Universitat Oberta de Catalunya

uoc.edu

Técnicas de Aprendizaje Automático para la detección de ataques en el tráfico de Red

Andrés Valencia

Máster en Seguridad de las Tecnologías de Información y Comunicación

Área de Análisis de Datos

Nombre Consultor/a : Enric Hernández Jiménez

Nombre Profesor/a responsable de la asignatura : Helena Rifà Pous

04/06/2019

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Técnicas de Aprendizaje Automático para la detección de ataques en el tráfico de Red*
Nombre del autor:	*Andres Valencia Peral*
Nombre del consultor/a:	*Enric Hernández Jiménez*
Nombre del PRA:	*Helena Rifà Pous*
Fecha de entrega (mm/aaaa):	06/2019
Titulación::	*Master en Seguridad en las Telecomunicaciones y Tecnologías de la Información)*
Área del Trabajo Final:	*Análisis de Datos*
Idioma del trabajo:	*Castellano*
Palabras clave	*Machine Learning, Intrusion Detection System, Deep Learning*

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados i conclusiones del trabajo.*

El proyecto realizado tiene como finalidad de realizar un análisis general de las técnicas de aprendizaje automático actualmente disponibles aplicadas a la implementación de un sistema de detección de intrusión de red. Se describirán de forma sencilla las diferentes técnicas existentes hoy en día, poniendo de relieve las diferencias fundamentales entre las técnicas de Machine Learning clásicas en contraposición a las de Deep Learning utilizando Redes Neuronales, así como la relación subyacente entre ellas.

Realizaremos un estudio mas exhaustivo de una de las técnicas de Aprendizaje automático, el Árbol de decisión, y finalmente nos ocuparemos con cierto detalle de los aspectos a tener en cuenta en la implementación de una red neuronal para acometer el problema, con especial foco en la elección de los hiperparámetros de entrenamiento y las consecuencias que tales decisiones acarrean.

Concluiremos que por la naturaleza del problema planteado, que dispone de conjuntos de muestras extremadamente abundantes para poder entrenar los modelos deseados, la aplicación de dichas técnicas ofrece una ventaja decisiva frente a otras técnicas tradicionales basadas en reglas y firmas.

Abstract (in English, 250 words or less):

The purpose of the project will be to carry out a general analysis of the currently available automatic learning techniques applied to the implementation of a network intrusion detection system. The different techniques existing today will be described in a simple way, highlighting the fundamental differences between classical Machine Learning techniques as opposed to those of Deep Learning using Neural Networks, as well as the underlying relationship between them.

We will proceed with a more exhaustive study of one of the techniques of automatic learning, the decision tree, and finally we will deal with some detail of the aspects to be taken into account in the implementation of a neural network to tackle the problem, with special focus on the choice of training hyperparameters and the consequences that such decisions entail.

We will conclude that due to the nature of the problem posed, which has extremely abundant sample sets to be used to train the desired models, the application of these techniques offers a decisive advantage over other traditional techniques based on rules and signatures.

Índice

1. Plan de Trabajo. 6
1.1 Introducción detallada del problema a resolver. 6
1.2 La enumeración de los objetivos que se quieren alcanzar con la realización del TFM. 8
1.3 La descripción de la metodología que se seguirá durante el desarrollo del TFM. 9
1.4 El listado de las tareas a realizar para alcanzar los objetivos descritos. 10
1.5 La planificación temporal detallada de estas tareas y sus dependencias. 11
1.6 Consideraciones adicionales 13
2. Analisis de las Tecnicas ML escogidas 14
2. Documentación 27
2.1 Definición de Machine Learning 14
2.2 Aprendizaje Supervisado vs. No Supervisado 14
2.3 Diferentes Aplicaciones para el Aprendizaje automático: 14
2.3.1 Clasificación 14
2.3.2 Regresión 15
2.3.3 Agrupación (Clustering) 16
2.3.4 Reducción de la dimensionalidad 16
2.3.5 Detección de Anomalías 17
2.4 Deep Learning 18
2.4.1 Redes Neuronales 19
2.4.2 Aplicaciones de las Redes Neuronales Artificiales 19
2.4.3 Overfitting 21
2.5 Esquema general para la elaboración de modelos de aprendizaje automático 21
2.6 Preparación del conjunto de Datos a analizar 22
3 Elaboración 23
3.1 Definición del Problema: Implementación de un Sistema de Detección de Intrusión. 23
3.2 Estructura del DataSet 24
3.3 ML vs DL 26
3.4 Arboles de decision 26
3.4 Implementación del Árbol de decisión 27
4.0 Implementación Avanzada. 35
4.1 Implementaciones con Machine Learning: 36
4.1.1. Arbol de decision 36
4.1.2 Logistic regression 37
4.1.3. K-Nearest Neighbours 39
4.1.4. Naive Bayes 40
4.1.5. Adaboost 41
4.1.6 Random Forests 43
4.1.7. Resultados utilizando técnicas de Machine Learning. 44
4.2. Implementación mediante una Red Neuronal 46
4.3 Generacion de una Matriz de Redes Neuronales 54
4.4. Interpretación de los resultados. 60
4.4.1 Resultados para la función de activación lineal: 60
4.4.2. Función de activación relu (Rectified Linear Unit): 61
4.4.3 Numero de epochs = 3: 63
4.4.4 Numero de epochs = 5: 64
4.4.5 Numero de epochs = 10: 65
4.4.6. Numero de neuronas de la capa oculta = 4 : 66
4.4.7. Numero de neuronas en la capa Oculta = 10 67
4.4.8. Numero de neuronas en la capa Oculta = 25 67
4.9. Normalizacion del conjunto de features 68
6. Conclusiones 70
7. Siguientes pasos 71
8. Correcciones 72
A. Documentación 79
A.1 Análisis de los recursos online disponibles 79
A.2 Análisis de la Bibliografía disponible 80
A.3 Selección de las herramientas necesarias 81
A.3.1 IDE y entorno de programación 81
A.4 Aprendizaje de las herramientas necesarias 83
A.4.1 Aprendizaje del Lenguaje de Programación: Python 3.7. 83
A.4.2 Aprendizaje de la API 84

1. Introduccion.

1.1 Introducción detallada del problema a resolver.

Para la elaboración de este proyecto final de Máster nos centramos en el análisis e implementación del aprendizaje automático aplicado al ámbito de la seguridad informática.

Es difícil encontrar un campo de las ciencias de la información en el que se haya producido una expansión tan rápida como recientemente en el de las técnicas de aprendizaje automático. En los últimos diez años, la adopción de dichas técnicas para afrontar problemas de la más diversa índole ha tenido lugar de forma fluida y continua, hasta participar hoy en día en prácticamente todos los ámbitos de nuestra interacción con sistemas de información.

El cuerpo de conocimiento llamado usualmente "Inteligencia Artificial" no es ni nuevo ni monolítico. Ya en su momento a principios de la década de los ochenta, algunas aplicaciones de sistemas expertos abrieron la puerta a la esperanza de conseguir desarrollar agentes inteligentes cuyas aplicaciones y flexibilidad pudieran con el tiempo afrontar una multitud de problemas diferentes, definidos de una manera no estática, y tener éxito en estos también.

Pero lo cierto es que, tras unos inicios prometedores, la disciplina no ofreció el desarrollo y rendimientos esperados, lo que más tarde se daría a conocer como el *invierno de la IA*.* Durante algo más de una década la percepción por parte de las burocracias gubernamentales y capitalistas de riesgo es que ni se hacían progresos ni se generaban aplicaciones que dieran a entender que la inversión en dichas tecnologías pudieran ser todavía determinantes en el desarrollo de la sociedad de la información.

No obstante, durante todo ese tiempo se sentaban las bases para la siguiente revolución en el campo de los agentes inteligentes: ya a mediados de la década de 1980, el procesamiento distribuido en paralelo para la resolución de problemas de IA se haría popular con el nombre de conexionismo. Rumelhart y McClelland (1986) describieron el uso del conexionismo para simular procesos neurales. Dichos avances, unidos a la presencia de chips para procesamiento paralelo masivo, capaces de obtener extraordinarios resultados en los cálculos necesarios para implementar los nuevos algoritmos, habilitarían el renacimiento de la disciplina.

Si bien parte de las razones que nos han llevarían a este advenimiento del aprendizaje automático tiene mucho que ver con las posibilidades que ese nuevo y potente hardware ofrecen a día de hoy, para realizar los cálculos necesarios con una velocidad inimaginable hasta hace poco y por una fracción de su coste, es sin duda la recolección de cantidades ingentes de conjuntos de datos mediante internet la que realmente lo posibilita. Con la unión de ambas circunstancias la posibilidad de desarrollar las técnicas de Machine Learning se materializa en ámbitos tan diversos como el procesamiento del lenguaje natural, el reconocimiento visual, o la conducción autónoma, campos todos ellos en los que su aplicación ha supuesto una revolución en toda regla.

En el caso que nos ocupa, aplicaremos técnicas de aprendizaje automático a un problema relacionado con la seguridad informática: la detección de intrusos. En todo sistema informático de seguridad perimetral, una de las preguntas a responder por la solución implementada consiste en "*Pueden las técnicas de análisis automático indicar por sí mismas si se está sufriendo una intrusión en nuestra red?"* Utilizaremos las técnicas anteriormente citadas para contestar la pregunta,

contrastando cual es el grado de confianza y coste en función de la utilización de dos técnicas diferentes.

1.2 La enumeración de los objetivos que se quieren alcanzar con la realización del TFM.

Los objetivos a alcanzar mediante la realización del TFM son:

- Familiarizarnos de manera general con las diferentes técnicas de aprendizaje automático y de sus diferentes fortalezas y debilidades en su aplicación a diferentes problemas, y específicamente al problema de clasificación de paquetes de trafico de red en un IDS.
- Desarrollar un conocimiento más detallado sobre dos de dichas técnicas: El árbol de decisión en el ámbito de Machine Learning clásico y las redes neuronales en el ámbito del Deep Learning. Asimismo, nuestro objetivo será también contrastar las ventajas y desventajas de ambas aproximaciones, poniéndolas en contexto en relación con su dificultad de implementación e infraestructura necesaria.
- Desarrollar un conocimiento suficiente de las tecnologías de soporte necesarias para el correcto desarrollo del proyecto (Bases de Datos NoSQL y otros)
- Desarrollar un conocimiento práctico acerca del lenguaje de programación Python, que nos permita a acceder a la programación del producto final.
- Obtener un grado de conocimiento aplicado suficiente de la librería Tensorflow y de su integración con el lenguaje de programación Python y CUDA.
- Por último y como objetivo final, evaluar cuales son los desafíos de integrar las diferentes tecnologías anteriormente citadas para la resolución del problema planteado y similares: selección y tratamiento del conjunto de datos, diferentes estrategias para el entrenamiento de una red neuronal, y eficacia de las distintas aproximaciones en diferentes contextos.

1.3 La descripción de la metodología que se seguirá durante el desarrollo del TFM.

Formación y análisis inicial.

Análisis de las fuentes disponibles, documentación y conclusiones.

Selección definitiva de los algoritmos a utilizar y su justificación.

Selección definitiva de las tecnologías a utilizar y su justificación.

Análisis de requerimientos hardware.

Desarrollo.

Recolección, clasificación y preparación del DataSet.

Prototipado y testeo de la aplicación.

Entrenamiento de la red neuronal y obtención de primeros resultados.

Evaluación y ajustes de forma iterativa hasta obtener los resultados deseados en materia de confiabilidad de la predicción obtenida.

Elaboración de la versión final y valoración de la eficacia de las predicciones obtenidas para distintos conjuntos de datos y en función de distintos parámetros (porcentaje de efectividad, porcentaje de falsos positivos, velocidad de convergencia en las soluciones, necesidades de procesamiento asociadas, necesidades en la selección de datasets iniciales).

Documentación.

Elaboración de una memoria de todo el proyecto donde se recoja tanto su elaboración detallada como sus conclusiones.

1.4 El listado de las tareas a realizar para alcanzar los objetivos descritos.

Documentación sobre las diferentes tecnologías a implementar para poder abordar las necesidades del proyecto: ML en general y Arboles de decisión en particular, DL: Redes Neuronales en general y Multilayered Perceptron Network/Bayesian en particular. Lenguaje de programación Python y librerías necesarias. Tensorflow. Tecnologías auxiliares como MongoDB para NoSQL.

Obtención, tratamiento y clasificación del conjunto de datos.

Elaboración de una aplicación en Python implementando las librerías necesarias para la ejecución de una técnica de ML (arboles de decisión) sobre dicho conjunto de datos.

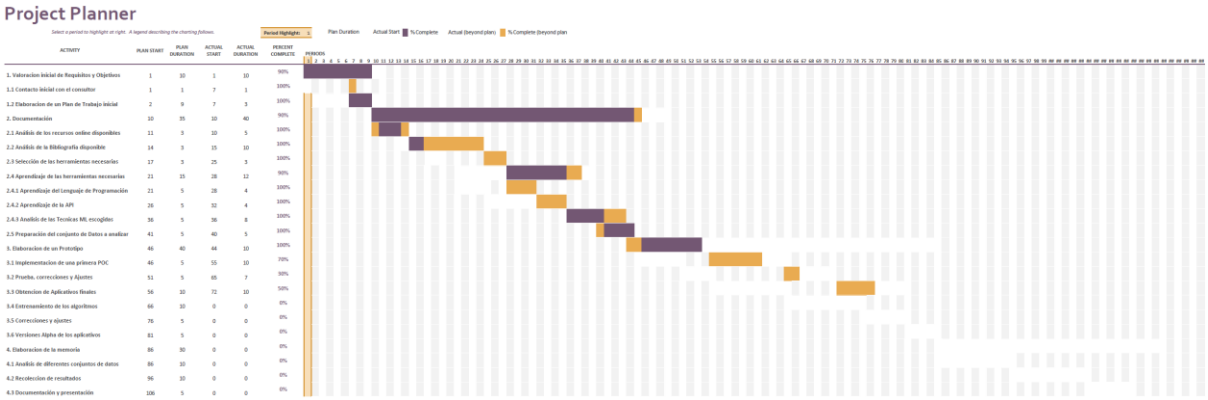
Elaboración de una aplicación en Python implementando las librerías necesarias para la ejecución de una técnica de DL (Red Neuronal) sobre dicho conjunto de datos.

Comparación inicial de resultados, ajuste iterativo de parámetros, análisis de los resultados obtenidos.

