detección de anomalías, incluyendo métodos supervisados que han sido adaptados a este cometido. Los modelos de esta librería esperan una codificación binaria de las clases, donde la clase 0 indicaría que los datos son normales y la clase 1, que son anómalos. Se utilizará a lo largo del proyecto para entrenar algoritmos de aprendizaje no supervisado.

3.3.6 Hardware

El proceso de extracción de atributos se realizó en un portátil Acer Aspire E5 con 4 cores, 4GB de RAM y 1TB de HDD.

3.6 Metodología aplicada

A la hora de llevar a cabo el desarrollo del proyecto, se ha seguido la metodología propuesta por Magán-Carrión et al en [22], ya mencionada en apartados anteriores. A continuación se describen los pasos seguidos y las medidas adoptadas en cada uno de ellos.

3.6.1 Extracción de atributos

El primer paso que se realizará consiste en representar los datos en un formato comprensible para la máquina. En este sentido, será necesario que todos los atributos del dataset se encuentren en formato numérico. Existen métodos proporcionados por la librería sklearn para este cometido, como OneHotEncoder, que permitirían procesar cada fila. Sin embargo, a la hora de analizar el tráfico e implementar el sistema en tiempo real es necesario ser capaces de procesar el tráfico de red y transformarlo al formato común que se utilice por el modelo de machine learning, independientemente de la fuente de la que provenga dicho tráfico. En este sentido, un solo paquete del tráfico de red no proporcionaría información significativa acerca del tráfico, sino que es necesario analizar un flujo de paquetes.

La herramienta FaaC permite obtener nuevos atributos a partir de contadores de los originales, partiendo del dataset original y creando otro con más atributos (contadores) y menos filas. Por ejemplo, en vez de tener el atributo “Protocolo”, el cual toma valores de distintos protocolos en formato texto, como “TCP” o “UDP”, se tendrían los atributos “Protocolo_TCP”, “Protocolo_UDP”, “Protocolo_otro”, que serían contadores de cuántos

paquetes o filas del conjunto tenían como protocolo el TCP, el UDP, etc. en un intervalo de tiempo determinado.

Atributo original	Atributos derivados
src_ip	nf_src_ip_private, nf_src_ip_public
dst_ip	nf_dst_ip_private, nf_dst_ip_public
src_port	sport_zero, sport_multiplex, sport_echo, sport_discard, … sport_reserved, sport_register, sport_private
dst_port	dport_zero, dport_multiplex, dport_echo, dport_discard, … dport_reserved, dport_register, dport_private
protocol	protocol_tcp, protocol_udp, protocol_icmp, protocol_igmp, protocol_other
tcp_flags	tcpflags_URG, tcpflags_ACK, tcpflags_PSH, tcpflags_RST, tcpflags_SYN, tcpflags_FIN
src_tos	srctos_zero, srctos_192, srctos_other
in_packets	in_npackets_verylow, in_npackets_low, in_npackets_medium, in_npackets_high, in_npackets_veryhigh
in_bytes	in_nbytes_verylow, in_nbytes_low, in_nbytes_medium, in_nbytes_high, in_nbytes_veryhigh
label	background, blacklist, botnet, dos, ssh_scan, scan, spam, udp_scan

Tabla 3-2: derivación de atributos mediante FaaC

En definitiva, esta herramienta reduce el tamaño del dataset y transforma los atributos en contadores. Para ello, analiza las diferentes características del tráfico durante un intervalo de tiempo y según un lapso. Un minuto se considera un lapso adecuado en problemas de detección de anomalías [22], pero en el caso de este trabajo, dado que la mayor parte de los ataques generados tienen una duración de 2 minutos, se ha elegido este lapso para realizar el análisis. De esta forma, se tiene en cuenta la evolución temporal del flujo, una característica fundamental tanto del dataset UGR'16 como del análisis de tráfico de red en tiempo real.

Para llevar a cabo este procedimiento, se ha configurado la herramienta para que cree los mismos contadores que se describen en el ejemplo proporcionado por los desarrolladores, modificando únicamente la parte de los lapsos temporales (para indicar un lapso de 2 minutos y las fechas adecuadas a cada conjunto de datos que se esté procesando) y el

número de cores para adaptarlo a la máquina del experimento (4 cores). Este procedimiento da lugar a un dataset compuesto por 138 atributos derivados de los originales, que podemos observar en la tabla 3-2, además de un atributo *timestamp* (*ts*), cuyo fin es identificar visualmente cada partición temporal. Este procedimiento se ha realizado para todos los ficheros de datos que componen el proyecto.

Esta parte de la metodología es fundamental a la hora de diseñar un NIDS real, ya que con sencillos cambios en la configuración de la herramienta FaaC es posible representar, en un formato común y en tiempo real (utilizando un lapso de tiempo corto, entre 60 y 180 segundos, lapso en el que encaja el elegido para las pruebas realizadas en este trabajo), datos de fuentes heterogéneas, tanto estructuradas como no estructuradas, combinando la información de ambos tipos y permitiendo construir un sistema más potente capaz de transformar la información en tiempo real.

3.6.2 Preprocesamiento de datos y reducción de dimensionalidad

Para garantizar que el modelo de machine learning aprenda bien se realiza un particionamiento train-test que consiste en dividir el conjunto de datos totales en una fracción de entrenamiento o training, para que el algoritmo aprenda, y una de pruebas o test, con datos diferentes para comprobar que el algoritmo generaliza correctamente. En este caso, se dividirán los datos benignos, ordenados cronológicamente, en dos fracciones: un 70% de los mismos, que contendrán datos hasta el 13/06/2016, como training y el 30% restante, con datos del 13/06/2016 al 29/08/2016, para test. Este segundo conjunto contiene datos posteriores en el tiempo con el objetivo de comprobar cómo se comportaría el modelo al implementarse en la realidad y ver datos que siguen una distribución similar a los que conoce, pero que son futuros.

Sin embargo, realizar esta simple partición no es un planteamiento adecuado para obtener buenos resultados en este tipo de problemas: el modelo podría simplemente aprender que, con estos datos, no necesita lanzar una alarma en ningún momento. De acuerdo a esta regla de clasificación, si llegaran ataques el modelo tampoco alertaría de que está ante un dato anómalo respecto al tráfico normal, ya que ha aprendido que no es necesario hacerlo. Por este motivo, el conjunto de datos de entrenamiento se contaminará con ataques en una proporción que represente un 1% del total (7 ataques) y cuya fecha sea anterior al 13 de Junio de 2016. Por otro lado, el de test se contaminará con un 4.04% de ataques (15 ataques), entre ellos ataques que no se hayan utilizado en el de entrenamiento, y con fecha posterior al 13 de Junio de 2016. De este modo, es posible emular una situación de tráfico real compuesto por tráfico benigno principalmente y algunas anomalías que es necesario poder detectar, independientemente de su tipo. En la tabla 3-1 se pueden ver la cantidad de ataques (filas utilizadas de cada fichero) y fechas que representan las particiones realizadas.

Un gran número de algoritmos de aprendizaje automático requieren que los datos sean normalizados previamente, puesto que en un conjunto de datos podemos tener distintas

escalas entre atributos, de forma que unos tomen valores muy diferentes en relación con el resto, como podemos apreciar en la imagen izquierda de la figura 3-2. Una forma de normalizar es que todos los atributos tengan media cero y desviación típica 1, como se observa en la imagen derecha de la figura 3-2. Para realizar esta normalización se ha utilizado una instancia de la clase *StandardScaler* de sklearn que realiza el proceso de estandarización y transformación de una matriz de entrada X de acuerdo a la fórmula:

$$z = \frac{x - \mu}{\sigma}$$

Donde x representa cada ejemplo del conjunto de datos, y μ y σ representan la media y desviación estándar de los N datos, respectivamente, cuyas fórmulas son:

$$\mu = \frac{1}{N}\sum_{i=1}^{N}(x_i) \qquad \sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

La matriz de entrada será el conjunto de datos de entrenamiento. Sin embargo, cuando se quieran probar los modelos con los datos de test, estos también deberán ser estandarizados de acuerdo a la misma escala usada para los anteriores. Para ello, se puede utilizar el método *transform* de la misma instancia de *StandardScaler* que se utilizó con los datos de entrenamiento.

	count	min	max	mean	std
nf_src_ip_private	1036.0	0.0	164.0	9.951737	18.090956
nf_src_ip_public	1036.0	660.0	250095.0	171829.327220	40349.306334
nf_dst_ip_private	1036.0	0.0	200.0	22.586873	16.495737
nf_dst_ip_public	1036.0	660.0	254294.0	171725.164093	40527.423611
sport_zero	1036.0	0.0	7835.0	1996.075290	466.138219
sport_multiplex	1036.0	0.0	13.0	1.021236	1.565181
sport_echo	1036.0	0.0	7.0	0.226834	0.696224
sport_discard	1036.0	0.0	4.0	0.165058	0.582162
sport_daytime	1036.0	0.0	7.0	0.169884	0.579939
sport_quote	1036.0	0.0	4.0	0.194981	0.601619
sport_chwhereen	1036.0	0.0	4.0	0.166988	0.556135
sport_ftp_data	1036.0	0.0	232.0	17.158301	33.407107
sport_ftp_control	1036.0	0.0	1643.0	791.457529	243.695449

	count	min	max	mean	std
-0.38387891162138293	1036.0	-0.383879	8.780784	0.172245	1.010960
-4.867561219144903	1036.0	-4.875559	1.708755	-0.357217	1.065097
-1.2751791062894857	1036.0	-1.275179	9.152180	-0.097572	0.860035
-4.8662189756644105	1036.0	-4.871860	1.813966	-0.362559	1.068308
-4.080756053318697	1036.0	-4.114578	12.447714	0.104896	0.985363
-0.5953450408581105	1036.0	-0.595345	7.899212	0.071958	1.022732
-0.3210350700580053	1036.0	-0.321035	11.489306	0.061677	1.174664
-0.3207464932149078	1036.0	-0.320746	5.756555	-0.069970	0.884494
-0.3295129516354391	1036.0	-0.329513	10.181533	-0.074419	0.870823
-0.3623042595454606	1036.0	-0.362304	5.532816	-0.074946	0.886655
-0.3327480534945408	1036.0	-0.332748	6.463808	-0.049012	0.944950
-0.5554326968852031	1036.0	-0.555433	5.420701	-0.113449	0.860540
-3.192843510019627	1036.0	-3.192844	3.012479	-0.203647	0.920395

Figura 3-2: fragmento de datos antes y después de estandarizar

En la figura 3-3 se puede observar la distribución, por atributo, de los datos de entrenamiento. En ella se observa, de manera global, cómo la naturaleza y comportamiento de los atributos es muy diferente y, por lo tanto, es necesario normalizar.

Por otro lado, el dataset derivado tras la aplicación de FaaC cuenta con un mayor número de atributos, de los cuales no todos proporcionan información relevante para el análisis que se pretende realizar (de hecho, algunos de ellos son iguales en todas las filas). Sin embargo, gracias a este aumento de la dimensionalidad se consigue agrupar los paquetes de tráfico de red que conforman el dataset mediante una ventana temporal, para así conocer la evolución del tráfico.