Elaboración de las versiones finales de ambas aplicaciones y pruebas aplicadas a conjuntos de datos adicionales.

Análisis de diferentes métricas. Elaboración de Conclusiones.

1.5 La planificación temporal detallada de estas tareas y sus dependencias.



Junto a cada entrega se adjuntará el documento xls con el análisis de Gannt, en el que se definen los diferentes hitos:

ACTIVITY	PLAN START	PLAN DURATION	ACTUAL START	ACTUAL DURATION	PERCENT COMPLETE
1. Valoracion inicial de Requisitos y Objetivos	1	10	1	10	90%
1.1 Contacto inicial con el consultor	1	1	7	1	100%
1.2 Elaboracion de un Plan de Trabajo inicial	2	9	7	3	100%
2. Documentación	10	35	10	40	90%
2.1 Análisis de los recursos online disponibles	11	3	10	5	100%
2.2 Análisis de la Bibliografía disponible	14	3	15	10	100%
2.3 Selección de las herramientas necesarias	17	3	25	3	100%
2.4 Aprendizaje de las herramientas necesarias	21	15	28	12	90%
2.4.1 Aprendizaje del Lenguaje de Programación	21	5	28	4	100%
2.4.2 Aprendizaje de la API	26	5	32	4	100%
2.4.3 Analisis de las Tecnicas ML escogidas	36	5	36	8	100%
2.5 Preparación del conjunto de Datos a analizar	41	5	40	5	100%
3. Elaboracion de un Prototipo	46	40	44	10	100%
3.1 Implementacion de una primera POC	46	5	55	10	70%
3.2 Prueba, correcciones y Ajustes	51	5	65	7	30%
3.3 Obtencion de Aplicativos finales	56	10	72	10	50%
3.4 Entrenamiento de los algoritmos	66	10	0	0	50%
3.5 Correcciones y ajustes	76	5	0	0	30%
3.6 Versiones Alpha de los aplicativos	81	5	0	0	50%
4. Elaboracion de la memoria	86	30	0	0	65%
4.1 Analisis de diferentes conjuntos de datos	86	10	0	0	50%
4.2 Recoleccion de resultados	96	10	0	0	50%
4.3 Documentación y presentación	106	5	0	0	65%

1.6 Consideraciones adicionales

Para la obtención de los conjuntos de datos necesarios para el desarrollo del proyecto tendremos en cuenta las consideraciones relativas a la licencia bajo las que se hayan publicado, atendiendo los requisitos establecidos por esta en los casos en que sean necesarios.

El coste económico de la elaboración del proyecto es a priori nulo en cuanto a la adquisición de material o licencias de hardware adicional. Se dispone de una máquina de escritorio con una CPU 8700K y una GPU 1080 que deberían bastar para cubrir los objetivos del proyecto. En cuanto al coste de licencias para acometer el desarrollo, al tratarse de proyectos de open source serán nulos también.

La dedicación temporal que podemos dar al proyecto es de unas 20 horas por semana, divididas entre 10 horas entre semana y 10 horas en fines de semana. Con las salvedades que se desprendan de necesidades puntuales de entrega a otros proyectos laborales, esta planificación nos ofrece las 300 horas de las que consta la asignatura.

La planificación se reevaluará con el consultor de manera dinámica para poder valorar mejor si la carga de trabajo inicial se ajusta a las necesidades del proyecto, pudiendo ser modificada a conveniencia de ambas partes hasta como máximo el inicio de la fase de prototipado (Día 45 de 116). Si se van cumpliendo los milestones de forma fluida se podrá acordar la implementación de un tercer aplicativo que intente combinar los métodos de ML y DL para generar mejores resultados. Si por el contrario las entregas se producen con dificultad o retraso, podrá redefinirse un objetivo final menos ambicioso, a convenir con el consultor.

Se asigna un margen de 10 días en la planificación del proyecto para poder hacer frente a bajas o imprevistos.

2. Analisis de las Tecnicas ML escogidas

2.1 Definición de Machine Learning

El aprendizaje automático (Machine Learning) puede definirse como el proceso de usar un conjunto de datos históricos (DataSet) compuesto de un numero suficientemente elevado de registros (Data Points) definidos en base a diferentes valores (Features) para crear un modelo de predicción para futuros datos basándonos en un algoritmo.

2.2 Aprendizaje Supervisado vs. No Supervisado

El aprendizaje automático puede ser supervisado (supervised), en cuyo caso cada uno de los registros del conjunto de datos tiene una etiqueta (label) que define la propiedad sobre la cual deseamos que nuestro modelo realice su predicción para los datos futuros.

Por ejemplo, dado un numero lo suficientemente elevado de correos electrónicos etiquetados como spam o no, podremos entrenar a un clasificador (Classifier) de spam que intente predecir si un nuevo mensaje entrante es spam.

Alternativamente, el aprendizaje automático puede ser sin supervisión (unsupervised), en cuyo caso el DataSet suministrado para la elaboración del modelo no dispone de estas etiquetas. Incluso, puede darse el caso de no saber cuáles son las etiquetas que se están tratando predecir, por ejemplo, si analizamos el tráfico de un sensor de red en el que sospechamos que se encuentran presentes un número indeterminado de hosts infectados por diferentes tipos de Malware que generen tráfico hacia un número desconocido de botnets .

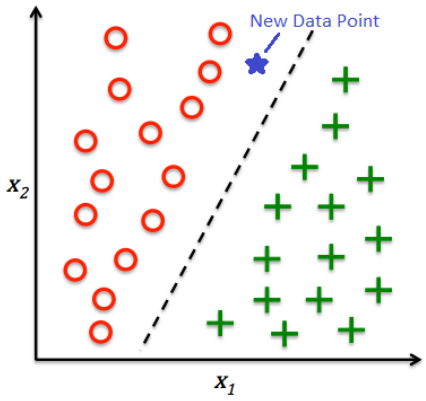
2.3 Diferentes Aplicaciones para el Aprendizaje automático:

2.3.1 Clasificación

Los algoritmos de Machine Learning que se definen como de Clasificación (Classifiers) tienen como objetivo predecir con la máxima precisión la categoría de un DataPoint no visto por el modelo anteriormente.

La clasificación es una subcategoría de aprendizaje supervisado donde el objetivo es predecir las etiquetas de clase categóricas (valores discretos, no ordenados, pertenencia a grupos) de nuevas instancias basadas en observaciones pasadas. Hemos citado anteriormente el ejemplo típico de la detección de correo electrónico no deseado, que es una clasificación binaria (ya sea un correo electrónico es -1- o no es -0- correo no deseado). Naturalmente, también existen métodos de clasificación en múltiples clases, como el clásico ejemplo del reconocimiento de cifras manuscritas (donde las clases van de 0 a 9).

Un ejemplo grafico de clasificación binaria: hay 2 clases, círculos y cruces, y 2 características, X1 y X2. El modelo puede encontrar la relación entre las características de cada punto de datos y su clase, y establecer una línea de límite entre ellas, de modo que cuando se le proporcionen nuevos datos, pueda estimar la clase a la que pertenece, dadas sus características.

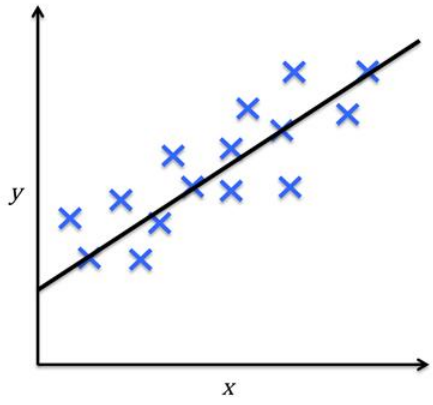


Naturalmente, este ejemplo es solo la base sobra la cual podemos entender cómo funciona un clasificador. Para el caso general, matemáticamente podemos entenderlo como un espacio de Dimension n, donde n corresponde al número de features de cada datapoint, describiendo cada uno de ellos un punto en él. El clasificador puede visualizarse como un Hiperplano que divide dicho espacio en las categorías a predecir, siendo este una función tal que $f(\alpha)+\beta$, siendo f lineal o no según la técnica de resolución empleada por el algoritmo que genere el modelo.

Naturalmente si el número de categorías es mayor que dos, un solo hiperplano no podrá clasificar dicho espacio necesitándose más.

2.3.2 Regresión

La regresión también se utiliza para asignar categorías a datos sin etiquetar. En este tipo de aprendizaje, recibimos una serie de variables predictoras (explicativas) y una variable de respuesta continua (resultado), y tratamos de encontrar una relación entre esas variables que nos permita predecir un resultado continuo.

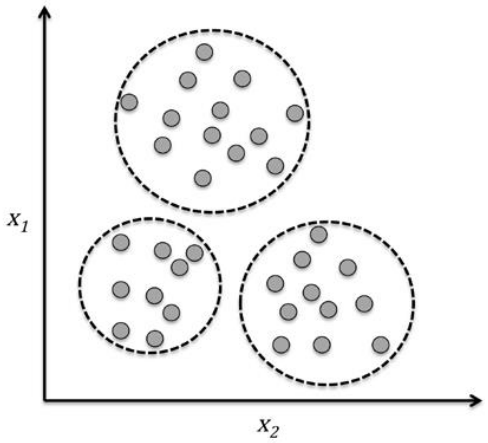


Un ejemplo de regresión lineal: dados X e Y, ajustamos una línea recta que minimiza la distancia (con algunos criterios como la distancia al cuadrado promedio (SSE)) entre los puntos de muestra y la línea ajustada. Luego, usaremos la intercepción y la pendiente aprendidas, de la línea ajustada, para predecir el resultado de los nuevos datos.

Al igual que en caso anterior en aplicaciones de clasificación, para aplicaciones en las que el número de features de cada DataPoint sea superior a dos, debemos extrapolar el ejemplo citado a un entorno de dimensionalidad más elevada donde la relación entre los predictores y la respuesta buscada se expresará mediante un hiperplano, cuya definición puede ser no lineal.

2.3.3 Agrupación (Clustering)

La agrupación es una técnica de análisis exploratorio de datos utilizada para organizar la información en agrupaciones o subgrupos significativos sin ningún conocimiento previo de su estructura. Naturalmente, en base a esta definición estamos hablando de un tipo de aprendizaje automático no supervisado. Cada grupo es un grupo de objetos similares que es diferente a los objetos de los otros grupos.

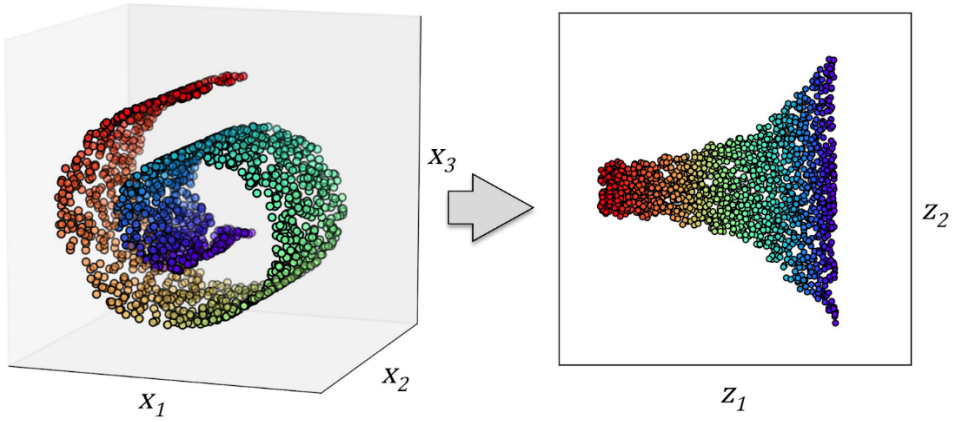


En general, se usa como un proceso para encontrar una estructura significativa, procesos subyacentes explicativos, características generativas y agrupaciones inherentes a un conjunto de datos.

Las técnicas de Clustering son muy importantes, ya que determinan la agrupación intrínseca entre los datos sin etiquetar presentes. Es importante destacar que no existen criterios unívocos z aplicables de forma general para una buena agrupación. Dependerá del analista el decidir cuáles son los criterios que pueden usar para satisfacer las necesidades de cada problema a analizar, siendo necesario cierto grado de ensayo y error para poder ajustar los parámetros elegidos y obtener una solución óptima. Por ejemplo, podríamos estar interesados en encontrar representantes para grupos homogéneos (reducción de datos), en encontrar "grupos naturales" y describir sus propiedades desconocidas (tipos de datos "naturales"), en encontrar grupos útiles y adecuados (clases de datos "útiles") o en la búsqueda de objetos de datos inusuales (detección de valores atípicos). Este algoritmo debe realizar algunas suposiciones que se constituyen en base a la similitud de los puntos a la proyección escogida y cada suposición hace grupos diferentes e igualmente válidos.

2.3.4 Reducción de la dimensionalidad

Es común trabajar con datos en los que cada observación viene con un alto número de características (features), en otras palabras, que tienen una alta dimensionalidad. Esto puede ser un desafío para el rendimiento computacional -ver más adelante *la maldición de la dimensionalidad*- de los algoritmos de aprendizaje automático, por lo que la reducción de dicha dimensionalidad es una de las técnicas utilizadas para tratar este problema.

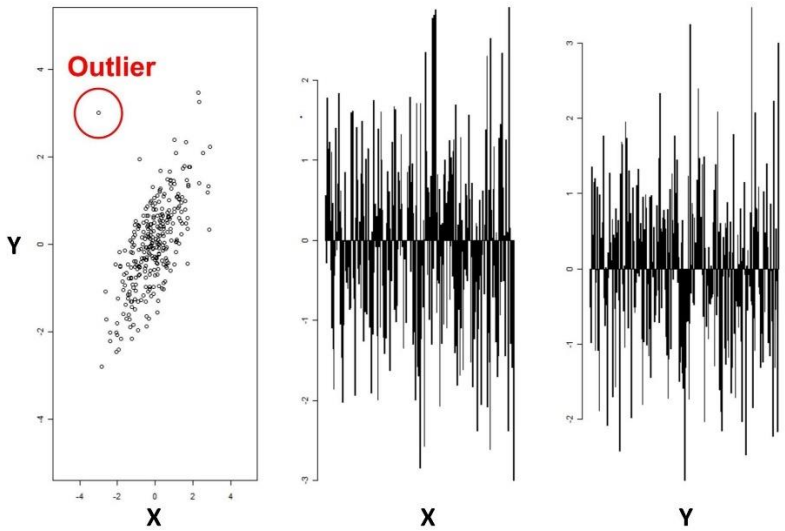


Los métodos de reducción de la dimensión funcionan mediante la búsqueda de correlaciones entre las características, lo que significaría que hay información redundante, ya que alguna característica podría explicarse parcialmente con las otras, aun sin llegar a ser una combinación lineal de estas. Mediante dicha técnica, se elimina el ruido de los datos (lo que en determinadas situaciones también puede disminuir el rendimiento del modelo) y se comprime los datos en un subespacio más pequeño, al tiempo que conserva la mayor parte de la información relevante. Técnicamente, esto puede entenderse como una proyección de un espacio *E* de dimensionalidad *m* a un espacio *E'* de dimensionalidad n, donde *m*>*n*.

2.3.5 Detección de Anomalías

La detección de anomalías (o detección de valores atípicos -*outliers*-) consiste en la identificación de elementos raros, eventos u observaciones que generan sospechas al diferir significativamente de la mayoría de los datos.

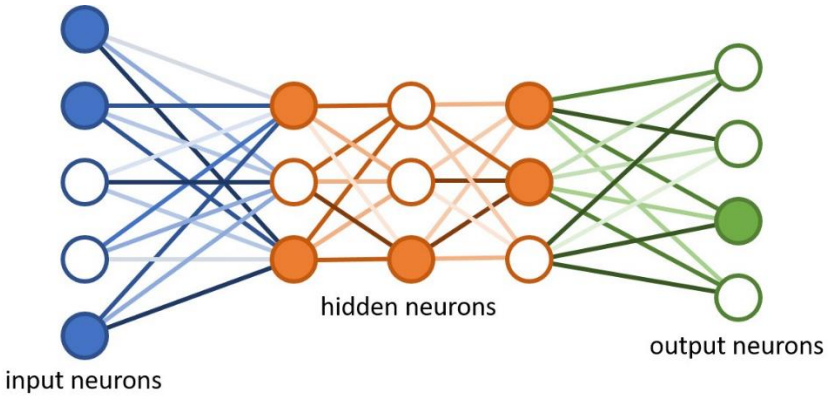
Normalmente, los datos anómalos se pueden conectar a algún tipo de característica subyacente o evento raro como, por ejemplo, el fraude bancario, problemas médicos, defectos estructurales, equipo defectuoso, etc. Esta conexión hace que sea muy interesante poder elegir qué puntos de datos pueden considerarse anomalías, ya que identificar estos eventos suele ser muy interesante para la solución de problemas de todo tipo en ingeniería y ciencias aplicadas.



Esto nos lleva a uno de los objetivos clave: ¿Cómo identificamos si los puntos de datos son normales o anómalos? En algunos casos simples, como en la anterior figura de ejemplo, la visualización de datos puede proporcionarnos información importante.

2.4 Deep Learning

El Deep learning o aprendizaje profundo es un subcampo del aprendizaje automático, que utiliza una estructura jerárquica de redes neuronales artificiales, que se construyen de manera análoga a las de un cerebro humano, con los nodos neuronales conectados como una red. Esa arquitectura para generar los modelos deseados ofrece diferentes ventajas, permitiendo entre otras cosas abordar el análisis de datos de forma no lineal y abordar una clase de problemas más amplia en las que el conocimiento a modelizar no nos es familiar, pudiendo prescindir hasta cierto punto de técnicas de Feature engineering..



La primera capa de la red neuronal toma datos en bruto como una entrada, la procesa, extrae cierta información y la pasa a la siguiente capa como una salida. Cada capa procesa la información dada por la anterior y se repite el proceso, hasta que los datos llegan a la capa final, cuyos resultados ofrecen una predicción.

Esta predicción se compara con el resultado conocido y luego, mediante un método llamado backpropagation, el modelo puede corregir los valores de la función peso *w* entre los nodos, reforzando aquellos cuyos resultados se ajusten a lo esperado y debilitando los que produzcan resultados erróneos.

Este método implica la elección de una función de pérdida (*loss function*) que será necesariamente una medida del grado de éxito del modelo según los parámetros que elijamos. Matemáticamente, dicha función de perdida será sucesivamente derivable según los pesos z de este modo podremos describir un gradiente matemático que nos conducirá a un mínimo (deseablemente no local) que maximizará nuestra función de evaluación.

Mediante el entrenamiento consistente en la sucesiva iteración de la ingestión de DataPoints en la red neuronal, esta desarrollará la capacidad de elaborar predicciones ajustadas al conjunto de datos suministrados, y si este es lo suficientemente heterogéneo, desarrollara la facultad de generalizar a partir del conocimiento suministrado, posibilitando la creación de conocimiento.

2.4.1 Redes Neuronales

En primer lugar, cuando estamos hablando de una red neuronal en el ámbito del aprendizaje automático, deberíamos correctamente utilizar el término "red neuronal artificial" (Artificial Neural Network), porque eso es lo que queremos decir. Las redes neuronales biológicas son mucho más complicadas que los modelos matemáticos utilizados para las ANN, tanto en su tamaño (número de nodos) como en las conexiones de estos.

No existe una definición universalmente aceptada de una ANN, pero una posible seria la siguiente: una ANN es una red constituidas por muchas unidades de proceso sencillas ("neuronas"), cada una de ellas posiblemente con una pequeña cantidad de memoria local. Las neuronas están conectadas por canales de comunicación ("conexiones") que por lo general llevan asociados valores numéricos (como función peso *w*), codificados por cualquiera de varios medios. Estos nodos operan solamente con sus datos locales y las entradas que reciben a través de las conexiones mediante una función llamada de activación.

Algunas ANN son modelos de redes neuronales biológicas y otras no, pero históricamente, gran parte de la inspiración para el campo de las ANN provino del deseo de producir sistemas artificiales capaces de sofisticados, tal vez "Inteligentes", cálculos similares a los que el cerebro humano rutinariamente realiza, y por lo tanto posiblemente para mejorar nuestra comprensión de la humana cerebro.

Como ya hemos citado anteriormente, la mayoría de las ANN tienen algún tipo de regla de "entrenamiento" por la cual los pesos de las conexiones se refuerzan y debilitan en base a los datos suministrados. En otras palabras, las ANN "aprenden" de los ejemplos, a medida que los niños aprenden a distinguir los perros de los gatos en función de ejemplos de perros y gatos. Si se entrena con cuidado, las NN pueden exhibir capacidad de generalización más allá de los datos de entrenamiento, es decir, para producir resultados aproximadamente correctos para casos nuevos que no se utilizaron para su entrenamiento.

Los ANN normalmente tienen un gran potencial para el paralelismo, ya que los cálculos de los componentes son en gran parte locales (independientes unos de otros). Algunas definiciones consideran Paralelismo masivo y alta conectividad como las características definitorias de ANNs, pero tales requisitos descartarían varios modelos simples, como los simples regresión lineal (una red de avance mínimo con solo dos unidades más sesgo), que son útiles como casos especiales de ANNs.

2.4.2 Aplicaciones de las Redes Neuronales Artificiales

Las aplicaciones prácticas de las NN con más frecuencia emplean aprendizaje supervisado. Ya hemos definido que entendemos por aprendizaje supervisado aquel que debe proporcionar datos de categorización que incluyan tanto la de entrada y el resultado deseado (el valor objetivo, que puede ser continuo o categórico).

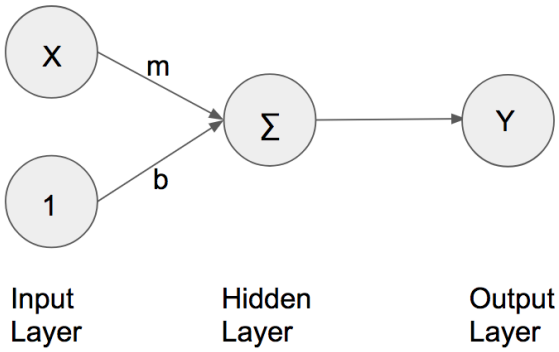
Después de un entrenamiento exitoso del modelo, se podrán presentar los datos de entrada solo a la ANN (es decir, los datos de entrada sin el resultado deseado), y esta calculará un valor de salida que se aproxima el resultado deseado. Sin embargo, para que la predicción tenga éxito, es posible que necesite muchos datos de entrenamiento y mucho tiempo de computadora para hacer el

entrenamiento. En muchas aplicaciones, como procesamiento de imágenes y texto, será necesario realizar un trabajo previo para seleccionar datos de entrada apropiados y codificar algunos datos como valores numéricos, lo que se conoce como Feature Engineering.

En la práctica, las ANN son especialmente útiles para la clasificación y la solución de problemas de aproximación / clasificación que son tolerantes a alguna imprecisión, que tienen una gran cantidad de datos de entrenamiento disponibles, pero para los cuales reglas lógicas de fácil expresión (tales como aquellos que podrían ser utilizados en un sistema experto) no se pueden aplicar fácilmente.

Casi cualquier función vectorial de dimensión finita en un conjunto compacto puede ser aproximada a la precisión arbitraria por ANNs de avance (que son el tipo más a menudo utilizado en aplicaciones prácticas) si tiene datos suficientes y suficientes recursos informáticos. Matemáticamente hablando, los algoritmos de machine learning anteriormente citados pueden entenderse como casos particulares de una red neuronal con unos determinados ajustes en cuanto a topología y algoritmos de solución claramente definidos.

Para ser un poco más precisos, las redes de avance con una sola capa oculta y entrenados por mínimos cuadrados son estimadores estadísticamente consistentes de funciones de regresión por mínimos cuadrados bajo ciertas supuestos en relación al muestreo que se dan prácticamente en todas las ocasiones. En la figura siguiente puede observarse como los pesos de los conectores *m* (input) y *b* (bias) corresponden a la pendiente y constante de la recta de regresión *y=mx+b*



Esto no quiere decir en ningún caso que las redes neuronales artificiales puedan sustituir a cualquier herramienta matemática. Un ejemplo de una función que una red neuronal típica no puede aprender es Y = 1 / X en el intervalo abierto (0,1), dado que, para cualquier función de activación de salida limitada, el error será arbitrariamente grande como la entrada se acerca a cero.

También hay muchos otros problemas importantes que no pueden resolverse mediante la aplicación de un algoritmo basado en redes neuronales artificiales, dado que en muchos casos la red no podrá aprenderlos sin memorizar todo el entrenamiento (overfitting), tales como:

- Predecir números aleatorios o pseudoaleatorios.
- Factorización de enteros grandes.
- Determinar si un entero grande es primo o compuesto.
- Descifrar cualquier cosa encriptada por un buen algoritmo.

Finalmente, sin tener en cuanta si el tipo de problema es atacable mediante esta metodología, es necesario hacer un último inciso en un aspecto fundamental: es importante entender que no hay métodos para entrenar a las NN para que puedan crear mágicamente información que no está contenida en conjunto de datos de entrenamiento. La ANN descubrirá el conocimiento si lo hay, previo ajuste correcto de su topología y sus hiperparámetros, pero eso es todo.

2.4.3 Overfitting

El problema crítico en el desarrollo de una red neuronal es la generalización: ¿cómo se comportará la red neuronal prediciendo valores para los casos que no están en el conjunto de entrenamiento? Las redes neuronales, así como otros métodos de estimación no lineales flexibles, tales como regresión del kernel y splines de suavizado, pueden sufrir de falta de adaptación o sobrealimentación -overfitting-.

Una red que no sea lo suficientemente compleja puede fracasar en la tarea de detectar parcial o completamente la señal en un conjunto de datos complicado, lo que lleva a un desajuste en sus predicciones.

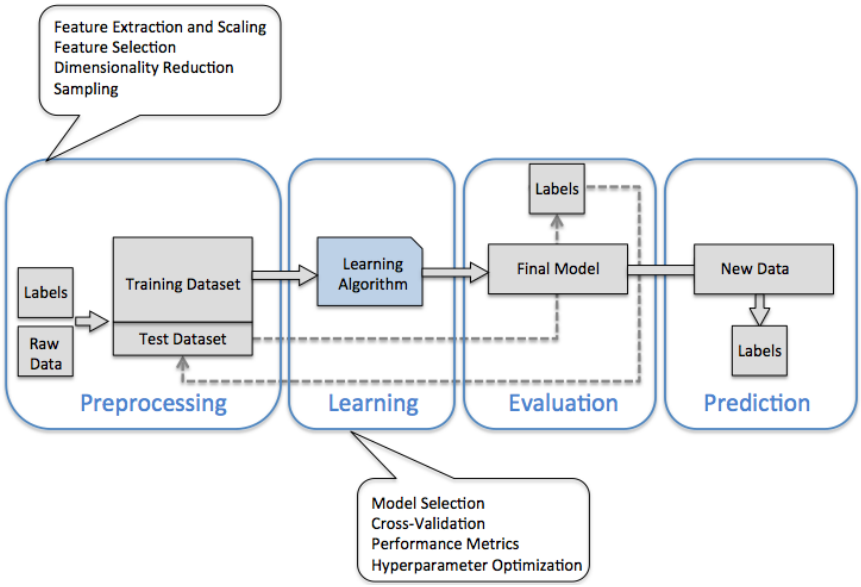
Asimismo, una red que es demasiado compleja puede adaptarse al ruido, no solo a la señal, llevando al anteriormente citado overfitting. Este es especialmente peligroso porque puede conducir fácilmente a predicciones con un nivel de precisión que están mucho más allá del alcance del entrenamiento de un modelo, pero solamente para datos presentes en el conjunto de los datos de entrenamiento.

Naturalmente, se deduce de lo anteriormente dicho que la mejor manera de evitar el overfitting es usar número lo suficientemente elevado de datos de entrenamiento. Es fácil e intuitivo entender que, si la red tiene una complejidad suficiente para aprenderse de memoria todo el dataset de entrenamiento, un dataset con un número más elevado de datapoints podría impedir que este comportamiento tuviera lugar.

Naturalmente, no siempre se puede aumentar el numero de datos solo con desearlo, por lo que existen técnicas para evitar que dicho comportamiento tenga lugar. En primer lugar, y de forma intuitiva, podríamos reducir la complejidad de la red neuronal, reduciendo su capacidad de memorizar el dataset de entrenamiento. No obstante, la reducción de la complejidad de la red puede asimismo reducir su capacidad de generar un modelo capaz de generalizar de manera optima en base a las muestras suministradas, lo que seria incurrir en el mencionado underfitting.