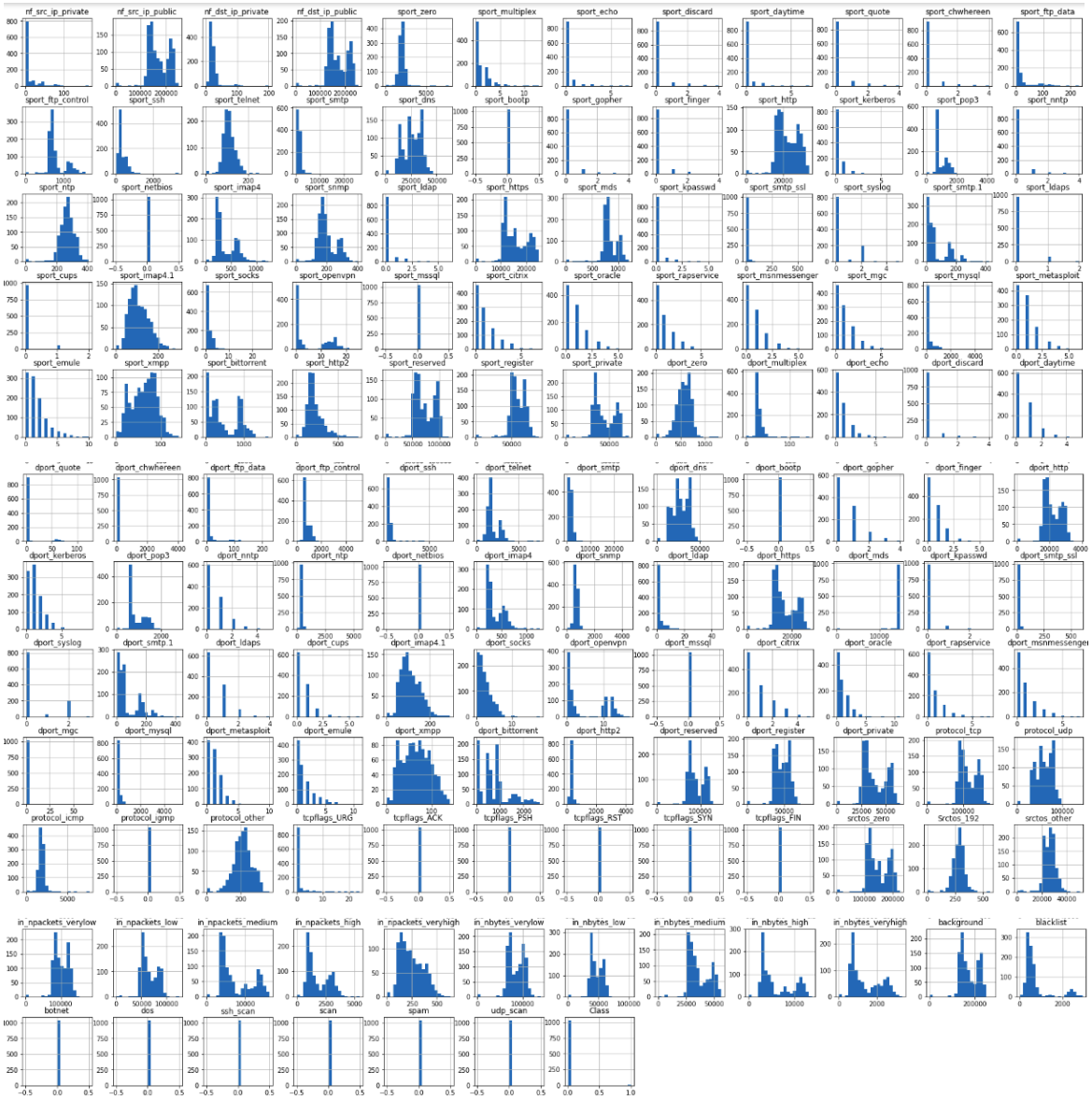

Figura 3-3: distribución de los datos por atributo

Existen numerosos métodos que permiten seleccionar los atributos más descriptivos del dataset, pero es necesario prestar atención al tipo de aprendizaje (supervisado o no supervisado) que se está aplicando, ya que no todos los métodos se pueden aplicar a ambos tipos. En el caso del presente trabajo nos centramos en el aprendizaje no supervisado: mientras que la selección de atributos es un paso que se realiza en problemas supervisados, para este caso podemos usar la reducción de dimensionalidad.

Según un estudio realizado por Zhou et al [35], a la hora de reducir la dimensionalidad de un problema no supervisado es necesario seleccionar aquellos atributos que mejor describan la distribución del conjunto de datos. Según los resultados experimentales del estudio, utilizar todos los atributos del conjunto de datos es lo que mejores resultados ofrece en la mayoría de casos. Esto tiene sentido, ya que lo que ocurre con los datos que utilizamos para entrenar el modelo podría no ocurrir con los datos utilizados para clasificación, y esto es lógico, porque ahí es donde se detectan las anomalías. Si se dejan

todos los atributos, incluso aquellos que no cambian en ningún dato de entrenamiento, se podría detectar fácilmente una anomalía cuando llegue un dato que tenga alguno de estos atributos diferentes. Como punto negativo, es necesario conservar todos los atributos, y cuando estos son muy numerosos, como es el caso del conjunto de datos de las pruebas realizadas, entonces la complejidad computacional del problema se eleva. Merece la pena, por tanto, sacrificar algo de precisión en pro de una mayor simplicidad del problema.

Entre los distintos métodos de reducción de dimensionalidad que existen se utilizará Principal Component Analysis o PCA, un algoritmo estándar utilizado ampliamente en la literatura para este tipo de problemas. Este método consiste en comprimir los atributos del problema en otros nuevos que resuman la información contenida en los originales, a sabiendas de que algo de información se perderá en este proceso. En estas pruebas se reducirá la dimensión a 10 atributos para todos los ficheros utilizados. El algoritmo PCA se ajustará según el fichero de tráfico benigno, que es el modelo de normalidad que se quiere utilizar, y el resto se transformará siguiendo este mismo ajuste.

En cualquier caso, los atributos que no sean numéricos, como el *timestamp*, deberán eliminarse, puesto que el algoritmo no podrá procesarlos. Este tipo de atributos son meramente informativos y no aportan conocimiento adicional al problema, de lo contrario habrían sido transformados a un formato numérico en el apartado anterior. Nótese que los últimos atributos se corresponden con las clases de los distintos ataques: cuando se introduzcan, por ejemplo, ataques de tipo escaneo SSH únicamente al modelo, está claro que las clases *botnet, udp_scan,* etc. no van a aportar información. A la hora de realizar pruebas sí podrían eliminarse, porque se sabrá con qué tipo de ataque se está trabajando, pero en la vida real esto no será así. Nótese, en relación con este aspecto, que según cómo se hayan obtenido los datos, las clases que aparezcan en estas etiquetas pueden no corresponderse con lo que se esperaría observar. Si se realiza un análisis exploratorio de los datos con los que se entrenará el modelo, que cuentan con ataques de escaneo SSH, se puede observar, en la imagen izquierda de la figura 3-4, que el atributo correspondiente a esta clase de ataque no está activado en ningún momento y tiene todos los valores a 0. Esto es debido a que dichos ataques fueron detectados mediante detectores de anomalías por los propios creadores del dataset, por tanto no fueron etiquetados manualmente como ataques. No ocurre así con los ataques de tipo spam que se utilizarán para realizar más pruebas, ya que estos sí se generaron artificialmente y, por tanto, están etiquetados como tal (imagen derecha de la figura 3-4).

	ssh_scan
count	1036.0
mean	0.0
std	0.0
min	0.0
25%	0.0
50%	0.0
75%	0.0
max	0.0

	spam
count	25.000000
mean	8269.880000
std	3940.254373
min	2303.000000
25%	4834.000000
50%	7676.000000
75%	10688.000000
max	16384.000000

Figura 3-4: estadísticas de ataques ssh (entrenamiento) y spam (pruebas)

3.6.3 Modelos utilizados

Para intentar resolver el problema se utilizarán tanto modelos supervisados como no supervisados, cuyo funcionamiento ya se ha explicado en el apartado 2.2. Los modelos supervisados que se probarán, proporcionados por la librería sklearn, serán LogisticRegression, SVC, MLPClassifier y DecisionTree. Los modelos no supervisados, proporcionados por las librerías PyOD y sklearn, serán KNN, LocalOutlierFactor, OneClassSVM, COPOD, IsolationForest y EllipticEnvelope. Nótese que alguno de estos modelos, como KNN, originalmente es supervisado, pero gracias a la librería PyOD se reimplementa como no supervisado.

3.6.4 Métricas de rendimiento

A la hora de evaluar los modelos probados, se entrena cada uno con la partición de datos de train y se prueba con la partición de test. De esta forma, al haber sido entrenados y probados con los mismos datos, se puede realizar una comparativa justa de sus resultados o *scores*. Para analizar la robustez de cada modelo se tendrán en cuenta los siguientes conceptos:

- Verdaderos positivos (TP): ataques o anomalías que el modelo ha clasificado como tal.
- Falsos positivos (FP): datos benignos que el modelo ha clasificado como anomalías. Conviene minimizarlos, ya que podrían resultar en un bloqueo de la IP emisora de dicho tráfico sin haber realizado ni un solo ataque.
- Verdaderos negativos (TN): datos benignos que el modelo ha clasificado correctamente.
- Falsos negativos (FN): ataques que el modelo clasifica como tráfico benigno. Conviene minimizarlos y, además, son más graves que los FP, ya que implicarían pasar por alto un ataque que podría poner en serio riesgo nuestro sistema.

Sklearn ofrece implementaciones de métricas interesantes que se pueden utilizar para medir el rendimiento de los modelos:

- *accuracy:* determina la capacidad del algoritmo para predecir correctamente. Debido a la baja cantidad de ataques en el tráfico, esta métrica normalmente arrojará resultados altos, aunque el modelo cometa errores.

$$accuracy = \frac{TP + TN}{TP + FN + TN + FP}$$

- *precision*: determina la capacidad del algoritmo para reducir los falsos positivos.

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

- *recall*: determina la capacidad del algoritmo para detectar los verdaderos positivos y, por ende, reducir los falsos negativos.

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

- *F1*: es la media armónica entre *precision* y *recall*, por tanto es la que más interesa para este trabajo. Es una métrica que crece cuando las otras dos crecen, por tanto penaliza cualquier error que cometa el clasificador, buscando así reducir tanto los FN como los FP. Concretamente, se utilizará *F1_micro* debido a que la proporción de ataques en el conjunto de datos es mucho menor que la de benignos, por tanto hay un gran desequilibrio entre ambas clases y la métrica *micro* es adecuada para estas situaciones.

$$F1 = \frac{2\ x\ precision\ x\ recall}{precision + recall}$$

3.6.5 Optimización de hiperparámetros

Las posibilidades para automatizar la búsqueda de los mejores hiperparámetros de cada modelo probado son o bien realizar una búsqueda completamente aleatoria, o bien una búsqueda en cuadrícula o *GridSearch*: dado un diccionario compuesto por pares parámetro-valores predefinidos, se realiza validación cruzada para evaluar cada combinación y una media de la puntuación que obtiene cada modelo con los datos de entrenamiento, obteniendo así la mejor combinación de parámetros para los mismos. Para ello, el algoritmo de *GridSearch* necesita conocer las clases de los datos, por lo que se pueden ver los ejemplos benignos como clase 0 y los ataques, o anomalías, como clase 1.

Para implementar esta solución, será necesario indicarle al algoritmo de *GridSearch* cómo determinar si un resultado ha sido correcto o no. Para ello, en primer lugar es necesario indicar unas clases con las que el algoritmo pueda ver qué es correcto: para esto basta con construir una lista de tantos 0 como elementos benignos contenga el conjunto de entrenamiento, y tantos 1 como ataques contenga. Asimismo, es necesario indicar una función de *scoring* que evalúe el rendimiento del clasificador durante las pruebas, ya que el modelo se va entrenando con cada combinación de parámetros. Para ello se utilizará la

métrica *F1_micro* descrita en el apartado anterior como función de *scoring* ya que, como se ha mencionado, al ser la media armónica entre *precision* y *recall* busca reducir tanto los FN como los FP.

Con la mejor puntuación de cada modelo devuelta por *GridSearch* se obtiene una idea de los mejores hiperparámetros que utiliza cada modelo para adaptarse a los datos recibidos gracias al atributo *best_params_*, y ya es posible utilizar los modelos entrenados con su mejor combinación de parámetros para predecir sobre los datos de test.

3.6.6 Reequilibrado del conjunto de datos

Como se ha explicado en apartados anteriores, si se pensara crear el modelo entrenando solo con datos benignos, sería posible comprobar que se obtendría un *accuracy* prometedor, y si luego se validara con los datos de *test*, donde ya se incorporan ataques, se vería que el score se reduciría ligeramente. Si bien seguiría siendo un resultado prometedor, no sería adecuado para el objetivo de este trabajo, porque fallaría al predecir todos y cada uno de los ataques: se estarían cometiendo falsos negativos, que es lo que se quería evitar. Tal modelo no es útil, pues simplemente identifica todo como benigno, y como realmente un ataque es una anomalía en el tráfico, su proporción es muy baja, lo que supone que la mayoría de veces el modelo acertará con esta forma tan simple de predecir, dejando pasar todos y cada uno de los ataques entrantes. Se puede mejorar el resultado de los modelos no supervisados entrenando con el conjunto de datos benignos contaminado ligeramente con ataques, como se ha explicado en apartados anteriores.

Aún así, los modelos supervisados no aprenderán correctamente, ya que no tienen suficientes ejemplos de cada clase para aprender, como se comprobará más adelante. Por este motivo, se ha utilizado la técnica SMOTE para reequilibrar las clases del conjunto de entrenamiento de forma realista. Este algoritmo genera nuevos ejemplos de la clase minoritaria basándose en la distribución de los existentes. Utilizando la idea de KNN, genera ejemplos a una distancia razonable de los k vecinos más próximos entre dos ejemplos de dicha clase [36]. En la figura 3-5 se observa un fragmento de código que realiza este procedimiento, donde se puede ver que originalmente hay solamente 7 anomalías en el conjunto y, tras aplicar SMOTE, hay tantas como datos benignos.

```
from collections import Counter
from imblearn.over_sampling import SMOTE

print('Original dataset shape %s' % Counter(y_train_bin))
sm = SMOTE(random_state=42)
x_train, y_train_bin = sm.fit_resample(x_train, y_train_bin)
print('Resampled dataset shape %s' % Counter(y_train_bin))
```

```
Original dataset shape Counter({0: 720, 1: 7})
Resampled dataset shape Counter({0: 720, 1: 720})
```

Figura 3-5: reequilibrado del conjunto de entrenamiento

3.6.7 Selección del modelo

Para comprobar la robustez durante las pruebas con los datos de test, se ha creado una función que, dada una lista con las clases de los datos y otra con las predicciones realizadas por el modelo, devuelva tres porcentajes: aciertos totales (*accuracy*), FP y FN. Asimismo, se evaluará cada modelo también sobre conjuntos que contengan únicamente ataques, contando los aciertos (TP) y errores (FN). Para estas pruebas se han probado tanto ataques de SSH como spam, DoS y botnet por separado (25 ejemplos de cada tipo), para comprobar si alguno de los tres es más complicado de detectar por los diferentes clasificadores y si han aprendido a generalizar. Se elige el modelo que mejor generalice para los distintos ataques.

3.7 Problemas encontrados y decisiones adoptadas

Debido al gran tamaño del dataset, no era viable utilizarlo entero para el desarrollo del proyecto en el entorno utilizado. Por este motivo, se intentó en un primer momento utilizar únicamente la parte de test del propio dataset, ya que contaba con tráfico representativo de todos los ataques disponibles en el dataset. La parte de test, comprimida, ocupa unos 55GB. Al comenzar la extracción de atributos con la herramienta FaaC, se descubrió que no era viable en el ordenador utilizado para realizar la investigación: la lentitud era excesiva y no era posible ejecutarlo en Google Colaboratory debido al sistema de directorios con el que la herramienta estaba preparada para trabajar. Este ha sido el principal problema a la hora de obtener tráfico benigno y procesarlo con FaaC, y es que si bien existen otras herramientas para extraer atributos, se ha decidido utilizar ésta porque es la más adaptada al dataset (ya que se desarrolló por los creadores del mismo y se probó con este).

Debido a este contratiempo, se decidió que se realizaría detección de anomalías sobre una pequeña parte del dataset, entre el 06/06/2016 y el 29/08/2016. Tras ser procesado con la herramienta FaaC, el conjunto de datos quedó con 720 patrones benignos y 7 ataques SSH para el conjunto de entrenamiento, y 309 patrones benignos y 15 ataques (8 SSH, 3 spam, 2 DoS y 2 botnet) para el conjunto de pruebas. Si bien se reduce el alcance del problema, los resultados obtenidos son igualmente válidos y extrapolables a cantidades mayores de datos, como se comprobará en el próximo apartado.

4 Análisis de resultados

4.1 Resultados con el conjunto de test

En primer lugar se realizan pruebas con el conjunto de test, que consta de 309 patrones benignos y un 4.04% de anomalías, es decir, 15 ataques: 8 SSH, 3 spam, 2 DoS y 2 botnet. En las tablas 4-1, 4-2 y 4-3 se pueden observar los porcentajes de *accuracy*, FP y FN para cada modelo no supervisado, supervisado entrenado sin balancear el conjunto de datos y supervisado tras reequilibrar el conjunto de datos, respectivamente. Se muestra, además, el tiempo que tardan en clasificar, en milisegundos.

Clasificador	Accuracy (%)	Error FP (%)	Error FN (%)	Tiempo clasificación (ms)
KNN	99.38	0	13.33	28.26
LOF	98.46	0	33.33	11.61
OCSVM	99.38	0	13.33	2.80
COPOD	100	0	0	13.23
IsolationForest	100	0	0	64.96
EllipticEnvelope	100	0	0	1.21

Tabla 4‑1: resultados en test, modelos no supervisados

Clasificador	Accuracy (%)	Error FP (%)	Error FN (%)	Tiempo clasificación (ms)
LogisticRegression	95.37	0	100	0.50
SVC	98.46	0	33.33	0.53
MLPClassifier	96.60	0	73.33	1.31
DecisionTree	96.91	0	66.67	0.43

Tabla 4‑2: resultados en test, modelos supervisados entrenados sin SMOTE

Se observa que ningún modelo comete errores de FP, por tanto se puede ver que sí han aprendido a detectar el modelo de normalidad por ser el tipo de dato más abundante. Según los resultados anteriores, parece que tanto IsolationForest como COPOD y EllipticEnvelope son buenos candidatos para resolver este problema, ya que son los que mejores resultados obtienen. Se observa claramente que un resultado alto de la métrica

accuracy no necesariamente implica que el modelo sea correcto para este problema, como se ve claramente en el caso de la regresión logística cuando el modelo se entrena sin balancear con SMOTE. Se observa, además, que los modelos supervisados clasifican los mismos datos en menos tiempo que los no supervisados, en la mayoría de casos.

Clasificador	Accuracy (%)	Error FP (%)	Error FN (%)	Tiempo clasificación (ms)
LogisticRegression	99.38	0	13.33	0.42
SVC	98.46	0	33.33	12.28
MLPClassifier	100	0	0	0.77
DecisionTree	98.46	0	33.33	0.45

Tabla 4-3: resultados en test, modelos supervisados entrenados balanceando con SMOTE

En la figura 4-1 se pueden visualizar los resultados anteriores. Se puede notar que, para los modelos de aprendizaje supervisado, los errores de FN son mayores que para los modelos de aprendizaje no supervisado. Se observa que estos son mucho mayores para el caso del conjunto de datos desequilibrado, y que mejoran bastante tras equilibrarlo con SMOTE. Esto tiene sentido: los modelos supervisados no pueden aprender correctamente a distinguir los ataques debido a que hay muy pocos ejemplos de los mismos con los que entrenar. Además, el perceptrón multicapa parece un buen candidato supervisado para resolver este problema, ya que es el que menos FN comete tas equilibrar el conjunto de datos.

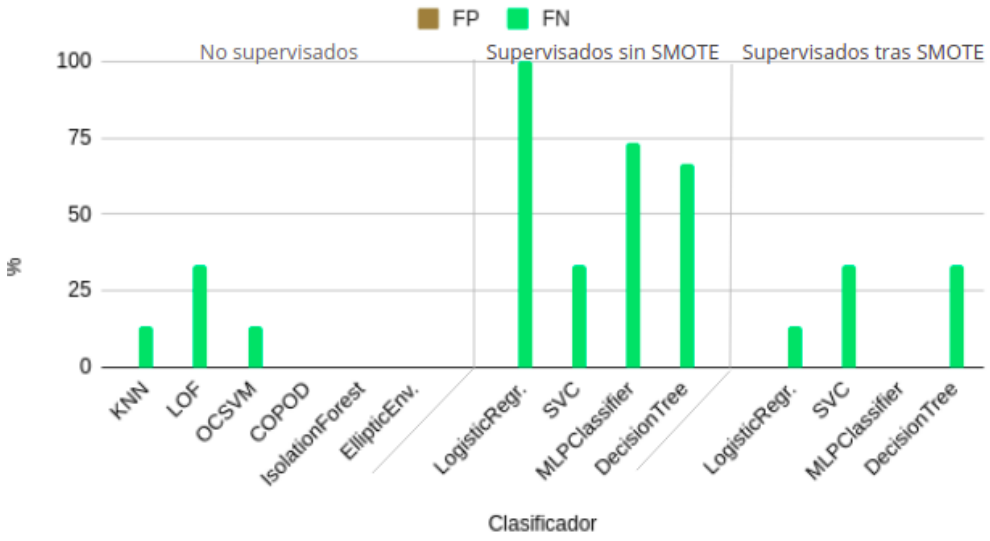

Figura 4-1: falsos positivos y falsos negativos de cada clasificador

4.2 Resultados con conjuntos de ataques

En las tablas 4-4, 4-5 y 4-6 se observa el porcentaje de aciertos (*accuracy*) de cada modelo ante cinco conjuntos de únicamente ataques de diferente tipo (25 patrones de cada tipo: spam, SSH, DoS y botnet), para así comprobar si realmente los modelos generalizan bien. Se muestran resultados para modelos no supervisados, supervisados entrenados con el conjunto de datos desequilibrado y supervisados entrenados con el conjunto equilibrado con SMOTE. Si bien la cantidad de ataques con la que han sido entrenados es escasa para emular una situación de tráfico real, interesa conocer si son capaces de detectar bien estos ataques cuantas veces sean necesarias. Los ataques DoS, spam y botnet, que los modelos no han visto previamente, emularían ataques *zero-day* del tráfico real.