Para encontrar el equilibrio entre ambos estados no deseables, reservaremos una parte del dataset de entrenamiento como dataset de validación. Este no se utilizará para entrenar a la red neuronal propiamente dicha, sino que será únicamente empleado para en cada epoch evaluar la capacidad de la red de realizar predicciones sobre un conjunto de datos (el de validación) anteriormente no visto. De forma natural, cuando la precisión en la evaluación del conjunto de validación decrezca, sabremos que el modelo estará empezando a generar overfitting y podremos frenar su entrenamiento.

2.5 Esquema general para la elaboración de modelos de aprendizaje automático



2.6 Preparación del conjunto de Datos a analizar

En este proyecto, usaremos el conjunto de datos NSL-KDD disponible públicamente en:

University of New Brunswick. NSL-KDD, **http://nsl.cs.unb.ca/nsl-kdd/**

NSL-KDD resuelve problemas de redundancias y problemas de datos duplicados en el conjunto de entrenamiento en el conjunto de datos KDDcup99. Muchos clasificadores convencionales no logran diferenciar entre Tráfico normal y de ataque. Haz que remarcar que el NSL-KDD contiene un desequilibrio entre labels correspondientes tráfico normal y al de datos de ataque. La relación entre dichas etiquetas de ataque y el tráfico normal es comparativamente baja. Lo que genera un fenómeno que se conoce como El problema de desequilibrio de clase (Class Imbalance Problem). Esto ocurre cuando la clase minoritaria, en nuestro DataSet la clase de ataque, exhibe una significativamente menor representación en comparación con la de la mayoría, o normal, datos etiquetados.

Por lo tanto, existe la necesidad de identificar técnicas especializadas para contrarrestar dicho desequilibrio de este tipo asignando una importancia mayor a las clases minoritarias.

3 Elaboración

3.1 Definición del Problema: Implementación de un Sistema de Detección de Intrusión.

Los sistemas de seguridad perimetrales van más allá de las barreras lógicas de control de acceso iniciales para detectar intentos o penetraciones con éxito en una red mediante un sensor pasivo. Los sistemas de prevención de intrusiones se desarrollaron como una mejora de los sistemas de detección de intrusión pasiva, teniendo la capacidad de interceptar la línea de comunicación entre los orígenes y el destino de cada paquete y poder actuar automáticamente sobre las anomalías detectadas.

La detección de paquetes en tiempo real es un requisito para cualquier sistema de detección o prevención de intrusiones; proporcionando tanto una capa de visibilidad para el contenido que fluye a través del perímetro de una red y datos con los que a partir de los cuales la detección de amenazas puede llevarse a cabo.

Pese a que la elaboración del modelo clasificador no puede realizarse en tiempo real, una vez dicho modelo ha sido implementado su desempeño sí que podrá aplicarse de este modo. En todas las soluciones posibles, ya sea académicas o comerciales, esto exige que se dedique un tiempo inicial a la recolección de un número significativo de datos que describan el comportamiento estándar del sistema para poder elaborar el modelo inicial, siendo naturalmente posible, e incluso deseable, la continua recopilación de datos adicionales para poder actualizar dicho modelo con el tiempo. Esto nos permitirá tanto una mejor detección del comportamiento de base del sistema, así como la posibilidad de descubrir en sucesivas ingestiones de paquetes nuevas amenazas. Del mismo modo, en las aplicaciones comerciales el conjunto de estos datos no puede estar etiquetado, lo que deviene en la necesidad de combinar un análisis basado en la detección de anomalías no supervisado con los clasificadores basados en datasets etiquetados. Esto se soluciona con un entrenamiento genérico inicial supervisado, común a todos los modelos en cada implementación del sistema, que será completado con los datos recogidos con el tiempo y una base de conocimiento compartida.

En aplicaciones académicas, como la que nos ocupa, dispondremos de un conjunto de datos (NSL-KDD1999) etiquetado previamente, lo que nos habilita para poder realizar un entrenamiento supervisado. Este Dataset adopta diferentes formas y tamaño según la variación escogida, pudiendo tener algunas de sus manifestaciones un feature engineering previo significativo.

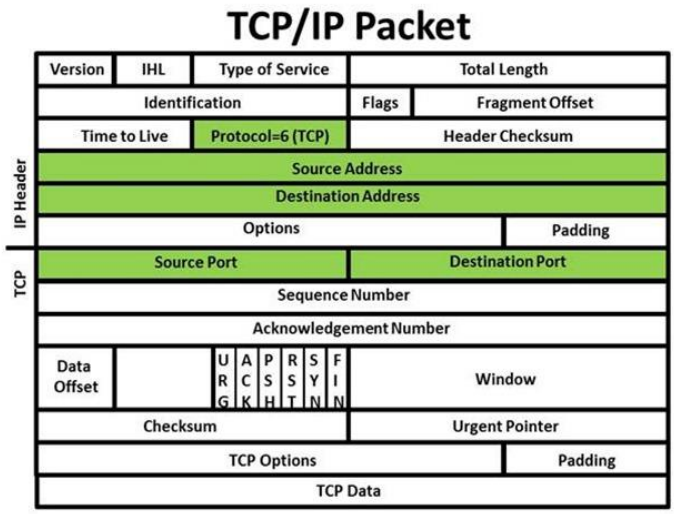
3.2 Estructura del DataSet

La captura de datos de red en vivo es la forma principal de registrar la actividad de la red para análisis online u offline. Como una cámara de video vigilancia en una intersección de tráfico, los analizadores de paquetes (también conocidos como rastreadores de paquetes/*sniffers*) interceptan y registran el tráfico en la red.

Estos registros son útiles no solamente para análisis de seguridad, sino también para depuración, estudios de desempeño y monitoreo operacional. Cuando se encuentra en los lugares correctos, como generalmente conmutadores de red e interfaces correctamente configurados, los analizadores de paquetes pueden ser una herramienta valiosa para generar ingentes conjuntos de datos detallados que proporcionan una imagen completa de todo lo que está sucediendo en la red. Esta imagen se basa en el modelo OSI de capas:

Layer	Function	Example
Application (7)	Services that are used with end user applications	SMTP,
Presentation (6)	Formats the data so that it can be viewed by the user Encrypt and decrypt	JPG, GIF, HTTPS, SSL, TLS
Session (5)	Establishes/ends connections between two hosts	NetBIOS, PPTP
Transport (4)	Responsible for the transport protocol and error handling	TCP, UDP
Network (3)	Reads the IP address form the packet.	Routers, Layer 3 Switches
Data Link (2)	Reads the MAC address from the data packet	Switches
Physical (1)	Send data on to the physical wire.	Hubs, NICS, Cable

Siendo la estructura de cada paquete capturado, en el caso de ser un paquete TCP, similar a:



Como puede imaginarse, estos datos pueden llegar a ser abrumadores por su cantidad. En entornos complejos de redes, incluso las tareas simples como rastrear el desarrollo normal de una sesión TCP no serán fáciles. Con el acceso a datos en bruto ricos en información, el siguiente paso será generar características útiles para el análisis de datos.

En el caso del Dataset NSL-KDD1999, las características en bruto de un datagrama se han filtrado para elaborar un conjunto de 41 diferentes, recogido en la siguiente figura:

S.NO	FEATURE NAME	S.NO	FEATURE NAME
1	Duration	22	Is_guest_login
2	Protocol type	23	Count
3	Service	24	Serror_rate
4	Src_byte	25	Rerror_rate
5	Dst_byte	26	Same_srv_rate
6	Flag	27	Diff_srv_rate
7	Land	28	Srv_count
8	Wrong_fragment	29	Srv_serror_rate
9	Urgent	30	Srv_rerror_rate
10	Hot	31	Srv_diff_host_rate
11	Num_failed_logins	32	Dst_host_count
12	Logged_in	33	Dst_host_srv_count
13	Num_compromised	34	Dst_host_same_srv_count
14	Root_shell	35	Dst_host_diff_srv_count
15	Su_attempted	36	Dst_host_same_src_port_rate
16	Num_root	37	Dst_host_srv_diff_host_rate
17	Num_file_creations	38	Dst_host_serror_rate
18	Num_shells	39	Dst_host_srv_serror_rate
19	Num_access_shells	40	Dst_host_rerror_rate
20	Num_outbound_cmds	41	Dst_host_srv_rerror_rate
21	Is_hot_login		

Constituyéndose cada Datapoint como lo que definiremos un “vector de conexión”.

El conjunto de datos original de 1999 KDD se creó para el Programa de evaluación de detección de intrusiones DARPA, siendo preparado y gestionado por MIT Lincoln Laboratory. Los datos fueron recolectados a lo largo de nueve semanas y consiste en tráfico *tcpdump* sin procesar en una red de área local (LAN) que simula el entorno de una típica red aérea de Estados Unidos LAN. Algunos ataques a la red.

Se llevaron a cabo deliberadamente durante el período de ingestión de datos. Hubo 38 diferentes tipos de ataques, pero solo 24 están disponibles en el conjunto de entrenamiento. Estos ataques pertenecen a cuatro categorías generales:

dos : Denegación de servicio
r2l : Acceso no autorizado desde servidores remotos
u2r: Intento de escalación de privilegios
probe: Sondeos de fuerza bruta

Y clasificaremos al trafico no malévolo con la quinta etiqueta : *normal*.

3.3 ML vs DL

Para el proyecto elegido hemos escogido desarrollar dos modelos de Machine Learning para intentar contrastar sus ventajas e inconvenientes. El Primer método será el de Arboles de decisión, que implementaremos en Python 3.7 con una mínima adaptación de los registros de cada vector de conexión del conjunto de datos. El segundo método que implementaremos será el de una red neuronal artificial, primero utilizando el *feature engineering* realizado para la implementación del árbol de decisión, y si es necesario luego modificándolo para poder contrastar los resultados.

Para la primera implementación de la red neural, además utilizaremos un modelo de unsupervised *feature engineering*, reduciendo la dimensionalidad para la segunda.

3.4 Arboles de decision

El primer modelo a desarrollar consistirá en la aplicación de un algoritmo de Árbol de decisión aplicado al Conjunto de dados NSL-KDD1999. Las ventajas a priori de dicha elección consisten en su naturaleza transparente. A diferencia de otros modelos de toma de decisiones, el árbol de decisiones hace explícitas todas las alternativas posibles y rastrea cada alternativa a su conclusión en una sola vista, lo que permite una comparación fácil entre las distintas alternativas. El uso de nodos separados para denotar las decisiones definidas por el usuario, las incertidumbres y el final del proceso otorga mayor claridad y transparencia al proceso de toma de decisiones.

Asimismo, su especificidad es una ventaja adicional en aquellos ámbitos en los que una de las variables en las cuales basar la toma de decisiones tenga una evaluación similar al coste para poder asignar valores específicos al problema, las decisiones y los resultados de cada decisión. Esto reduce la ambigüedad en la toma de decisiones. Todos los escenarios posibles de una decisión encuentran representación mediante una bifurcación y un nodo claros, lo que permite ver todas las soluciones posibles claramente en una sola vista. Esta métrica del coste puede ser monetario, pero también computacional, temporal, etc.

El árbol de decisión presenta otras ventajas adicionales, ya que permite un análisis exhaustivo de las consecuencias de cada decisión posible, como a qué conduce la decisión, si termina en incertidumbre o una conclusión definitiva, o si conduce a nuevos problemas para los cuales el proceso necesita repetición. Por último, un árbol de decisiones también permite la partición de datos en un nivel mucho más profundo, que no se logra tan fácilmente con otros clasificadores de toma de decisiones, como la regresión logística o el soporte de máquinas de vectores.

En cuanto a la facilidad de uso, los árboles de decisión también resultan interesantes: El árbol de decisiones proporciona una ilustración gráfica del problema y varias alternativas en un formato simple y fácil de entender que no requiere explicación. Los árboles de decisión también permiten la clasificación de datos sin cómputo, pueden manejar tanto variables continuas como categóricas, y proporcionan una indicación clara de los campos más importantes para la predicción o clasificación, todas las características no coincidentes al comparar este modelo con otros modelos compatibles como el vector de soporte o Regresión logística.

3.4 Implementación del Árbol de decisión

Comenzaremos asignando las etiquetas a cada feature de cada Datapoint y asignando cada tipo de ataque a su categoría principal:

```
header_names = ['duration', 'protocol_type', 'service', 'flag', 'src_bytes',
'dst_bytes', 'land', 'wrong_fragment', 'urgent', 'hot', 'num_failed_logins',
'logged_in', 'num_compromised', 'root_shell', 'su_attempted', 'num_root',
'num_file_creations', 'num_shells', 'num_access_files', 'num_outbound_cmds',
'is_host_login', 'is_guest_login', 'count', 'srv_count', 'serror_rate',
'srv_serror_rate', 'rerror_rate', 'srv_rerror_rate', 'same_srv_rate',
'diff_srv_rate', 'srv_diff_host_rate', 'dst_host_count', 'dst_host_srv_count',
'dst_host_same_srv_rate', 'dst_host_diff_srv_rate', 'dst_host_same_src_port_rate',
'dst_host_srv_diff_host_rate', 'dst_host_serror_rate', 'dst_host_srv_serror_rate',
'dst_host_rerror_rate', 'dst_host_srv_rerror_rate', 'attack_type', 'success_pred']


# The directory containing all of the relevant data files

data_folder = Path("/root/Documents/TFM/dataset/")

file_to_open = data_folder / "training_attack_types.txt"

category = defaultdict(list)
category['benign'].append('normal')

with open(file_to_open, 'r') as f:
    for line in f.readlines():
        attack, cat = line.strip().split(' ')
        category[cat].append(attack)

print (category)

attack_mapping = dict((v,k) for k in category for v in category[k])

print (attack_mapping)
```

Seguiremos leyendo los datos, que organizaremos en dos partes train_df, que utilizaremos para el entrenamiento del modelo, y test_df, que será utilizado para su evaluación:

```
file_train = data_folder / "KDDTrain+.txt"
file_test = data_folder / "KDDTest+.txt"

train_file = os.path.join(file_train)
test_file = os.path.join(file_test)

# Read training data
train_df = pd.read_csv(train_file, names=header_names)

train_df['attack_category'] = train_df['attack_type'] \
.map(lambda x: attack_mapping[x])

train_df.drop(['success_pred'], axis=1, inplace=True)

# Read test data
test_df = pd.read_csv(train_file, names=header_names)

test_df['attack_category'] = test_df['attack_type'] \
.map(lambda x: attack_mapping[x])

test_df.drop(['success_pred'], axis=1, inplace=True)
```

La necesidad de dividir el conjunto de los datos de este modo obedece a una razón fundamental: deseamos evitar que el modelo solo se comporte correctamente en presencia de los datos que ya

conoce, porque la validez de su resultado solo puede comprobarse en presencia de datos previamente no encontrados. A tal efecto, dividimos el conjunto de datos inicial en dos subconjuntos disjuntos, mediante lo cual conseguiremos disponer de un conjunto de datos valido para nuestros objetivos. Una vez hecho esto, nuestro siguiente paso será empezar con el tratamiento de los datos propiamente dicho.