Clasificador	Acc. Spam (%)	SSH (%)	DoS (%)	Botnet (%)	Total (%)
KNN	100	100	60	100	90
LOF	36	100	0	100	59
OCSVM	80	100	92	100	93
COPOD	32	12	24	24	23
IsolationForest	100	100	88	100	97
EllipticEnvelope	100	100	84	100	96
Total (%)	74.67	85.33	58	87.33	

Tabla 4-4: resultados ante ataques, modelos no supervisados

Los modelos no supervisados obtienen, en general, resultados prometedores para detectar ataques novedosos, sin cometer FP en las pruebas del conjunto de test anteriores y con un porcentaje de FN bastante bajo.

Clasificador	Acc. Spam (%)	SSH (%)	DoS (%)	Botnet (%)	Total (%)
LogisticRegression	0	0	0	0	0
SVC	0	100	0	100	50
MLPClassifier	0	0	12	0	3
DecisionTree	100	0	96	0	49

Tabla 4-5: resultados ante ataques, modelos supervisados entrenados sin SMOTE

Se confirma que los modelos supervisados no resuelven el problema correctamente si el conjunto de datos de entrenamiento no está equilibrado, logrando acertar en el mejor de los

casos un 50% de los ataques. Se observa cómo mejoran significativamente los modelos supervisados y su capacidad de detección de ataques, no solo conocidos sino *zero-day*, una vez que el modelo ha sido entrenado con un conjunto de datos equilibrado. Llama la atención el caso del árbol de decisión, que no es capaz de aprender los ataques con los que ha entrenado, pero sí otros que no ha visto nunca, como Spam o DoS.

Clasificador	Acc. Spam (%)	SSH (%)	DoS (%)	Botnet (%)	Total (%)
LogisticRegression	100	100	20	100	80
SVC	0	100	0	100	50
MLPClassifier	100	100	88	100	97
DecisionTree	100	0	96	0	49
Total (%)	75	75	51	75	

Tabla 4-6: resultados ante ataques, modelos supervisados entrenados balanceando con SMOTE

Además, la mitad de los algoritmos supervisados probados obtienen una puntuación de 0 en algún tipo de ataque, mientras que sólo uno de los seis algoritmos de aprendizaje no supervisado han fallado totalmente en algún ataque. Esto parece indicar que, en general, son mejores los algoritmos no supervisados a la hora de detectar ataques nunca antes vistos por el modelo, si bien también tardan más a la hora de clasificar. Se confirma que los datos de ataques generados mediante SMOTE para equilibrar el conjunto de datos son realistas, ya que los modelos supervisados son capaces de obtener buenos resultados con ataques reales, dentro de sus limitaciones. En la figura 4-2 se observan gráficamente estos resultados.

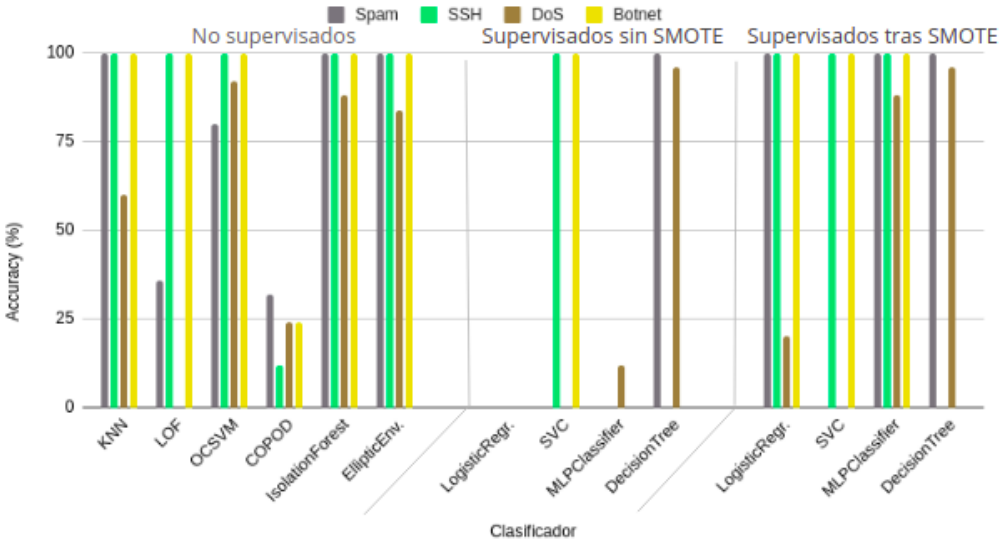

Figura 4-2: accuracy de cada clasificador ante cada tipo de ataque

Se observa, además, en la figura 4-3, el porcentaje total de aciertos frente a ataques de cada clasificador. Se comprueba, de nuevo, que los resultados son mejores para los modelos supervisados, siendo el mejor de ellos IsolationForest, con un 97% de aciertos ante ataques y ningún FN en las pruebas con el conjunto de test. De los métodos supervisados, la red neuronal, concretamente perceptrón multicapa, es el que mejor resultado obtiene, también con un 97% de aciertos ante ataques y ningún FN con el conjunto de test, tardando bastante menos tiempo en clasificar que el IsolationForest.

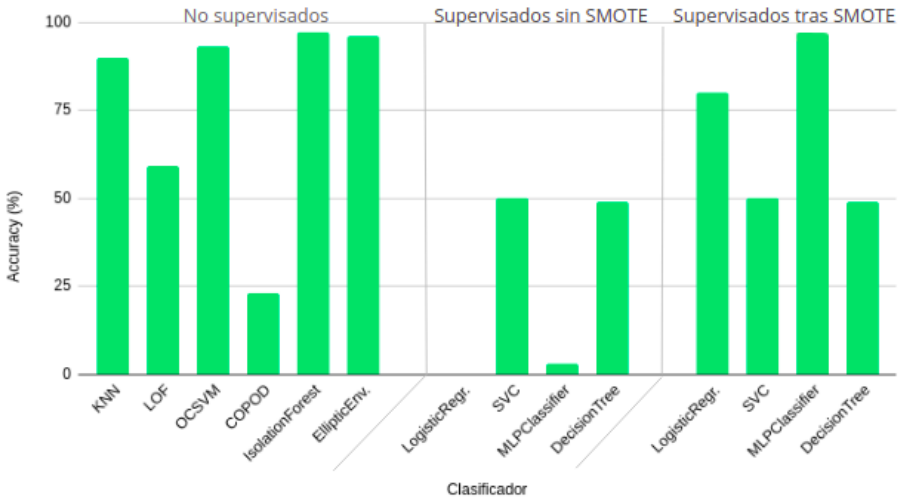

Figura 4-3: accuracy total de cada clasificador

Se observa en la figura 4-4 el rendimiento acumulado por tipo de aprendizaje frente a cada ataque. El resultado para aprendizaje supervisado se ha realizado tomando los resultados de los modelos entrenados con el conjunto de datos balanceado con SMOTE. Nótese que el ataque de DoS es el más complicado de detectar para ambos tipos de aprendizaje. Destaca la capacidad de ambos tipos de aprendizaje de detectar el ataque Botnet, un ataque que, por naturaleza, se oculta entre el resto de tráfico, haciéndolo más difícil de detectar.

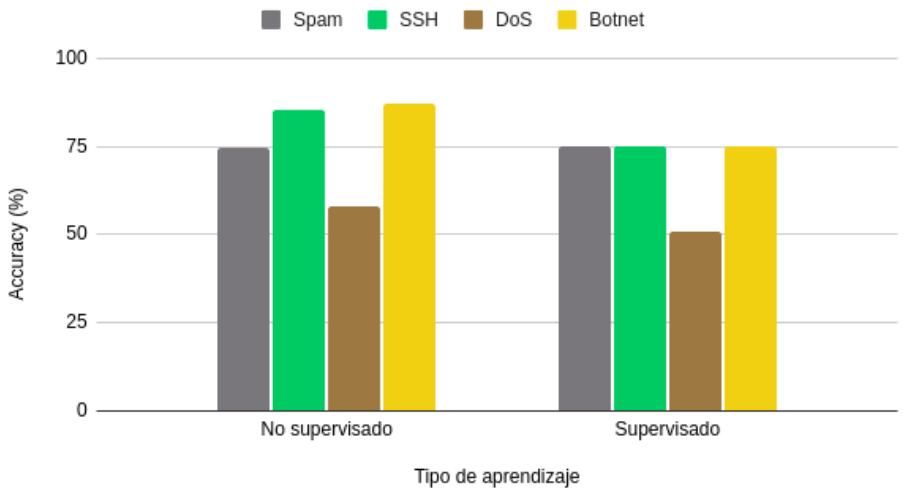

Figura 4-4: accuracy acumulado frente a cada tipo de ataque

En vista de los resultados obtenidos, se puede concluir que ambos tipos de aprendizaje resuelven correctamente el problema, con los mejores modelos de cada tipo obteniendo resultados similares. El modelo que mejor lo resuelve es IsolationForest: con él es posible detectar, sin necesidad de reequilibrar el conjunto de datos, no solo ataques conocidos, sino también ataques *zero-day* que no se hayan visto previamente. Además, el modelo también es capaz de aceptar actividad legítima no vista hasta el momento, evitando alertar ante falsos positivos. También se podría elegir un perceptrón multicapa para realizar la clasificación en mucho menos tiempo, obteniendo también resultados adecuados y que podría merecer la pena a la hora de implementar el sistema en una infraestructura en tiempo real, si bien requiere equilibrar el conjunto de datos a la hora de entrenar.

5 Conclusiones y trabajo futuro

5.1 Conclusiones

El objetivo de este Trabajo Fin de Grado era construir un sistema de detección de anomalías en tráfico de red utilizando métodos de machine learning. La arquitectura que lo resolviera debía ser capaz de detectar anomalías correspondientes a ataques, incluyendo ataques *zero-day*, pero no alertar ante tráfico benigno novedoso. Además, debía permitir trabajar con datos de fuentes heterogéneas para poder ser utilizado en una infraestructura real.

Para resolver el problema, se realizó un análisis de los distintos modelos de machine learning disponibles para ello, así como de otros esfuerzos de la comunidad orientados a este fin y de los conjuntos de datos disponibles para realizar las pruebas. Con ello se pretendía conocer las limitaciones de las soluciones propuestas recientemente y de los conjuntos de datos. Se eligió el aprendizaje no supervisado como opción más precisa para la resolución del problema y se realizaron pruebas de distintos modelos de este tipo sobre una parte de los datos NetFlow extraídos del dataset UGR'16, que contiene tráfico real que representa usuarios muy variados. También se probaron métodos supervisados, antes y después de equilibrar el conjunto de datos, para comparar su rendimiento con el de los anteriores. Para el desarrollo de las pruebas se realizó un proceso de extracción y selección de atributos, preprocesamiento de datos, selección de los mejores hiperparámetros de cada modelo y selección del mejor modelo de acuerdo a distintas métricas. La metodología seguida durante el diseño y desarrollo del proyecto facilita la comparativa de este con otros trabajos relacionados de la comunidad. Los datos utilizados durante las pruebas fueron previamente procesados con la herramienta FaaC para analizar el flujo temporal de los mismos y homogeneizar a un formato comprensible por los modelos datos provenientes de fuentes heterogéneas.

Tras realizar las diferentes pruebas y analizar los resultados obtenidos, queda demostrada la incapacidad de los modelos supervisados para resolver este problema si el dataset está desequilibrado, pero alcanzan resultados adecuados en menor tiempo que los no supervisados si el dataset se equilibra de forma realista. Las pruebas efectuadas demuestran cómo el modelo IsolationForest es el que resuelve el problema con mayor precisión, seguido por el perceptrón multicapa. Queda demostrado, por tanto, que las técnicas utilizadas a lo largo del proyecto permiten construir un sistema capaz de detectar anomalías en tráfico de red de fuentes heterogéneas con precisión y podrían extrapolarse a conjuntos de datos mayores.

5.2 Trabajo futuro

El objetivo más inmediato que se podría tener es el de integrar el sistema de detección de anomalías desarrollado con un sistema capaz de procesar el tráfico NetFlow en tiempo real

y transformarlo al formato *csv* requerido por el primero. De este modo, se tendría un nuevo sistema que capture, transforme y analice el tráfico de red para detectar anomalías en tiempo real.

Más adelante, se podría hacer que el sistema fuera capaz no solo de detectar ataques, sino de clasificarlos por tipo para aquellos que no sean *zero-day*. Para ello, se podrían tener precargadas las distribuciones de cada tipo de ataque (esto es posible porque UGR'16 cuenta con ficheros aparte con los datos separados por tipo de ataque) y comparar la anomalía detectada con cada distribución. De este modo, se podría ver a cual se asemeja más y determinar ante qué tipo de ataque se encuentra el sistema. Con este planteamiento, la arquitectura final contaría con dos capas: la primera, un sistema de alerta rápido y eficaz capaz de trabajar en tiempo real. La segunda, una más compleja que detecte el tipo de ataque ante el que se encuentra el sistema y permita tomar decisiones en función de esa información.

Otra aproximación sería construir el modelo de machine learning sobre todo el conjunto de datos, pero esta vez en un entorno big data como Hadoop o Spark. En el presente trabajo se demuestra que es posible encontrar modelos que den buenos resultados con el dataset filtrado, por tanto sería posible pasar al entorno big data y obtener resultados todavía más precisos al trabajar con todo el dataset.

Referencias

[1] "Most common cyber attacks experienced by companies in the United States in 2019". 2020. Available: https://rb.gy/dccwca [Último acceso 26 9 2020].

[2] Laura Montero Carretero. "Las ciberamenazas ensombrecen la pista de despegue de la empresa digitalizada". ABC, 2020.

[3] Alberto R. Aguilar. "El gran hackeo al SEPE es obra de Ryuk: cómo ha podido llegar el ciberataque, qué ha fallado y por qué ahora mismo es una de las mayores amenazas de internet". 2021. [En línea]. Available: https://rb.gy/f9avni. [Último acceso 10 4 2021]

[4] Cybercrime: COVID-19 impact. Interpol, p. 8 , 2020.

[5] T. Kaurl et al. "Comparison of network security tools- Firewall, Intrusion Detection System and Honeypot". International Journal of Enhanced Research in Science Technology & Engineering, 3, pp. 200-204, 2014.

[6] R. Di Pietro, L. V. Mancini (eds). "Intrusion detection systems". Springer, 38, 2008.

[7] C. Y. Ho et al. "False Positives and Negatives from Real Traffic with Intrusion Detection/Prevention Systems". International Journal of Future Computer and Communication, 1(2), 2012.

[8] Stuart J. Russell, Peter Norvig. "Artificial Intelligence: A Modern Approach", p. 9 (prefacio), p. 2 (introducción), 2010.

[9] M. Khanum et al. "A Survey on Unsupervised Machine Learning Algorithms for Automation, Classification and Maintenance". International Journal of Computer Applications, 119(13), 2015.

[10] Charu C. Aggarwal. "Proximity-Based Outlier Detection". Outlier Analysis. Springer, pp. 101-113, 2013.

[11] Philip W. "Local Outlier Factor for Anomaly Detection". Medium. 2018. [En línea]. Available: https://rb.gy/bh0bny [Último acceso: 15 3 2021]. [Último acceso: 15 3 2021]

[12] "Reconocimiento de señales de tráfico para un sistema de ayuda a la conducción". [En línea].
Available: https://docplayer.es/78743856-4-teoria-de-clasificadores.html.

[Último acceso: 28 4 2021]

[13] Paolo Oliveri. "Class-modelling in food analytical chemistry: Development, sampling, optimisation and validation issues - A tutorial". Analytica Chimica Acta. 982, pp. 9-19, 2017. doi:10.1016/j.aca.2017.05.013

[14] Z. Li, Y. Zhao, N. Botta, C. Ionescu, X. Hu, “COPOD: copula-based outlier detection”, IEEE International Conference on Data Mining (ICDM). IEEE, 2020.

[15] J. Cerquides, R. López de Màntaras. “Robust Bayesian Linear Classifier Ensembles”. Springer, 2005.

[16] Anders Krogh. “What are artificial neural networks?”. Nature Biotechnology, 26, pp. 195-197, 2008. doi:10.1038/nbt1386

[17] Ned Horning. “Introduction to decision trees and random forests”. American Museum of Natural History, 2, pp. 1-27, 2013.

[18] Miguel Ángel N. “Services Monitoring with Probabilistic Fault Detection”. Percona. 2017. [En línea]. Available: https://rb.gy/jylauk. [Último acceso: 15 3 2021]

[19] Gabriel M. F., José C., Roberto M. C., Pedro G. T., Roberto T. “UGR‘16: A new dataset for the evaluation of cyclostationarity-based network IDSs”. Computers & Security, 73, pp.411-424, 2017.

[20] KDD Cup 1999.

Available: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html,

[Último acceso: 23 8 2020].

[21] M. Tavallaee, E. Bagheri, W. Lu, A. A. Ghorbani. “A Detailed Analysis of the KDD CUP 99 Data Set”. Second IEEE international conference on Computational intelligence for security and defense applications, IEEE Press, Piscataway, NJ, USA, CISDA’09, pp. 53-58, 2009.

[22] Roberto M. C., D. Urda, Ignacio D. C., Bernabe D. “Towards a Reliable Comparison and Evaluation of Network Intrusion Detection Systems Based on Machine Learning Approaches”. Applied Sciences, 10, p. 1775, 2020.

[23] F. Salo et al. “Data Mining Techniques in Intrusion Detection Systems: A Systematic Literature Review”. IEEE Access, 6, pp. 56046-56058, 2018.

[24] “The UNSW-NB15 Data Set Description”. UNSW Canberra. 2020. [En línea]. Available:

www.unsw.adfa.edu.au/unsw-canberra-cyber/cybersecurity/ADFA-NB15-Datasets/ [Último acceso 14 9 2020].

[25] Intrusion Detection Evaluation Dataset (CIC-IDS2017). [En línea]. Available: https://www.unb.ca/cic/datasets/ids-2017.html. [Último acceso 14 9 2020].

[26] CSE-CIC-IDS2018 on AWS. [En línea].
Available: https://www.unb.ca/cic/datasets/ids-2018.html.
[Último acceso 14 9 2020].

[27] I. Yilmaz, R. Masum. "Expansion of Cyber Attack Data From Unbalanced Datasets Using Generative Techniques". Department of Computer Science Tennessee Tech University Cookeville, TN, USA, 2019.

[28] R. Magán-Carrión, I. Díaz-Cano. "Evaluación de algoritmos de clasificación para la detección de ataques en red sobre conjuntos de datos reales: UGR'16 dataset como caso de estudio". Actas de las V Jornadas Nacionales de Ciberseguridad, pp. 46-52, 2019.

[29] K. Kostas. "Anomaly Detection in Networks Using Machine Learning". [Trabajo de Fin de Máster]. Colchester, UK: School of Computer Science and Electronic Engineering, University of Essex, 2018.

[30] P. Lin et al. "Dynamic Network Anomaly Detection System by using Deep Learning Techniques". Cloud Computing-CLOUD 2019. CLOUD 2019. Lecture Notes in Computer Science, Springer, 11513, pp. 161-176.

[31] Q. Phong Nguyen et al. "GEE: A Gradient-based Explainable Variational Autoencoder for Network Anomaly Detection". IEEE Conference on Communications and Network Security (CNS), Washington, D.C., 2019.

[32] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". JMLR 12, pp. 2825-2830, 2011.

[33] A. Pérez-Villegas, J. García-Jiménez, J. Camacho. FaaC (Feature-as-a-Counter) Parser. Available: https://github.com/josecamachop/FCParser. [Último acceso 8 7 2020]

[34] Y. Zhao, Z. Nasrullah, Z. Li. "PyOD: A Python Toolbox for Scalable Outlier Detection". Journal of machine learning research (JMLR), 20(96), pp. 1-7, 2019

[35] P. Zhou et al. "Unsupervised feature selection for balanced clustering". Knowledge-Based Systems, 193, 2019.

[36] N. V. Chawla, K. W. Bowyer, L. O.Hall, W. P. Kegelmeyer. “SMOTE: synthetic minority over-sampling technique”, Journal of artificial intelligence research, pp. 321-357, 2002.