Procederemos inicialmente a separar para cada Datapoint las características (features) de sus etiquetas (labels):

```
# Let's begin by splitting the test and training DataFrames into data and labels

train_Y = train_df['attack_category']
train_x_raw = train_df.drop(['attack_category','attack_type'], axis=1)
test_Y = test_df['attack_category']
test_x_raw = test_df.drop(['attack_category','attack_type'], axis=1)
```

Antes de que podamos aplicar cualquier algoritmo a los datos, debemos preparar los datos para el consumo. Primero codifiquemos las variables categóricas (referidas en el conjunto de datos como variables simbólicas) como *dummies*. Esto consiste en convertir un feature del data set que puede tomar un número N limitado, no ordenado y no continuo de valores en N features de las cuales una y solo una de ellas puede tomar el valor 1, siendo el valor del resto de 0. Un ejemplo en caso Male/female seria:

```
df = pd.concat([df, dummy], axis=1)
df.head()
```

	sx	rk	yr	dg	yd	sl	female	male
0	male	full	25	doctorate	35	36350	0	1
1	male	full	13	doctorate	22	35350	0	1
2	male	full	10	doctorate	23	28200	0	1
3	female	full	7	doctorate	27	26775	1	0
4	male	full	19	masters	30	33696	0	1

Por conveniencia, generemos la lista de variables categóricas y una lista de variables continuas. Una vez hecho esto, seguiremos dividiendo el conjunto que hemos hecho de features categóricas entre nominales y binarias, porque las procesaremos de manera diferente:

```
file_names = data_folder / "kddcup.names"

feature_names = defaultdict(list)
with open(file_names, 'r') as f:
    for line in f.readlines()[1:]:
        name, nature = line.strip()[:-1].split(': ')
        feature_names[nature].append(name)

print("\n")
print(feature_names)

# Concatenate DataFrames
combined_df_raw = pd.concat([train_x_raw, test_x_raw])

# Keep track of continuous, binary, and nominal features
continuous_features = feature_names['continuous']
continuous_features.remove('root_shell')
binary_features = ['land','logged_in','root_shell',
```

```
    'su_attempted','is_host_login',
    'is_guest_login']

nominal_features = list(set(feature_names['symbolic']) - set(binary_features))
```

Luego, usamos la función pandas.get_dummies () para convertir las variables nominales en variables dummies. Combinamos train_x_raw y test_x_raw, ejecutamos el conjunto de datos a través de esta función, y luego los volvemos a separar en conjuntos de prueba y entrenamiento

Esto es necesario porque puede darse la posibilidad de que algunos valores de variables simbólicas aparezcan solamente en un conjunto de datos y no en el otro, y generar por separado variables ficticias para ellos podría, de forma evidente, tener como resultado inconsistencias en las columnas de ambos conjuntos de datos.

Normalmente no deben realizarse acciones de preprocesamiento en una combinación de entrenamiento y datos de prueba, pero es aceptable en este escenario. No estamos haciendo nada que perjudicará a la elaboración del modelo, ni mezclando elementos de los conjuntos de entrenamiento y prueba. Básicamente lo que realizamos es una transformación semántica para poder realizar la ingestión de los Datapoints en el algoritmo, pero aún no hemos empezado a elaborarlo y ninguna de las transformaciones altera el contenido de los conjuntos. Típicamente, tendremos pleno conocimiento de todas las variables categóricas, ya sea porque las definiremos o porque el conjunto de datos proporcionará esta información.

En nuestro caso, el conjunto de datos no vino con una lista de valores posibles de cada variable categórica, por lo que podemos preprocesar de la siguiente manera:

```
# Generate dummy variables

combined_df = pd.get_dummies(combined_df_raw, \
    columns=feature_names['symbolic'], \
    drop_first=True)

# Separate into training and test sets again

train_x = combined_df[:len(train_x_raw)]
test_x = combined_df[len(train_x_raw):]

# Keep track of dummy variables

dummy_variables = list(set(train_x)-set(combined_df_raw))
```

Ahora que ya tenemos una estructura de features adecuada, podemos proceder a una inspección de nuestro DataSet modificado:

```
print(train_x.describe())
```

Observando el siguiente resultado en la consola:

```
           duration     src_bytes  ...  is_host_login_1  is_guest_login_1
count  125973.00000  1.259730e+05  ...    125973.000000     125973.000000
mean      287.14465  4.556674e+04  ...         0.000008          0.009423
std      2604.51531  5.870331e+06  ...         0.002817          0.096612
min         0.00000  0.000000e+00  ...         0.000000          0.000000
25%         0.00000  0.000000e+00  ...         0.000000          0.000000
50%         0.00000  4.400000e+01  ...         0.000000          0.000000
75%         0.00000  2.760000e+02  ...         0.000000          0.000000
max     42908.00000  1.379964e+09  ...         1.000000          1.000000 P
```

¿Qué quiere decir exactamente esto? Pues que la distribución de valores para algunas de nuestras variables es extremadamente desigual. ¿Es esto relevante para la elaboración de nuestro modelo? Si, porque dentro de cada uno de los algoritmos de Machine Learning se utilizan métricas de distancia (Euclidiana, Manhattan…) para calcular la clasificación.

Cualquier método basado en la distancia para la clasificación será extremadamente sensible a una distribución de datos no normalizada. Por ejemplo, la media de *src_bytes* es mayor que la media de *num_failed_logins* por siete órdenes de magnitud, y su desviación estándar es mayor en ocho órdenes de magnitud. Si no procediéramos a realizar una normalización de valores, la característica *src_bytes* dominaría el resultado de cualquier cálculo de métrica de distancia, causando que el modelo perdiese información potencialmente importante contenida en *num_failed_logins*. Si se quiere visualizar este hecho de la forma más intuitiva, piénsese en K-Means como método de aprendizaje en un entorno no normalizado y con gran varianza en sus distribuciones de datos.

La estandarización es un proceso que vuelve a normaliza y reescala una serie de datos para tener una media de 0 y una desviación estándar de 1 (varianza unitaria). Es un requisito para muchos algoritmos de aprendizaje automático, y útil siempre que las funciones de los datos de entrenamiento varíen ampliamente en sus características de distribución.

La biblioteca scikit-learn incluye la clase sklearn.preprocessing.StandardScaler que proporciona esta funcionalidad de forma sencilla. Procedamos a utilizarla:

```
# Fit StandardScaler to the training data
standard_scaler = StandardScaler().fit(train_x[continuous_features])

# Standardize training data
train_x[continuous_features] = \
standard_scaler.transform(train_x[continuous_features])

# Standardize test data with scaler fitted to training data
test_x[continuous_features] = \
standard_scaler.transform(test_x[continuous_features])
```

Y de nuevo:

```
print(train_x.describe())
```

obteniendo:

```
           duration     src_bytes  ...  is_host_login_1  is_guest_login_1
count  1.259730e+05  1.259730e+05  ...    125973.000000     125973.000000
mean   2.549477e-17 -4.512349e-19  ...         0.000008          0.009423
std    1.000004e+00  1.000004e+00  ...         0.002817          0.096612
min   -1.102492e-01 -7.762241e-03  ...         0.000000          0.000000
25%   -1.102492e-01 -7.762241e-03  ...         0.000000          0.000000
50%   -1.102492e-01 -7.754745e-03  ...         0.000000          0.000000
75%   -1.102492e-01 -7.715224e-03  ...         0.000000          0.000000
max    1.636428e+01  2.350675e+02  ...         1.000000          1.000000
```

Con este mínimo tratamiento de datos en el que mas allá de la estandarización implementada no hemos reducido la dimensionalidad ni aplicado técnicas de sobremuestreo para sobrerepresentar a las categorías menos numerosas en el conjunto de datos, procedamos a obtener nuestro modelo de Árbol de decisión:

```
classifier = DecisionTreeClassifier(random_state=0)
classifier.fit(train_x, train_Y)
```

```
pred_y = classifier.predict(test_x)

results = confusion_matrix(test_Y, pred_y)
error = zero_one_loss(test_Y, pred_y)

error_formatted = "%.9f" % error

print (results)
print("\n")
print ("Error: "+ error_formatted)
```

Donde obtendremos:

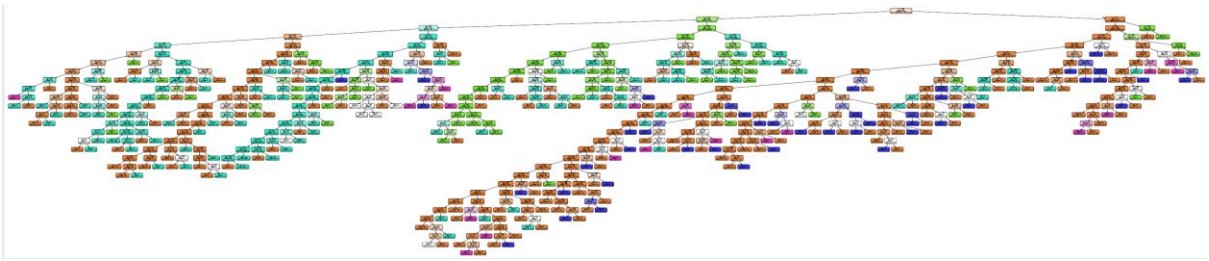
```
 [[67343     0     0     0     0]
 [    5 45922     0     0     0]
 [    1     0 11655     0     0]
 [    0     0     0   995     0]
 [    1     0     0     0    51]]


Error: 0.000055567
```

Un error negligible, en el que solamente siete muestras de 126K han sido clasificadas de forma errónea. Vemos por tanto que la eficacia de nuestro clasificador no solo es mejor de la esperada, sino que difícilmente podrá ser mejorada por otros métodos.

Si observamos la representación grafica del árbol generado resulta fácil ver que el motivo de esto es la extraordinaria complejidad de este:

```
# Visualize the tree

from sklearn.externals.six import StringIO
from IPython.display import Image
from sklearn.tree import export_graphviz
import pydotplus

dot_data: StringIO = StringIO()

export_graphviz(classifier, out_file = dot_data, filled=True, rounded=True,
special_characters=True)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
Image(graph.create_png())

graph.write_pdf("iris.pdf")
```

Donde:



Veamos que pasa si limitamos la capacidad del algoritmo para efectuar una predicción tan exacta determinando la profundidad máxima del árbol a 5 niveles mediante:

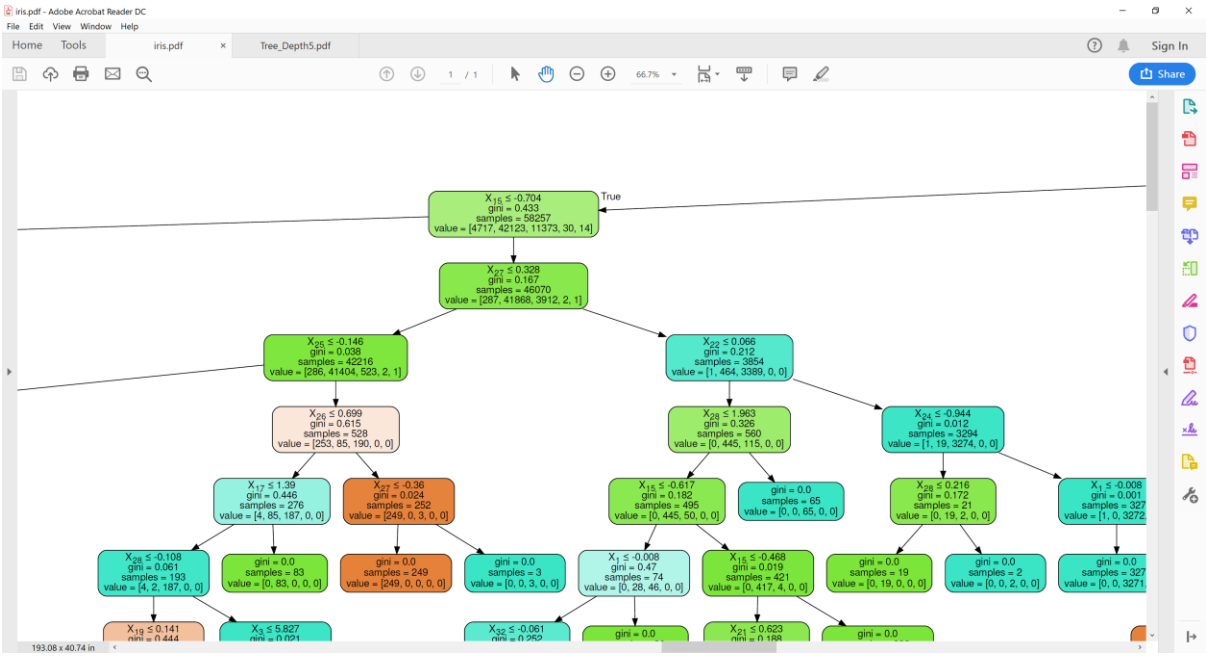
```
classifier = DecisionTreeClassifier(random_state=0, max_depth=5)
```

Lo que nos ofrece un resultado de:

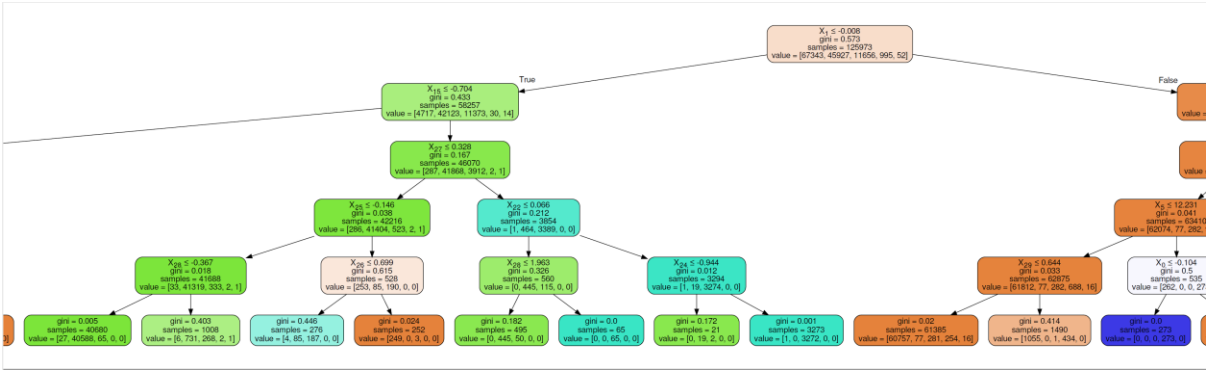
```
[[67183    58   102     0     0]
 [  143 45674   110     0     0]
 [  446   385 10825     0     0]
 [  714     2     6   273     0]
 [   50     1     1     0     0]]
```

```
Error: 0.016019306
```

Y un error de un 1.6%, donde la matriz de confusión nos muestra un numero superior de muestras mal clasificadas, incluyendo falsos positivos. Naturalmente esto es así por que el Árbol es menos complejo, pudiendo clasificar con menos eficacia el conjunto de test. Dado que no hemos variado el método para generar el árbol (por defecto Gini), en realidad se trata del mismo árbol, pero podado a una profundidad de cinco bifurcaciones:



Cuya imagen corresponde al Árbol sin límite de profundidad, y:

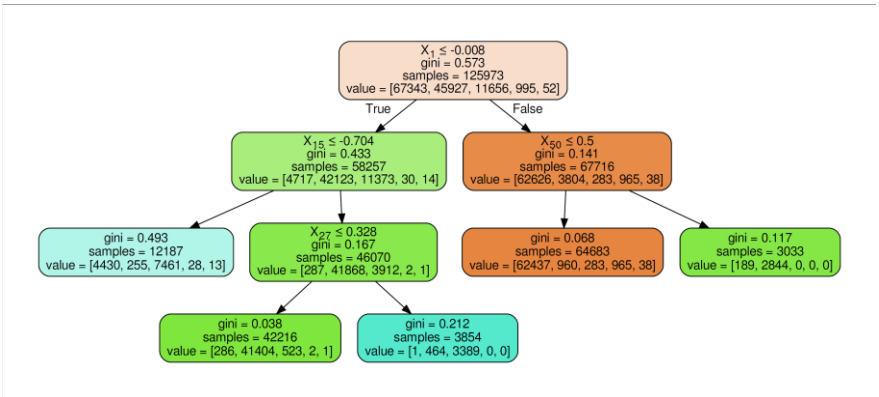


Para seguir observando la eficiencia de este clasificador, limitemos ahora el numero de hojas a 5, que configuran el caso particular en el que el número de hojas del árbol clasificador se correspondería con el de categorías:

```
[[62437   475  4431     0     0]
 [  960 44248   719     0     0]
 [  283   523 10850     0     0]
 [  965     2    28     0     0]
 [   38     1    13     0     0]]
```

```
Error: 0.066982607
```

Viendo como el algoritmo renuncia a clasificar a las clases menos numerosas para obtener la máxima precisión.



Para terminar con la discusión de la implementación utilizando técnicas de Mchine Learning, podemos destacar un aspecto adicional. La Normalizacion que hemos realizado previamente no es necesaria para la elaboración del Árbol de decisión debido a como en realidad este modelo segmenta cada bifurcación utilizando solo una variable, lo que elimina la interacción entre variables posiblemente desequilibradas.

¿Qué pasaría si utilizásemos una técnica que utilizase la métrica de una distancia para generar sus predicciones? Probemos con K-neighbours sin re-escalar las features:

```
from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier(n_neighbors=5)
classifier.fit(train_x, train_Y)

pred_y = classifier.predict(test_x)
```

Obteniendo:

```
[[67124    60   129    26     4]
 [   24 45851    52     0     0]
 [   73   122 11459     2     0]
 [   40     0     1   954     0]
 [   23     0     1     0    28]]
```

```
Error: 0.004421582
```

Y pudiendo comprobar que una vez normalizadas conseguimos una precisión superior:

```
[[67249    23    42    29     0]
 [   32 45892     3     0     0]
 [   73     3 11580     0     0]
 [   47     1     2   943     2]
 [   21     0     0     3    28]]

Error: 0.002230637
```

4.0 Implementación Avanzada.

Hasta ahora nos hemos limitado a realizar una primera aproximación a la aplicación de técnicas de aprendizaje automático para la detección de paquetes maliciosos. Podemos comprobar en el apartado anterior como con un mínimo de tratamiento realizado sobre el conjunto de datos y la implementación de un árbol de decisión, obteníamos una precisión extremadamente elevada, que superaba al 99.99%.

Si bien esto es naturalmente deseable en un entorno de producción, dado que el objetivo de la elaboración de un modelo clasificador es el de poder elaborar predicciones sobre conjuntos de datos nunca anteriormente vistos lo más precisas posibles, para el estudio que nos ocupa un grado de precisión tan elevado puede hacer relativamente difícil observar diferencias sustanciales entre la aplicación de ciertas técnicas y elección de ciertos parámetros a la hora de escoger un modelo.

¿Por qué hemos obtenido una precisión tan elevada? Comentamos anteriormente como en el proceso de generación de un modelo de aprendizaje automático, el tratamiento previo de los datos disponibles -*Feature Engineering*- era tan importante como el proceso de generación del modelo en sí. El conjunto de datos escogido NSL-KDD1999 tiene como particularidad el hecho de que ha sido previamente adaptado y editado para eliminar ciertos problemas:

- No incluye registros redundantes en el conjunto de entrenamiento, por lo que los clasificadores no estarán sesgados hacia registros más frecuentes.

- No hay registros duplicados en los conjuntos de prueba propuestos; por lo tanto, el rendimiento de los modelos no está sesgado por los métodos que tienen mejores tasas de detección en los registros frecuentes.

- El número de registros seleccionados de cada grupo de nivel de dificultad es inversamente proporcional al porcentaje de registros en el conjunto de datos KDD original. Como resultado, las tasas de clasificación de los distintos métodos de aprendizaje automático varían en un rango más amplio, lo que hace que sea más eficiente tener una evaluación precisa de diferentes técnicas de aprendizaje.

- El número de registros en los conjuntos de entrenamientos y los conjuntos de pruebas es razonable, lo que hace que sea asequible ejecutar los experimentos en el conjunto completo sin la necesidad de seleccionar al azar una pequeña parte. En consecuencia, los resultados de la evaluación de diferentes trabajos de investigación serán consistentes y comparables.

Para la implementación que a continuación llevaremos a cabo utilizaremos el DataSet KDDCup-'99', que no realiza una preproducción en los datos tan intensa, y ofrecerá índices de precisión inferiores si se utilizan las mismas técnicas. Esta decisión solo es justificable, de nuevo, en el ámbito puramente académico que nos ocupa. No olvidemos que, en primer lugar, el conjunto de datos es defectuoso y no debe utilizarse (declaración *KDNuggets*). Existen para ello dos motivos de peso:

a) no es en absoluto realista, en particular no para los ataques modernos (ni siquiera para ataques reales en 1998, ya que todo el entorno fue simulado): Hoy en día, la mayoría de los ataques son inyección de SQL y robo de contraseñas a través de troyanos, ninguno de los cuales será detectable con este tipo de datos.

b) el conjunto de datos se centra en los ataques, por lo que consiste en ataques con algo de ruido de fondo; mientras que el tráfico real será en gran parte datos (ruido) y algunos ataques.
c) Se simuló con una red virtual en su mayor parte, y la topología de red simulada en si misma ya ofrece información suficiente para identificar dichos "ataques" solamente por su proveniencia.

4.1 Implementaciones con Machine Learning:

Procedamos a implementar entonces una serie de modelos sobre este dataset utilizando técnicas de machine learning:

4.1.1. Arbol de decision

```
import numpy as np
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import Normalizer
from sklearn.metrics import (precision_score, recall_score,f1_score,
accuracy_score,mean_squared_error,mean_absolute_error, roc_curve,
classification_report,auc)

traindata = pd.read_csv('kddtrain.csv', header=None)
testdata = pd.read_csv('kddtest.csv', header=None)

X = traindata.iloc[:,1:42]
Y = traindata.iloc[:,0]
C = testdata.iloc[:,0]
T = testdata.iloc[:,1:42]

scaler = Normalizer().fit(X)
trainX = scaler.transform(X)

scaler = Normalizer().fit(T)
testT = scaler.transform(T)


traindata = np.array(trainX)
trainlabel = np.array(Y)

testdata = np.array(testT)
testlabel = np.array(C)


print("-------------------------------------------DT---------------------------------
")

model = DecisionTreeClassifier()
model.fit(traindata, trainlabel)
print(model)

# make predictions
expected = testlabel
predicted = model.predict(testdata)
proba = model.predict_proba(testdata)

np.savetxt('classical/predictedlabelDT.txt', predicted, fmt='%01d')
np.savetxt('classical/predictedprobaDT.txt', proba)

# summarize the fit of the model
y_train1 = expected
```

```
y_pred = predicted
accuracy = accuracy_score(y_train1, y_pred)
recall = recall_score(y_train1, y_pred , average="binary")
precision = precision_score(y_train1, y_pred , average="binary")
f1 = f1_score(y_train1, y_pred, average="binary")


print("---------------------------------------------")
print("accuracy")
print("%.6f" %accuracy)
print("precision")
print("%.6f" %precision)
print("racall")
print("%.6f" %recall)
print("f1score")
print("%.6f" %f1)
```

Que ejecutándose genera:

```
C:\Users\AndresValencia\Anaconda3\envs\untitled\python.exe
C:/Users/AndresValencia/Documents/GitHub/TF1.14/Intrusion-Detection-Systems-
master/DecissionTree.py
----------------------------------------DT-------------------------------
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
                       max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort=False,
                       random_state=None, splitter='best')
---------------------------------------------
accuracy
0.929817
precision
0.998574
racall
0.914142
f1score
0.954494

Process finished with exit code 0
```

Ofreciendo una precisión de un **92.9817%**

4.1.2 Logistic regression

En este caso, cambiando la ultima parte del código:

```
import numpy as np
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import Normalizer
from sklearn.metrics import (precision_score, recall_score,f1_score,
accuracy_score,mean_squared_error,mean_absolute_error, roc_curve,
classification_report,auc)

traindata = pd.read_csv('kddtrain.csv', header=None)
testdata = pd.read_csv('kddtest.csv', header=None)

X = traindata.iloc[:,1:42]
Y = traindata.iloc[:,0]
C = testdata.iloc[:,0]
T = testdata.iloc[:,1:42]

scaler = Normalizer().fit(X)
```

```
trainX = scaler.transform(X)

scaler = Normalizer().fit(T)
testT = scaler.transform(T)


traindata = np.array(trainX)
trainlabel = np.array(Y)

testdata = np.array(testT)
testlabel = np.array(C)


print("-----------------------------------------LR---------------------------------
")
model = LogisticRegression()
model.fit(traindata, trainlabel)

# make predictions
expected = testlabel
np.savetxt('classical/expected.txt', expected, fmt='%01d')
predicted = model.predict(testdata)
proba = model.predict_proba(testdata)

np.savetxt('classical/predictedlabelLR.txt', predicted, fmt='%01d')
np.savetxt('classical/predictedprobaLR.txt', proba)

y_train1 = expected
y_pred = predicted
accuracy = accuracy_score(y_train1, y_pred)
recall = recall_score(y_train1, y_pred , average="binary")
precision = precision_score(y_train1, y_pred , average="binary")
f1 = f1_score(y_train1, y_pred, average="binary")

print("accuracy")
print("%.6f" %accuracy)
print("precision")
print("%.6f" %precision)
print("racall")
print("%.6f" %recall)
print("f1score")
print("%.6f" %f1)
```

Ofreciendo:

```
C:\Users\AndresValencia\Anaconda3\envs\untitled\python.exe
C:/Users/AndresValencia/Documents/GitHub/TF1.14/Intrusion-Detection-Systems-
master/LogisticRegression.py
-----------------------------------------LR---------------------------------
C:\Users\AndresValencia\Anaconda3\envs\untitled\lib\site-
packages\sklearn\linear_model\logistic.py:432: FutureWarning: Default solver will
be changed to 'lbfgs' in 0.22. Specify a solver to silence this warning.
  FutureWarning)
accuracy
0.848072
precision
0.988522
racall
0.820844
f1score
0.896914

Process finished with exit code 0
```

Y Ofreciendo una precisión de **84.8072%**

4.1.3. K-Nearest Neighbours

Donde del mismo modo:

```
import numpy as np
import pandas as pd
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import Normalizer
from sklearn.metrics import (precision_score, recall_score,f1_score,
accuracy_score,mean_squared_error,mean_absolute_error, roc_curve,
classification_report,auc)

traindata = pd.read_csv('kddtrain.csv', header=None)
testdata = pd.read_csv('kddtest.csv', header=None)

X = traindata.iloc[:,1:42]
Y = traindata.iloc[:,0]
C = testdata.iloc[:,0]
T = testdata.iloc[:,1:42]

scaler = Normalizer().fit(X)
trainX = scaler.transform(X)

scaler = Normalizer().fit(T)
testT = scaler.transform(T)


traindata = np.array(trainX)
trainlabel = np.array(Y)

testdata = np.array(testT)
testlabel = np.array(C)


# fit a k-nearest neighbor model to the data
print("-------------------------------------------KNN-------------------------------
-")
model = KNeighborsClassifier()
model.fit(traindata, trainlabel)
print(model)
# make predictions
expected = testlabel
predicted = model.predict(testdata)
proba = model.predict_proba(testdata)


np.savetxt('classical/predictedlabelKNN.txt', predicted, fmt='%01d')
np.savetxt('classical/predictedprobaKNN.txt', proba)

# summarize the fit of the model

y_train1 = expected
y_pred = predicted
accuracy = accuracy_score(y_train1, y_pred)
recall = recall_score(y_train1, y_pred , average="binary")
precision = precision_score(y_train1, y_pred , average="binary")
f1 = f1_score(y_train1, y_pred, average="binary")

print("---------------------------------------------")
print("accuracy")
print("%.6f" %accuracy)
print("precision")
```

```
print("%.6f" %precision)
print("racall")
print("%.6f" %recall)
print("f1score")
print("%.6f" %f1)
```