Glosario

IoT	Internet of Things
IDS	Intrusion Detection System
HIDS	Host-based Intrusion Detection System
NIDS	Network-based Intrusion Detection System
KNN	K Nearest Neighbours
LOF	Local Outlier Factor
OCSVM	One-Class Support Vector Machine
SVC	C-Support Vector Classification
COPOD	COPula-based Outlier Detection
MLP	Multi-Layer Perceptron
ISP	Internet Server Provider
FaaC	Feature as a Counter
PyOD	Python Outlier Detection
PCA	Principal Component Analysis
TP	True Positive
TN	True Negative
FP	False Positive
FN	False Negative

Anexos

A Notebook del proyecto

```
# -*- coding: utf-8 -*-
"""TFG_nb.ipynb

Automatically generated by Colaboratory.

Original file is located at

https://colab.research.google.com/drive/1qtVuCRbVQptp17PpP-IuO29qW0934Lns
"""

!pip install pyod

import pandas as pd
import numpy as np

benign = pd.read_csv("./benign.csv") # 1029 filas

x_train_ssh =
pd.read_csv("./attacksSSH.csv").sort_values('ts').iloc[:7] # En
abril, 7 filas, un 1% respecto a x_train_benign (70% de benign)
x_test_ssh =
pd.read_csv("./attacks_ssh_august.csv").sort_values('ts').iloc[:8] #
En agosto, 8 filas
x_test_spam =
pd.read_csv("./attacks_spam.csv").sort_values('ts').iloc[:3] # En
agosto, 3 filas
x_test_dos =
pd.read_csv("./attacks_dos_august.csv").sort_values('ts').iloc[:2] #
En agosto, 2 filas
x_test_botnet =
pd.read_csv("./attacks_botnet_august.csv").sort_values('ts').iloc[:2]
# En agosto, 2 filas
# Ataques de test representan un 4.04% del total, son posteriores e
incluyen ataques nuevos (spam, DoS y botnet)
```

```
# Con estos probaremos qué tal predicen los modelos solo ataques
attacks_ssh_full =
pd.read_csv("./attacks_ssh_august.csv").sort_values('ts').iloc[:25]
print("SSH: ", len(attacks_ssh_full))
attacks_spam_full =
pd.read_csv("./attacks_spam.csv").sort_values('ts').iloc[:25]
print("Spam: ", len(attacks_spam_full))
attacks_dos_full =
pd.read_csv("./attacks_dos_august.csv").sort_values('ts').iloc[:25]
print("DoS: ", len(attacks_dos_full))
attacks_botnet_full =
pd.read_csv("./attacks_botnet_august.csv").sort_values('ts').iloc[:25
]
print("Botnet: ", len(attacks_dos_full))
# TODO: misma cantidad de los 3 tipos

"""# Análisis exploratorio de los datos"""

df = benign.append(x_train_ssh)
df['Class'] = pd.Series([0]*len(benign) + ([1]*len(x_train_ssh)),
index=df.index)

df

# Mostrar estadísticas de los atributos: para cada atributo, se
muestran el total de apariciones en el dataframe,
# los valores máximo y mínimo, la media y la desviación estándar.

pd.set_option('display.max_rows', 500)
df.describe().T[["count", "min", "max", "mean", "std"]]

(df.describe())[['ssh_scan']]

"""A pesar de tener ataques de escaneo SSH, no se muestran para el
atributo ssh_scan, que tiene todo a 0. Esto es debido a que estos
ataques se detectaron mediante detectores de anomalías por los
creadores del dataset, no fueron generados artificialmente ni
etiquetados como tal. Para ataques generados artificialmente, como el
de spam de agosto, sí vemos que lo etiqueta correctamente y aparece
en las estadísticas, como se muestra a continuación."""
```

```
(attacks_spam_full.describe())[['spam']]

df.hist(figsize=(30,30), bins=20);

"""# Preprocesamiento de datos y selección de atributos"""

# lo ideal es un sistema que entrene (gridsearch) con fechas
anteriores a test
# y que le metas un ataque (de una fecha posterior al entrenamiento)
que no ha visto nunca y que
# te lo detecte como ataque
# ej: hasta 1 de junio es train, y en adelante es test con benignos y
ataques.

benign = benign.sort_values('ts')
print(benign)

# Eliminar atributos irrelevantes

nombres_atrs = list(benign.columns)
nombres_atrs.remove('ts')
X_benign = benign[nombres_atrs].values
x_train_ssh = x_train_ssh[nombres_atrs].values
x_test_ssh = x_test_ssh[nombres_atrs].values
x_test_spam = x_test_spam[nombres_atrs].values
x_test_dos = x_test_dos[nombres_atrs].values
x_test_botnet = x_test_botnet[nombres_atrs].values

# Conjuntos para ver cómo generalizan los algoritmos y ver si han
aprendido los ataques
attacks_ssh_full = attacks_ssh_full[nombres_atrs].values
attacks_spam_full = attacks_spam_full[nombres_atrs].values
attacks_dos_full = attacks_dos_full[nombres_atrs].values
attacks_botnet_full = attacks_botnet_full[nombres_atrs].values

import math

idx = math.floor(len(benign) * 70 / 100)
print(idx)
```

```
x_train_benign = (benign.iloc[:idx])[nombres_atrs].values # 720
filas, 70% de los datos, hasta el 13 de Junio
x_test_benign = (benign.iloc[idx:])[nombres_atrs].values # 309 filas,
30% restante y con fechas posteriores a test

from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

pca = PCA(n_components=10)

scaler = StandardScaler()
scaler.fit(x_train_benign)

#TODO: hacerlo con todos, si añadimos nuevos ataques también
x_train_benign = scaler.transform(x_train_benign)
x_train_ssh = scaler.transform(x_train_ssh)
x_test_benign = scaler.transform(x_test_benign)
x_test_ssh = scaler.transform(x_test_ssh)
x_test_spam = scaler.transform(x_test_spam)
x_test_dos = scaler.transform(x_test_dos)
x_test_botnet = scaler.transform(x_test_botnet)
attacks_ssh_full = scaler.transform(attacks_ssh_full)
attacks_spam_full = scaler.transform(attacks_spam_full)
attacks_dos_full = scaler.transform(attacks_dos_full)
attacks_botnet_full = scaler.transform(attacks_botnet_full)

# Tras estandarizar los datos, quedan con media 0 y desviación típica
1 aproximadamente
data = np.concatenate((x_train_benign, x_test_benign))
data = np.concatenate((data, x_train_ssh))
df = pd.DataFrame(data=data,
                  index=data[0:,0],
                  columns=data[0,0:])
df['Class'] = pd.Series([0]*len(benign) + ([1]*len(x_train_ssh)),
index=df.index)

df.describe().T[["count", "min", "max", "mean", "std"]]

pca.fit(x_train_benign)
```

```
x_train_benign = pca.transform(x_train_benign)
x_train_ssh = pca.transform(x_train_ssh)
x_test_benign = pca.transform(x_test_benign)
x_test_ssh = pca.transform(x_test_ssh)
x_test_spam = pca.transform(x_test_spam)
x_test_dos = pca.transform(x_test_dos)
x_test_botnet = pca.transform(x_test_botnet)
attacks_ssh_full = pca.transform(attacks_ssh_full)
attacks_spam_full = pca.transform(attacks_spam_full)
attacks_dos_full = pca.transform(attacks_dos_full)
attacks_botnet_full = pca.transform(attacks_botnet_full)

# Juntamos todo train y creamos y_train con clases
x_train = np.concatenate((x_train_benign, x_train_ssh)) # Benignos
con un 1% de contaminación
#y_train = [1]*len(x_train_benign) + ([-1]*len(x_train_ssh))
y_train_bin = [0]*len(x_train_benign) + ([1]*len(x_train_ssh)) #
0=inlier, 1=outlier

# Creamos test
attacks = np.concatenate((x_test_ssh, x_test_spam, x_test_dos,
x_test_botnet))
x_test = np.concatenate((x_test_benign, attacks))
#y_test = [1]*len(x_test_benign) + ([-1]*(len(attacks)))
y_test_bin = [0]*len(x_test_benign) + ([1]*(len(attacks)))
print(len(attacks))

from sklearn.utils.multiclass import type_of_target
type_of_target(y_train_bin)

"""# Se prueban diferentes clasificadores (f1 metric)"""

# Calcula el score de un modelo en base a las predicciones que
realiza
# sobre los conjuntos de datos benignos (pred1) y malignos (pred2)
# def score(pred1, pred2):
#   # +1 = inlier, -1 = outlier
#   tn = (pred1 == 1).sum()
#   fp = (pred1 != 1).sum()
#   fn = (pred2 == 1).sum()
```

```
#   tp = (pred2 != 1).sum()

#   return round ( ( (tp + tn) / (tp + tn + fp + fn) ) * 100, 2 )


# preds = clf.best_estimator_.predict(x_test)
# print((preds == y_test).mean())

def score(preds, y_test):
 contScore=0
 contFP=0
 contFN=0

 for i in range(0, len(preds)):
   #print("Has predicho {} y era {}".format(preds[i], y_test[i]))
   if preds[i] == y_test[i]:
     contScore+=1
   else:
     if y_test[i] == 0:
       contFP+=1
       print("Falso positivo, no pasa mucho, en indice ", i)
     else:
       contFN+=1
       print("Falso negativo, esto es mas grave, en indice ", i)

 return (contScore / len(y_test), contFP / len(x_test_benign), contFN
/ len(attacks))

from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import cross_val_score
import time

"""## KNN (PyOD)"""

# Buscamos los mejores parametros para k-nn
from pyod.models.knn import KNN

print(KNN().get_params())

parameters = {'n_neighbors': range(1, 33, 2),
```

```
              'metric': ['manhattan', 'euclidean'],
              'contamination': [0.01]
              }

clf = GridSearchCV(KNN(), parameters, cv = 10, scoring='f1_micro')
clf.fit(x_train, y_train_bin)

print("Parametros mejores: " + str(clf.best_params_))
print("Mejor score: " + str(clf.best_score_))

# Ahora testeamos
scores = cross_val_score(clf.best_estimator_, x_train, y_train_bin,
cv=10, scoring='f1_micro')

print("Score {:.2f}".format(scores.mean()))
print("+/- {:.2f}".format(scores.std()))

# Ahora con los datos de test
start = time.time()
preds = clf.best_estimator_.predict(x_test)
end = time.time()

print("{:.8f} seconds to test".format(end - start))
res = score(preds, y_test_bin)
print("Score {:.4f}, FP {:.4f}, FN {:.4f}".format(res[0], res[1],
res[2]))
knn = clf.best_estimator_

"""## LocalOutlierFactor (PyOD)"""

from pyod.models.lof import LOF

parameters = {'n_neighbors': range(3, 41, 2),
              'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
              'metric': ['euclidean', 'manhattan', 'hamming',
'sqeuclidean', 'minkowski'],
              'contamination': [0.01]}


clf = GridSearchCV(LOF(), parameters, cv = 10, scoring='f1_micro')
```

```
clf.fit(x_train, y_train_bin)

print("Parametros mejores: " + str(clf.best_params_))
print("Mejor score: " + str(clf.best_score_))

# Ahora testeamos
start = time.time()
scores = cross_val_score(clf.best_estimator_, x_train, y_train_bin,
cv=10, scoring='f1_micro')
end = time.time()

print("Score {:.2f}".format(scores.mean()))
print("+/- {:.2f}".format(scores.std()))
print("{:.8f} seconds to train".format(end - start))

# Ahora con los datos de test
start = time.time()
preds = clf.best_estimator_.predict(x_test)
end = time.time()