Ofreciendo:

```
C:\Users\andre\Anaconda3\envs\untitled\python.exe "C:\Program Files\JetBrains\PyCharm
2019.1.1\helpers\pydev\pydevconsole.py" --mode=client --port=65361

import sys; print('Python %s on %s' % (sys.version, sys.platform))
sys.path.extend(['C:\\Users\\andre\\PycharmProjects\\TF1.14', 'C:/Users/andre/
PycharmProjects/TF1.14'])

PyDev console: starting.

Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)] on win32
>>> runfile('C:/Users/andre/PycharmProjects/TF1.14/Intrusion-Detection-Systems-master/K-
NearestNeighbours.py', wdir='C:/Users/andre/PycharmProjects/TF1.14/Intrusion-Detection-
Systems-master')
----------------------------------------KNN---------------------------------
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=None, n_neighbors=5, p=2,
           weights='uniform')
-----------------------------------------------
accuracy
0.928727
precision
0.998354
racall
0.912988
f1score
0.953764

```

Lo que vuelve a ofrecer una precisión de un **92.8727%**

4.1.4. Naive Bayes

El código en este caso será:

```
import numpy as np
import pandas as pd
from sklearn.naive_bayes import GaussianNB
from sklearn.preprocessing import Normalizer
from sklearn.metrics import (precision_score, recall_score,f1_score,
accuracy_score,mean_squared_error,mean_absolute_error, roc_curve,
classification_report,auc)

traindata = pd.read_csv('kddtrain.csv', header=None)
testdata = pd.read_csv('kddtest.csv', header=None)

X = traindata.iloc[:,1:42]
Y = traindata.iloc[:,0]
C = testdata.iloc[:,0]
T = testdata.iloc[:,1:42]

scaler = Normalizer().fit(X)
trainX = scaler.transform(X)

scaler = Normalizer().fit(T)
```

```
testT = scaler.transform(T)


traindata = np.array(trainX)
trainlabel = np.array(Y)

testdata = np.array(testT)
testlabel = np.array(C)


# fit a Naive Bayes model to the data
print("----------------------------------------NB----------------------------------
")
model = GaussianNB()
model.fit(traindata, trainlabel)
print(model)
# make predictions
expected = testlabel
predicted = model.predict(testdata)
proba = model.predict_proba(testdata)

np.savetxt('classical/predictedlabelNB.txt', predicted, fmt='%01d')
np.savetxt('classical/predictedprobaNB.txt', proba)

y_train1 = expected
y_pred = predicted
accuracy = accuracy_score(y_train1, y_pred)
recall = recall_score(y_train1, y_pred , average="binary")
precision = precision_score(y_train1, y_pred , average="binary")
f1 = f1_score(y_train1, y_pred, average="binary")

print("accuracy")
print("%.6f" %accuracy)
print("precision")
print("%.6f" %precision)
print("racall")
print("%.6f" %recall)
print("f1score")
print("%.6f" %f1)
```

Con el resultado:

```
C:\Users\AndresValencia\Anaconda3\envs\untitled\python.exe
C:/Users/AndresValencia/Documents/GitHub/TF1.14/Intrusion-Detection-Systems-
master/NaiveBayes.py
----------------------------------------NB----------------------------------
GaussianNB(priors=None, var_smoothing=1e-09)
accuracy
0.929470
precision
0.988398
racall
0.923242
f1score
0.954710

Process finished with exit code 0
```

Con una precisión de un **92.9470%**

4.1.5. Adaboost

En este caso el Código es:

```
import numpy as np
import pandas as pd
from sklearn.ensemble import AdaBoostClassifier
from sklearn.preprocessing import Normalizer
from sklearn.metrics import (precision_score, recall_score,f1_score,
accuracy_score,mean_squared_error,mean_absolute_error, roc_curve,
classification_report,auc)

traindata = pd.read_csv('kddtrain.csv', header=None)
testdata = pd.read_csv('kddtest.csv', header=None)

X = traindata.iloc[:,1:42]
Y = traindata.iloc[:,0]
C = testdata.iloc[:,0]
T = testdata.iloc[:,1:42]

scaler = Normalizer().fit(X)
trainX = scaler.transform(X)

scaler = Normalizer().fit(T)
testT = scaler.transform(T)


traindata = np.array(trainX)
trainlabel = np.array(Y)

testdata = np.array(testT)
testlabel = np.array(C)


print("-------------------------------------------Adaboost---------------------------
------")

model = AdaBoostClassifier(n_estimators=100)
model.fit(traindata, trainlabel)

# make predictions
expected = testlabel
predicted = model.predict(testdata)
proba = model.predict_proba(testdata)

np.savetxt('classical/predictedlabelAB.txt', predicted, fmt='%01d')
np.savetxt('classical/predictedprobaAB.txt', proba)
# summarize the fit of the model

y_train1 = expected
y_pred = predicted
accuracy = accuracy_score(y_train1, y_pred)
recall = recall_score(y_train1, y_pred , average="binary")
precision = precision_score(y_train1, y_pred , average="binary")
f1 = f1_score(y_train1, y_pred, average="binary")


print("-----------------------------------------------")
print("accuracy")
print("%.6f" %accuracy)
print("precision")
print("%.6f" %precision)
print("racall")
print("%.6f" %recall)
print("f1score")
print("%.6f" %f1)
```

Con el Resultado:

```
C:\Users\AndresValencia\Anaconda3\envs\untitled\python.exe
C:/Users/AndresValencia/Documents/GitHub/TF1.14/Intrusion-Detection-Systems-
master/AdaBoost.py
----------------------------------------Adaboost---------------------------------
-----------------------------------------
accuracy
0.924830
precision
0.995267
racall
0.910975
f1score
0.951257

Process finished with exit code 0
```

Y llegando de nuevo **al 92.4830%**

4.1.6 Random Forests

Codigo:

```
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import Normalizer
from sklearn.metrics import (precision_score, recall_score,f1_score,
accuracy_score,mean_squared_error,mean_absolute_error, roc_curve,
classification_report,auc)

traindata = pd.read_csv('kddtrain.csv', header=None)
testdata = pd.read_csv('kddtest.csv', header=None)

X = traindata.iloc[:,1:42]
Y = traindata.iloc[:,0]
C = testdata.iloc[:,0]
T = testdata.iloc[:,1:42]

scaler = Normalizer().fit(X)
trainX = scaler.transform(X)

scaler = Normalizer().fit(T)
testT = scaler.transform(T)


traindata = np.array(trainX)
trainlabel = np.array(Y)

testdata = np.array(testT)
testlabel = np.array(C)



print("-------------------------------------RF-----------------------------------
--")

model = RandomForestClassifier(n_estimators=100)
model = model.fit(traindata, trainlabel)

# make predictions
expected = testlabel
predicted = model.predict(testdata)
proba = model.predict_proba(testdata)
np.savetxt('classical/predictedlabelRF_1.txt', predicted, fmt='%01d')
```

```
np.savetxt('classical/predictedprobaRF_1.txt', proba)
# summarize the fit of the model

y_train1 = expected
y_pred = predicted
accuracy = accuracy_score(y_train1, y_pred)
recall = recall_score(y_train1, y_pred , average="binary")
precision = precision_score(y_train1, y_pred , average="binary")
f1 = f1_score(y_train1, y_pred, average="binary")


print("-------------------------------------------")
print("accuracy")
print("%.6f" %accuracy)
print("precision")
print("%.6f" %precision)
print("racall")
print("%.6f" %recall)
print("f1score")
print("%.6f" %f1)
```

Resultado:

```
C:\Users\andre\Anaconda3\envs\untitled\python.exe "C:\Program Files\JetBrains\PyCharm 2019.1.1\helpers\pydev\pydevconsole.py" --mode=client --port=65156

import sys; print('Python %s on %s' % (sys.version, sys.platform))
sys.path.extend(['C:\\Users\\andre\\PycharmProjects\\TF1.14', 'C:/Users/andre/PycharmProjects/TF1.14'])

PyDev console: starting.

Python 3.7.3 (default, Apr 24 2019, 15:29:51) [MSC v.1915 64 bit (AMD64)] on win32
>>> runfile('C:/Users/andre/PycharmProjects/TF1.14/Intrusion-Detection-Systems-master/RandomForest.py', wdir='C:/Users/andre/PycharmProjects/TF1.14/Intrusion-Detection-Systems-master')
-------------------------------------RF-------------------------------------
-------------------------------------------
accuracy
0.926441
precision
0.998720
racall
0.909809
f1score
0.952194

```

Con un **92.6441%** de nuevo.

4.1.7. Resultados utilizando técnicas de Machine Learning.

Podemos observar como los resultados ofrecen un grado de precisión máxima rozando un 93%:

Metodo	Accuracy
Arbol de decision	0.929817
Logistic Regression	0.848072
Naive Bayes	0.929470
KNN	0.928727
Adaboost	0.924830
Random Forest	0.926441

Todos estos resultados son obtenidos normalizando el Dataset y utilizando las optimizaciones por defecto, donde solamente cuando generamos el modelo según el algoritmo de regresión logística obtenemos un resultado sensiblemente inferior en torno a un 85%.

Podríamos interpretar en el caso de haber utilizado solo algunos de dichos métodos, que, si utilizando métodos diferentes convergemos siempre al mismo límite máximo de precisión, esto puede ser debido a dos factores diferentes:

a) Los métodos implementados no son lo bastante potentes para generar un modelo que ofrezca mayor precisión. Esto se ha demostrado falso en la primera implementación de un árbol de decisión en el apartado anterior.

b) El conocimiento contenido en el Dataset solo puede interpretar con éxito un número máximo de las muestras del Conjunto de evaluación. Dicho de otro modo, existen muestras en el conjunto de evaluación que no se encuentran significativamente representadas en el conjunto de entrenamiento.

En este estudio y en este momento, nuestra conjetura es que parece evidente que nos enfrentamos a lo segundo. De algún modo parece claro que, aunque los modelos de ML utilizados no fueran lo bastante potentes, el hecho de que cinco de los seis empleados converjan en la misma precisión no puede ser simplemente una casualidad. En cualquier caso, esto es por el momento solamente una suposición, y disponemos de las herramientas para intentar demostrarlo o al menos profundizar más al respecto para poder entender que está sucediendo.

Para poder proceder en esta dirección, comenzaremos la segunda parte de nuestra implementación, utilizando redes neuronales.

4.2. Implementación mediante una Red Neuronal

Emplearemos la implementación que ya hemos utilizado para los algoritmos de ML tratados en el anterior apartado, modificándolo convenientemente. Como API utilizaremos **Tensorflow** 1.13 tomando **Keras** como Wrapper, debido a la sencillez de uso y a la abundante documentación.

Describamos someramente el código que vamos a utilizar, paso a paso:

Importamos las APIs necesarias

```
from __future__ import print_function
from sklearn.model_selection import train_test_split
import pandas as pd
import numpy as np
np.random.seed(2041)  # for reproducibility
from keras.preprocessing import sequence
from keras.utils import np_utils
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Embedding
from keras.layers import LSTM, SimpleRNN, GRU
from keras.datasets import imdb
from keras.utils.np_utils import to_categorical
from sklearn.metrics import (precision_score, recall_score,f1_score,
accuracy_score,mean_squared_error,mean_absolute_error)
from sklearn import metrics
from sklearn.preprocessing import Normalizer
import h5py
import keras
from keras import callbacks
from keras.callbacks import ModelCheckpoint, EarlyStopping, ReduceLROnPlateau,
CSVLogger
import datetime
```

Inicializamos los ficheros de resultados y procedemos a leer los datasets Training.csv y Testing.csv:

```
currentDT = datetime.datetime.now()
print (str(currentDT))

filename="TestResults/TFM_Tester_"+currentDT.strftime("%Y%m%d")+"_"+currentDT.strft
ime("%H%M%S")
filenameTxt="TestResults/TFM_Tester_"+currentDT.strftime("%Y%m%d")+"_"+currentDT.st
rftime("%H%M%S")+".txt"
filenameCsv="TestResults/TFM_Tester_"+currentDT.strftime("%Y%m%d")+"_"+currentDT.st
rftime("%H%M%S")+".csv"

matrixFile=open(filenameTxt,"x")
matrixFile.write("WELCOME to Tester ----------------------------------------------
-------------\n")
matrixFile.write(str(filenameTxt))
matrixFile.write(str(currentDT))
matrixFile.write("\nWELCOME to Tester ----------------------------------------------
---------------\n")

matrixCsvFile=open(filenameCsv,"x")
matrixCsvFile.write("DataSetSize, NumberOfEpochs, LayerDimension, Normalized,
ActivationFuntion, TestLoss, TestAccuracy, WastedLoss, WastedAccuracy")

traindata = pd.read_csv('kdd/binary/Training.csv', header=None)
```

```
testdata = pd.read_csv('kdd/binary/Testing.csv', header=None)
```

Separamos las Features de las labels:

```
X = traindata.iloc[:,1:42]
Y = traindata.iloc[:,0]
C = testdata.iloc[:,0]
T = testdata.iloc[:,1:42]
```

Reservamos 10000 Datapoints para realizar la validación. Como comentamos anteriormente en la sección **2.6.3**, esto es necesario para que con cada epoch del entrenamiento del modelo podamos comparar como este se ajusta a la predicción de un numero de features que no se encontraban en el conjunto de entrenamiento, para evitar obtener un modelo que genere overfitting.

```
# Reserve 10,000 samples for validation
x_val = X[-10000:]
y_val = Y[-10000:]
X= X[:-10000]
Y= Y[:-10000]
```

Y a partir de este momento empezamos a solicitar los hiperparámetros de la Red Neuronal, siendo el primero el tamaño relativo del training set a utilizar:

```
whichSize = float(input("Chose the size of the Dataset [0.01-0.95] : "))
print("\nDataset Size : "+str(whichSize))
matrixFile.write("\nDataset Size : "+str(whichSize))
```