print("{:.8f} seconds to test".format(end - start))
res = score(preds, y_test_bin)
print("Score {:.4f}, FP {:.4f}, FN {:.4f}".format(res[0], res[1],
res[2]))
lof = clf.best_estimator_

"""## OneClassSVM (PyOD)"""

from pyod.models.ocsvm import OCSVM

parameters = {'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],
              'nu': [0.1, 0.001, 0.0001, 0.2, 0.5],
              'degree': [1, 2, 3, 4, 5],
              'contamination': [0.01]}


clf = GridSearchCV(OCSVM(), parameters, cv = 10, scoring='f1_micro')
clf.fit(x_train, y_train_bin)

print("Parametros mejores: " + str(clf.best_params_))
print("Mejor score: " + str(clf.best_score_))
```

```
# Ahora testeamos
start = time.time()
scores = cross_val_score(clf.best_estimator_, x_train, y_train_bin,
cv=10, scoring='f1_micro')
end = time.time()

print("Score {:.2f}".format(scores.mean()))
print("+/- {:.2f}".format(scores.std()))
print("{:.8f} seconds to train".format(end - start))

# Ahora con los datos de test
start = time.time()
preds = clf.best_estimator_.predict(x_test)
end = time.time()
print("{:.8f} seconds to test".format(end - start))
res = score(preds, y_test_bin)
print("Score {:.4f}, FP {:.4f}, FN {:.4f}".format(res[0], res[1],
res[2]))
ocsvm = clf.best_estimator_

"""## Logistic Regression"""

from sklearn.linear_model import LogisticRegression

model = LogisticRegression()

parameters = {'C': [0.01, 0.1, 1, 10, 100],
              'max_iter':[1000],
              "penalty":['l2', 'l1']}

clf = GridSearchCV(model, parameters, cv = 10, scoring="f1_micro")
clf.fit(x_train, y_train_bin)

print("Parametros mejores: " + str(clf.best_params_))

# Ahora testeamos
start = time.time()
scores = cross_val_score(clf.best_estimator_, x_train, y_train_bin,
cv=10, scoring='f1_micro')
```

```
end = time.time()

print("Score {:.2f}".format(scores.mean()))
print("+/- {:.2f}".format(scores.std()))
print("{:.8f} seconds to train".format(end - start))

# Ahora con los datos de test
start = time.time()
preds = clf.best_estimator_.predict(x_test)
end = time.time()
print("{:.8f} seconds to test".format(end - start))
res = score(preds, y_test_bin)
print("Score {:.4f}, FP {:.4f}, FN {:.4f}".format(res[0], res[1],
res[2]))
logReg = clf.best_estimator_

"""## SVC"""

from sklearn.svm import SVC

model = SVC()

parameters = {'C': [0.1, 1, 10, 100, 1000],
              'gamma': [1, 0.1, 0.01, 0.001, 0.0001],
              'kernel': ['rbf','poly','linear','sigmoid'],
              'probability': [True, False]}

clf = GridSearchCV(model, parameters, cv = 10, scoring="f1_micro")
clf.fit(x_train, y_train_bin)

print("Parametros mejores: " + str(clf.best_params_))
print("Mejor score: " + str(clf.best_score_))

# Ahora testeamos
scores = cross_val_score(clf.best_estimator_, x_train, y_train_bin,
cv=10, scoring='f1_micro')

print("Score {:.2f}".format(scores.mean()))
print("+/- {:.2f}".format(scores.std()))
```

```
# Ahora con los datos de test
start = time.time()
preds = clf.best_estimator_.predict(x_test)
end = time.time()
print("{:.8f} seconds to test".format(end - start))
res = score(preds, y_test_bin)
print("Score {:.4f}, FP {:.4f}, FN {:.4f}".format(res[0], res[1],
res[2]))
svc = clf.best_estimator_

"""## COPOD (PyOD)"""

from pyod.models.copod import COPOD

parameters = {'contamination': [0.01]}

clf = GridSearchCV(COPOD(), parameters, cv = 10, scoring='f1_micro')
clf.fit(x_train, y_train_bin)

print("Parametros mejores: " + str(clf.best_params_))
print("Mejor score: " + str(1-clf.best_score_))

# Ahora testeamos
scores = cross_val_score(clf.best_estimator_, x_train, y_train_bin,
cv=10, scoring='f1_micro')

print("Score {:.2f}".format(scores.mean()))
print("+/- {:.2f}".format(scores.std()))

# Ahora con los datos de test
start = time.time()
preds = clf.best_estimator_.predict(x_test)
end = time.time()
print("{:.8f} seconds to test".format(end - start))
res = score(preds, y_test_bin)
print("Score {:.4f}, FP {:.4f}, FN {:.4f}".format(res[0], res[1],
res[2]))
copod = clf.best_estimator_

"""## IsolationForest (PyOD)"""
```

```
from pyod.models.iforest import IForest

parameters = {'n_estimators': range(81, 120, 2),
              'bootstrap': [True, False],
              'contamination': [0.01]}

clf = GridSearchCV(IForest(), parameters, cv = 10,
scoring='f1_micro')
clf.fit(x_train, y_train_bin)

print("Parametros mejores: " + str(clf.best_params_))
print("Mejor score: " + str(1-clf.best_score_))


# Ahora testeamos
start = time.time()
scores = cross_val_score(clf.best_estimator_, x_train, y_train_bin,
cv=10, scoring='f1_micro')
end = time.time()

print("Score {:.2f}".format(scores.mean()))
print("+/- {:.2f}".format(scores.std()))
print("{:.8f} seconds to train".format(end - start))

# Ahora con los datos de test
start = time.time()
preds = clf.best_estimator_.predict(x_test)
end = time.time()
print("{:.8f} seconds to test".format(end - start))
res = score(preds, y_test_bin)
print("Score {:.4f}, FP {:.4f}, FN {:.4f}".format(res[0], res[1],
res[2]))
iforest = clf.best_estimator_

"""## Red neuronal"""

from sklearn.neural_network import MLPClassifier

model = MLPClassifier()
```

```
parameters = {"hidden_layer_sizes":[(2,),(5,),(2,2,),
(5,5,),(5,10,)],
              "alpha":[0.1, 0.25, 0.5, 1],
              "learning_rate_init": [0.0001, 0.001, 0.01, 0.3, 1],
              "max_iter":[100]}


clf = GridSearchCV(model, parameters, cv = 10, scoring="f1_micro")
clf.fit(x_train, y_train_bin)

print("Parametros mejores: " + str(clf.best_params_))
print("Mejor score: " + str(clf.best_score_))

print("Score {:.2f}".format(scores.mean()))
print("+/- {:.2f}".format(scores.std()))

# Ahora testeamos
start = time.time()
scores = cross_val_score(clf.best_estimator_, x_train, y_train_bin,
cv=10, scoring='f1_micro')
end = time.time()

print("Score {:.2f}".format(scores.mean()))
print("+/- {:.2f}".format(scores.std()))
print("{:.8f} seconds to train".format(end - start))

# Ahora con los datos de test
start = time.time()
preds = clf.best_estimator_.predict(x_test)
end = time.time()
print("{:.8f} seconds to test".format(end - start))
res = score(preds, y_test_bin)
print("Score {:.4f}, FP {:.4f}, FN {:.4f}".format(res[0], res[1],
res[2]))
redneuronal = clf.best_estimator_

"""## EllipticEnvelope"""

from sklearn.covariance import EllipticEnvelope
```

```
parameters = {'contamination': [0.01, 'auto'],
              'support_fraction': [None, 0.01, 0.001, 0.1],
              'assume_centered': [True, False]}


y_test_bin = [1]*len(x_test_benign) + ([-1]*(len(attacks)))
y_train_bin = [1]*len(x_train_benign) + ([-1]*len(x_train_ssh)) #
1=inlier, -1=outlier
clf = GridSearchCV(EllipticEnvelope(), parameters, cv = 10,
scoring='f1_micro')
clf.fit(x_train, y_train_bin)

print("Parametros mejores: " + str(clf.best_params_))
print("Mejor score: " + str(clf.best_score_))

# Ahora testeamos
start = time.time()
scores = cross_val_score(clf.best_estimator_, x_train, y_train_bin,
cv=10, scoring='f1_micro')
end = time.time()

print("Score {:.2f}".format(scores.mean()))
print("+/- {:.2f}".format(scores.std()))

# Ahora con los datos de test
start = time.time()
preds = clf.best_estimator_.predict(x_test)
end = time.time()
print("{:.8f} seconds to test".format(end - start))
res = score(preds, y_test_bin)
print("Score {:.4f}, FP {:.4f}, FN {:.4f}".format(res[0], res[1],
res[2]))
ellipticEnvelope = clf.best_estimator_

"""## Arbol de decision"""

# El que mejor relacion score-varianza obtiene
from sklearn.tree import DecisionTreeClassifier
```

```
model = DecisionTreeClassifier()

parameters = {'max_depth': range(1, 15),
              'max_leaf_nodes': range(2, 25)}

clf = GridSearchCV(model, parameters, cv = 10, scoring="f1_micro")
clf.fit(x_train, y_train_bin)

print("Parametros mejores: " + str(clf.best_params_))
print("Mejor score: " + str(clf.best_score_))

# Ahora testeamos
start = time.time()
scores = cross_val_score(clf.best_estimator_, x_train, y_train_bin,
cv=10, scoring='f1_micro')
end = time.time()

print("Score {:.2f}".format(scores.mean()))
print("+/- {:.2f}".format(scores.std()))
print("{:.8f} seconds to train".format(end - start))

# Ahora con los datos de test
start = time.time()
preds = clf.best_estimator_.predict(x_test)
end = time.time()
print("{:.8f} seconds to test".format(end - start))
res = score(preds, y_test_bin)
print("Score {:.4f}, FP {:.4f}, FN {:.4f}".format(res[0], res[1],
res[2]))
decisionTree = clf.best_estimator_

"""# Vemos qué tal predicen ataques únicamente"""

def score(preds, y_test):
 contScore=0

 for i in range(0, len(preds)):
   if preds[i] == y_test[i]:
     contScore+=1
 print("Score = {:.4f}".format(contScore / len(y_test)))
```

```
print("SSH: ", len(attacks_ssh_full))
print("Spam: ", len(attacks_spam_full))
print("DoS: ", len(attacks_dos_full))
print("Botnet: ", len(attacks_botnet_full))

"""## Ataques spam"""

preds = knn.predict(attacks_spam_full)
score(preds, [1]*len(attacks_spam_full))

preds = lof.predict(attacks_spam_full)
score(preds, [1]*len(attacks_spam_full))

preds = ocsvm.predict(attacks_spam_full)
score(preds, [1]*len(attacks_spam_full))

preds = logReg.predict(attacks_spam_full)
score(preds, [1]*len(attacks_spam_full))

preds = svc.predict(attacks_spam_full)
score(preds, [1]*len(attacks_spam_full))

preds = copod.predict(attacks_spam_full)
score(preds, [1]*len(attacks_spam_full))

preds = iforest.predict(attacks_spam_full)
score(preds, [1]*len(attacks_spam_full))

preds = redneuronal.predict(attacks_spam_full)
score(preds, [1]*len(attacks_spam_full))

preds = ellipticEnvelope.predict(attacks_spam_full)
score(preds, [-1]*len(attacks_spam_full))

preds = decisionTree.predict(attacks_spam_full)
score(preds, [1]*len(attacks_spam_full))

"""## Ataques ssh"""
```

```
preds = knn.predict(attacks_ssh_full)
score(preds, [1]*len(attacks_ssh_full))

preds = lof.predict(attacks_ssh_full)
score(preds, [1]*len(attacks_ssh_full))

preds = ocsvm.predict(attacks_ssh_full)
score(preds, [1]*len(attacks_ssh_full))

preds = logReg.predict(attacks_ssh_full)
score(preds, [1]*len(attacks_ssh_full))

preds = svc.predict(attacks_ssh_full)
score(preds, [1]*len(attacks_ssh_full))

preds = copod.predict(attacks_ssh_full)
score(preds, [1]*len(attacks_ssh_full))

preds = iforest.predict(attacks_ssh_full)
score(preds, [1]*len(attacks_ssh_full))

preds = redneuronal.predict(attacks_ssh_full)
score(preds, [1]*len(attacks_ssh_full))

preds = ellipticEnvelope.predict(attacks_ssh_full)
score(preds, [-1]*len(attacks_ssh_full))

preds = decisionTree.predict(attacks_ssh_full)
score(preds, [1]*len(attacks_ssh_full))

"""## Ataques dos"""

preds = knn.predict(attacks_dos_full)
score(preds, [1]*len(attacks_dos_full))

preds = lof.predict(attacks_dos_full)
score(preds, [1]*len(attacks_dos_full))

preds = ocsvm.predict(attacks_dos_full)
score(preds, [1]*len(attacks_dos_full))
```

```
preds = logReg.predict(attacks_dos_full)
score(preds, [1]*len(attacks_dos_full))

preds = svc.predict(attacks_dos_full)
score(preds, [1]*len(attacks_dos_full))

preds = copod.predict(attacks_dos_full)
score(preds, [1]*len(attacks_dos_full))

preds = iforest.predict(attacks_dos_full)
score(preds, [1]*len(attacks_dos_full))

preds = redneuronal.predict(attacks_dos_full)
score(preds, [1]*len(attacks_dos_full))

preds = ellipticEnvelope.predict(attacks_dos_full)
score(preds, [-1]*len(attacks_dos_full))

preds = decisionTree.predict(attacks_dos_full)
score(preds, [1]*len(attacks_dos_full))

"""## Ataques botnet"""

preds = knn.predict(attacks_botnet_full)
score(preds, [1]*len(attacks_botnet_full))

preds = lof.predict(attacks_botnet_full)
score(preds, [1]*len(attacks_botnet_full))

preds = ocsvm.predict(attacks_botnet_full)
score(preds, [1]*len(attacks_botnet_full))

preds = logReg.predict(attacks_botnet_full)
score(preds, [1]*len(attacks_botnet_full))

preds = svc.predict(attacks_botnet_full)
score(preds, [1]*len(attacks_botnet_full))

preds = copod.predict(attacks_botnet_full)
```

```
score(preds, [1]*len(attacks_botnet_full))

preds = iforest.predict(attacks_botnet_full)
score(preds, [1]*len(attacks_botnet_full))

preds = redneuronal.predict(attacks_botnet_full)
score(preds, [1]*len(attacks_botnet_full))

preds = ellipticEnvelope.predict(attacks_botnet_full)
score(preds, [-1]*len(attacks_botnet_full))

preds = decisionTree.predict(attacks_botnet_full)
score(preds, [1]*len(attacks_botnet_full))

"""# Equilibramos el conjunto de datos"""

from collections import Counter
from imblearn.over_sampling import SMOTE

print('Original dataset shape %s' % Counter(y_train_bin))
sm = SMOTE(random_state=42)
x_train, y_train_bin = sm.fit_resample(x_train, y_train_bin)
print('Resampled dataset shape %s' % Counter(y_train_bin))

"""# Probamos de nuevo modelos supervisados"""

from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import cross_val_score
import time

def score(preds, y_test):
 contScore=0
 contFP=0
 contFN=0

 for i in range(0, len(preds)):
   #print("Has predicho {} y era {}".format(preds[i], y_test[i]))
   if preds[i] == y_test[i]:
     contScore+=1
   else:
```

```
      if y_test[i] == 0:
        contFP+=1
        print("Falso positivo, no pasa mucho, en indice ", i)
      else:
        contFN+=1
        print("Falso negativo, esto es mas grave, en indice ", i)

 return (contScore / len(y_test), contFP / len(x_test_benign), contFN
/ len(attacks))

"""## Logistic Regression"""

from sklearn.linear_model import LogisticRegression

model = LogisticRegression()

parameters = {'C': [0.01, 0.1, 1, 10, 100],
              'max_iter':[1000],
              "penalty":['l2', 'l1']}

clf = GridSearchCV(model, parameters, cv = 10, scoring="f1_micro")
clf.fit(x_train, y_train_bin)

print("Parametros mejores: " + str(clf.best_params_))

# Ahora testeamos
start = time.time()
scores = cross_val_score(clf.best_estimator_, x_train, y_train_bin,
cv=10, scoring='f1_micro')
end = time.time()

print("Score {:.2f}".format(scores.mean()))
print("+/- {:.2f}".format(scores.std()))
print("{:.8f} seconds to train".format(end - start))

# Ahora con los datos de test
start = time.time()
preds = clf.best_estimator_.predict(x_test)
end = time.time()
print("{:.8f} seconds to test".format(end - start))
```

```
res = score(preds, y_test_bin)
print("Score {:.4f}, FP {:.4f}, FN {:.4f}".format(res[0], res[1],
res[2]))
logReg = clf.best_estimator_

"""## SVC"""

from sklearn.svm import SVC

model = SVC()

parameters = {'C': [0.1, 1, 10, 100, 1000],
              'gamma': [1, 0.1, 0.01, 0.001, 0.0001],
              'kernel': ['rbf','poly','linear','sigmoid'],
              'probability': [True, False]}

clf = GridSearchCV(model, parameters, cv = 10, scoring="f1_micro")
clf.fit(x_train, y_train_bin)

print("Parametros mejores: " + str(clf.best_params_))
print("Mejor score: " + str(clf.best_score_))

# Ahora testeamos
scores = cross_val_score(clf.best_estimator_, x_train, y_train_bin,
cv=10, scoring='f1_micro')

print("Score {:.2f}".format(scores.mean()))
print("+/- {:.2f}".format(scores.std()))

# Ahora con los datos de test
start = time.time()
preds = clf.best_estimator_.predict(x_test)
end = time.time()
print("{:.8f} seconds to test".format(end - start))
res = score(preds, y_test_bin)
print("Score {:.4f}, FP {:.4f}, FN {:.4f}".format(res[0], res[1],
res[2]))
svc = clf.best_estimator_

"""## Red neuronal"""
```

```
from sklearn.neural_network import MLPClassifier

model = MLPClassifier()

parameters = {"hidden_layer_sizes":[(2,),(5,),(2,2,),
(5,5,),(5,10,)],
              "alpha":[0.1, 0.25, 0.5, 1],
              "learning_rate_init": [0.0001, 0.001, 0.01, 0.3, 1],
              "max_iter":[100]}


clf = GridSearchCV(model, parameters, cv = 10, scoring="f1_micro")
clf.fit(x_train, y_train_bin)

print("Parametros mejores: " + str(clf.best_params_))
print("Mejor score: " + str(clf.best_score_))

print("Score {:.2f}".format(scores.mean()))
print("+/- {:.2f}".format(scores.std()))

# Ahora testeamos
start = time.time()
scores = cross_val_score(clf.best_estimator_, x_train, y_train_bin,
cv=10, scoring='f1_micro')
end = time.time()

print("Score {:.2f}".format(scores.mean()))
print("+/- {:.2f}".format(scores.std()))
print("{:.8f} seconds to train".format(end - start))

# Ahora con los datos de test
start = time.time()
preds = clf.best_estimator_.predict(x_test)
end = time.time()
print("{:.8f} seconds to test".format(end - start))
res = score(preds, y_test_bin)
print("Score {:.4f}, FP {:.4f}, FN {:.4f}".format(res[0], res[1],
res[2]))
redneuronal = clf.best_estimator_
```

```
from sklearn.metrics import confusion_matrix
tn, fp, fn, tp = confusion_matrix(y_test_bin, preds).ravel()
(tn, fp, fn, tp)
# Sus falsos positivos son mis falsos negativos

"""## Arbol de decision"""

# El que mejor relacion score-varianza obtiene
from sklearn.tree import DecisionTreeClassifier

model = DecisionTreeClassifier()

parameters = {'max_depth': range(1, 15),
              'max_leaf_nodes': range(2, 25)}

clf = GridSearchCV(model, parameters, cv = 10, scoring="f1_micro")
clf.fit(x_train, y_train_bin)

print("Parametros mejores: " + str(clf.best_params_))
print("Mejor score: " + str(clf.best_score_))

# Ahora testeamos
start = time.time()
scores = cross_val_score(clf.best_estimator_, x_train, y_train_bin,
cv=10, scoring='f1_micro')
end = time.time()

print("Score {:.2f}".format(scores.mean()))
print("+/- {:.2f}".format(scores.std()))
print("{:.8f} seconds to train".format(end - start))

# Ahora con los datos de test
start = time.time()
preds = clf.best_estimator_.predict(x_test)
end = time.time()
print("{:.8f} seconds to test".format(end - start))
res = score(preds, y_test_bin)
print("Score {:.4f}, FP {:.4f}, FN {:.4f}".format(res[0], res[1],
res[2]))
```

```
decisionTree = clf.best_estimator_

"""# Vemos qué tal predicen ataques únicamente"""

def score(preds, y_test):
 contScore=0

 for i in range(0, len(preds)):
   if preds[i] == y_test[i]:
     contScore+=1
 print("Score = {:.4f}".format(contScore / len(y_test)))

print("SSH: ", len(attacks_ssh_full))
print("Spam: ", len(attacks_spam_full))
print("DoS: ", len(attacks_dos_full))
print("Botnet: ", len(attacks_botnet_full))

"""## Ataques spam"""

preds = logReg.predict(attacks_spam_full)
score(preds, [1]*len(attacks_spam_full))

preds = svc.predict(attacks_spam_full)
score(preds, [1]*len(attacks_spam_full))

preds = redneuronal.predict(attacks_spam_full)
score(preds, [1]*len(attacks_spam_full))

preds = ellipticEnvelope.predict(attacks_spam_full)
score(preds, [1]*len(attacks_spam_full))

preds = decisionTree.predict(attacks_spam_full)
score(preds, [1]*len(attacks_spam_full))

"""## Ataques ssh"""

preds = logReg.predict(attacks_ssh_full)
score(preds, [1]*len(attacks_ssh_full))

preds = svc.predict(attacks_ssh_full)
```

```
score(preds, [1]*len(attacks_ssh_full))

preds = redneuronal.predict(attacks_ssh_full)
score(preds, [1]*len(attacks_ssh_full))

preds = ellipticEnvelope.predict(attacks_ssh_full)
score(preds, [1]*len(attacks_ssh_full))

preds = decisionTree.predict(attacks_ssh_full)
score(preds, [1]*len(attacks_ssh_full))

"""## Ataques dos"""

preds = logReg.predict(attacks_dos_full)
score(preds, [1]*len(attacks_dos_full))

preds = svc.predict(attacks_dos_full)
score(preds, [1]*len(attacks_dos_full))

preds = redneuronal.predict(attacks_dos_full)
score(preds, [1]*len(attacks_dos_full))

preds = ellipticEnvelope.predict(attacks_dos_full)
score(preds, [1]*len(attacks_dos_full))

preds = decisionTree.predict(attacks_dos_full)
score(preds, [1]*len(attacks_dos_full))

"""## Ataques botnet"""

preds = logReg.predict(attacks_botnet_full)
score(preds, [1]*len(attacks_botnet_full))

preds = svc.predict(attacks_botnet_full)
score(preds, [1]*len(attacks_botnet_full))

preds = redneuronal.predict(attacks_botnet_full)
score(preds, [1]*len(attacks_botnet_full))

preds = ellipticEnvelope.predict(attacks_botnet_full)
```

```
score(preds, [1]*len(attacks_botnet_full))

preds = decisionTree.predict(attacks_botnet_full)
score(preds, [1]*len(attacks_botnet_full))
```