Una vez hecho esto lo gestionamos y seguimos solicitando parámetros:

```
X, X_wasted, Y, y_wasted = train_test_split(X, Y, test_size=whichSize,
random_state=42)

trainX = np.array(X)

testT = np.array(T)

numberOfEpochs = int(input("Choose number of Epochs [1-99] : "))
print("\nNumber of Epochs? : " + str(numberOfEpochs))
matrixFile.write("\nNumber of Epochs? : " + str(numberOfEpochs))

layerDimension = int(input("Choose hidden layer's Dimension [1-100] : "))
print("\nHidden Layer Dimension? : " + str(layerDimension))
matrixFile.write("\nHidden Layer Dimension? : " + str(layerDimension))

numberOfHiddenLayers = int(input("Choose hidden layer's number [1-10] : "))
print("\nHidden Layer Number? : " + str(layerDimension))
matrixFile.write("\nHidden Layer Number? : " + str(layerDimension))
```

```
normalized = bool(int(input("Choose if DataSet will be normalized [0/1]")))
print("\nNormalizado? : " + str(normalized == True))
matrixFile.write("\nNormalizad? : " + str(normalized == True))

activationFunction = input("Choose the activation funtion [linear, relu, tanh,
sigmoid...] : ")
print("\nActivation Function? : " + str(activationFunction))
matrixFile.write("\nActivation Function? : " + str(activationFunction))

dropoutValue = float(input("Choose dropout Value [0.001 - 0.9] : "))
print("\nDropout Value? : " + str(layerDimension))
matrixFile.write("\nDropout Value? : " + str(layerDimension))
```

Una vez definidas las condiciones de contorno procedemos a generar el modelo:

```
trainX.astype(float)
testT.astype(float)

if normalized:
    scaler = Normalizer().fit(trainX)
    trainX = scaler.transform(trainX)

    scaler = Normalizer().fit(testT)
    testT = scaler.transform(testT)

y_train = np.array(Y)
y_test = np.array(C)

X_train = np.array(trainX)
X_test = np.array(testT)

keras.backend.clear_session()

batch_size = 64

# 1. define the network

model = Sequential()
model.add(Dense(layerDimension, input_dim=41, activation=activationFunction))
print ("Adding First Hidden Layer\n")
if numberOfHiddenLayers > 1:
    for addLayer in range(2,numberOfHiddenLayers+1):
        model.add(Dense(layerDimension, activation=activationFunction))
        model.add(Dropout(dropoutValue))
        print("Adding hidden layer number : "+str(addLayer)+"\n")

model.add(Dense(1))
model.add(Activation('sigmoid'))
```

Si hubiéramos deseado seguir con un clasificador múltiple, naturalmente la última capa debería ser de 5 neuronas con activación *softmax*, que generarían la probabilidad de cada paquete de pertenecer a una de las clasificaciones – estas son disjuntas-, y asimismo la función de perdida no seria *binary_crossentropy*, sino *categorical_crossentropy* :

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
checkpointer = callbacks.ModelCheckpoint(filepath="Data/AVP_checkpoint-
```

```
{epoch:02d}.hdf5", verbose=1,
                                                    save_best_only=True,
monitor='loss')
csv_logger = CSVLogger(
    filename
    + "_NR" + str(normalized == True)
    + "_LSZ" + str(layerDimension)
    + "_NE" + str(numberOfEpochs)
    + "_AF" + str(activationFunction)
    + '_Training_set.csv', separator=',', append=False)

model.fit(X_train, y_train,
    batch_size=batch_size,
    nb_epoch=numberOfEpochs,
    validation_data=(x_val, y_val),
    callbacks=[checkpointer, csv_logger])
```

Una vez compilado el modelo procederemos a evaluar los resultados:

```
# Evaluate the model on the test data using `evaluate`
print("\n_____________________________________________________________________________
_________________")
print('\n# Evaluate on test data')
results = model.evaluate(X_test, y_test, verbose=0, batch_size=128)
print('\ntest loss, test acc:', results)
print("\n_____________________________________________________________________________
_________________\n")



matrixFile.write("\n_____________________________________________________________________________
_____________________________")
matrixFile.write('\n# Evaluate on test data')
matrixFile.write('\ntest loss, test acc:')
matrixFile.write(str(results))
matrixFile.write("\n_____________________________________________________________________________
_____________________________\n")


matrixCsvFile.write("\n" + str(whichSize) + ", ")
matrixCsvFile.write(str(numberOfEpochs) + ", ")
matrixCsvFile.write(str(layerDimension) + ", ")
matrixCsvFile.write(str(normalized == True) + ", ")
matrixCsvFile.write(str(activationFunction) + ", ")
matrixCsvFile.write(str(results[0]) + ", ")
matrixCsvFile.write(str(results[1]))



model.save(filename + str(normalized == True) + "_" + str(layerDimension) + "_" +
str(numberOfEpochs) + "model.hdf5")

matrixCsvFile.close()

matrixFile.write("\nFinished Matrix ----------------------------------------------
-------------\n")
matrixFile.write("\nElapsed Time\n")
finishedDT = datetime.datetime.now()
matrixFile.write(str(finishedDT-currentDT))
matrixFile.write("\nFinished Matrix ----------------------------------------------
-------------\n")

matrixFile.close()
```

Llegados a este punto, se nos presenta un problema. Por una parte, la potencia de computación requerida -y con ello el tiempo para calcular cada modelo- para poder realizar las pruebas necesarias generando diferentes modelos es relativamente alta, por lo que no podemos simplemente dedicarnos a seleccionar parámetros aleatoriamente hasta encontrar algo que parezca funcionar, e incluso si aceptásemos dicha estrategia como válida, no podríamos estar seguros acerca de si esa elección de parámetros, que podría ofrecer una precisión aceptable, es la que menos coste de computación requiere para dicha precisión.

Repasemos brevemente cuales son los hiperparámetros definibles en nuestro script:

- Tamaño relativo del Dataset
- Numero de epochs
- Numero de neuronas por capa oculta
- Numero de capas ocultas
- DataSet Normalizado?
- Función de activación
- Dropout value

Todas estas variables tienen asociado un coste computacional mayor o menor, siendo de fundamental importancia para dirimirlo las cuatro primeras. A mayor complejidad de la red, mayor número de muestras para generar el modelo y mayor número de iteraciones para generarlo, mayor coste computacional.

Introduzcamos ahora algunos valores y observemos los resultados. Procedamos con el caso límite de una capa oculta con una neurona con activación lineal:

```
C:\Users\AndresValencia\Anaconda3\envs\untitled\python.exe
C:/Users/AndresValencia/Documents/GitHub/TF1.14/Intrusion-Detection-Systems-
master/dnn/tester.py
Using TensorFlow backend.
2019-05-29 19:39:23.152784
Chose the size of the Dataset [0.01-0.95] : 0.001

Dataset Size : 0.001
Choose number of Epochs [1-99] : 1

Number of Epochs? : 1
Choose hidden layer's Dimension [1-100] : 1

Hidden Layer Dimension? : 1
Choose hidden layer's number [1-10] : 1

Hidden Layer Number? : 1
Choose if DataSet will be normalized [0/1]0

Normalizado? : False
Choose the activation funtion [linear, relu, tanh, sigmoid...] : linear

Activation Function? : linear
Choose dropout Value [0.001 - 0.9] : 0.01

Dropout Value? : 1
WARNING:tensorflow:From C:\Users\AndresValencia\Anaconda3\envs\untitled\lib\site-
packages\tensorflow\python\framework\op_def_library.py:263: colocate_with (from
tensorflow.python.framework.ops) is deprecated and will be removed in a future
version.
Instructions for updating:
Colocations handled automatically by placer.
Adding First Hidden Layer
```

```
_________________________________________________________________
________

# Evaluate on test data

test loss, test acc: [1.1148900528669738, 0.8110015464851462]

_________________________________________________________________
________

Process finished with exit code 0
```

Obteniendo un 81%. Puede parecer mucho, pero es solamente algo inferior al que obteníamos mediante una regresión logística. Básicamente, una neurona de activación lineal es equivalente a una combinación lineal simple de los parámetros de entrada del vector de features. Los coeficientes w_x correspondientes a cada conexión determinan el peso de cada feature a la hora de determinar si el paquete es normal o anormal. Recordemos que naturalmente, este es un Dataset sintético generado artificialmente en una red virtual, en el que la topología de la red ofrece una información probablemente excesiva acerca de la clasificación de cada uno de ellos. Es de esperar que en entornos reales nuestra precisión fuese muy inferior, pero como caso de estudio inicial es interesante y demuestra como las redes neuronales son una generalización de las técnicas de machine learning citadas y aplicadas anteriormente.

Procedamos ahora escogiendo otros valores para los hiperparámetros de la red. Existen varias heurísticas según las cuales elegir el número de neuronas y capas, casi todas coincidiendo en que no debe ser superior al número de neuronas de la capa de entrada (correspondiente al feature vector) y que el número óptimo se encuentra entre el número de neuronas de dicha capa y la de salida. El problema de las heurísticas es que, al tratarse de un conocimiento no demostrable ni debidamente documentado, no puede constituirse como una aproximación rigurosa.

No obstante, sabemos que a mayor número de neuronas, y en general a mayor complejidad en la red, mayor tendencia de esta a aprender de memoria el dataset de entrada y perder la posibilidad de generalizar conocimiento. Del mismo modo, sabemos que para problemas que describen clasificadores de vectores sin componente temporal, en pocas ocasiones es necesario establecer redes neuronales profundas con más de una capa oculta.

En todo caso, comprobémoslo para algunos valores más, todos para un DropOut del 0.01 y con una sola capa de profundidad:

DataSetSize	NumberOfEpochs	LayerDimension	Normalized	ActivationFuntion	TestLoss	TestAccuracy
0,2	10	10	FALSE	linear	0,574359198	88,408476%
0,03	5	4	TRUE	relu	0,492217786	84,663488%
0,2	10	10	TRUE	relu	0,503536655	90,254928%

Donde podemos confirmar como la precisión del modelo viene determinada por la elección de los hiperparametros.

Probemos ahora con dos capas, 41 neuronas y 10 epochs para un 0.6 del dataset:

```
C:\Users\AndresValencia\Anaconda3\envs\untitled\python.exe
C:/Users/AndresValencia/Documents/GitHub/TF1.14/Intrusion-Detection-Systems-
master/dnn/tester.py
Using TensorFlow backend.
2019-05-29 20:12:50.459923
Chose the size of the Dataset [0.01-0.95] : 0.6

Dataset Size : 0.6
Choose number of Epochs [1-99] : 10

Number of Epochs? : 10
Choose hidden layer's Dimension [1-100] : 41

Hidden Layer Dimension? : 41
Choose hidden layer's number [1-10] : 2

Hidden Layer Number? : 41
Choose if DataSet will be normalized [0/1]1

Normalizado? : True
Choose the activation funtion [linear, relu, tanh, sigmoid...] : relu

Activation Function? : relu
Choose dropout Value [0.001 - 0.9] : 0.01

Dropout Value? : 41


    64/193608 [..............................] - ETA: 22:06 - loss: 0.6962 - acc:
0.6875
  2432/193608 [..............................] - ETA: 38s - loss: 0.5954 - acc:
0.8150

[...]
170752/193608 [=========================>....] - ETA: 0s - loss: 0.0061 - acc:
0.9981
172672/193608 [=========================>....] - ETA: 0s - loss: 0.0061 - acc:
0.9981
174656/193608 [==========================>...] - ETA: 0s - loss: 0.0060 - acc:
0.9981
176576/193608 [==========================>...] - ETA: 0s - loss: 0.0061 - acc:
0.9981
178560/193608 [==========================>...] - ETA: 0s - loss: 0.0061 - acc:
0.9981
180480/193608 [==========================>...] - ETA: 0s - loss: 0.0061 - acc:
0.9981
182464/193608 [==========================>..] - ETA: 0s - loss: 0.0061 - acc:
0.9981
184448/193608 [==========================>..] - ETA: 0s - loss: 0.0062 - acc:
0.9981
186368/193608 [==========================>..] - ETA: 0s - loss: 0.0062 - acc:
0.9981
188352/193608 [==========================>.] - ETA: 0s - loss: 0.0061 - acc:
0.9981
190272/193608 [==========================>.] - ETA: 0s - loss: 0.0062 - acc:
0.9981
192128/193608 [==========================>.] - ETA: 0s - loss: 0.0062 - acc:
0.9981
193608/193608 [==============================] - 5s 28us/step - loss: 0.0062 - acc:
0.9981 - val_loss: 0.1848 - val_acc: 0.9881

Epoch 00010: loss improved from 0.00673 to 0.00620, saving model to
Data/AVP_checkpoint-10.hdf5

_____________________________________________________________________________
________

# Evaluate on test data
```

```
test loss, test acc: [0.8011051903150456, 0.9214446241402845]

_________________________________________________________________
________

Process finished with exit code 0
```

El número de precisión es el 92% se acerca mucho al máximo que obteníamos con los otros métodos de machine learning. ¿Por qué esto es así? Hemos utilizado un modelo relativamente complejo para generarlo, pero la utilización de un set de validación ha impedido que se produzca el overfitting que vendría asociado a tal complejidad, de forma que, si estábamos en lo cierto con respecto al límite de precisión que podemos obtener en base al DataSet de entrenamiento suministrado, dicha red neuronal ha podido extraer el máximo conocimiento y generalizar a partir de este.

Pero vayamos mas allá. ¿Por qué generar solamente un número limitado de modelos cuando podemos automatizar la generación de estos mediante un script y paciencia?

Utilizaremos CUDA y una NVIDIA GTX1080 para generar más -muchos más- modelos y con ello producir un espacio de configuraciones mucho más elevado que nos ayude a visualizar mejor como afectan los hiperparámetros a la precisión del modelo.

4.3 Generación de una Matriz de Redes Neuronales

El objetivo será elaborar una matriz de 108 configuraciones diferentes que nos relacione los hiperparámetros con la precisión obtenida. Por una parte, esto nos permitirá comprender mejor la relación entre aquellos (y por tanto la complejidad de cálculo requerida) y la precisión obtenida. Por otra parte, nos permitirá observar si existen patrones asociados a dichas elecciones.

El código del script será necesariamente una variación sencilla del anterior:

```
from __future__ import print_function
from sklearn.model_selection import train_test_split
import pandas as pd
import numpy as np
np.random.seed(2041)  # for reproducibility
from keras.preprocessing import sequence
from keras.utils import np_utils
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Embedding
from keras.layers import LSTM, SimpleRNN, GRU
from keras.datasets import imdb
from keras.utils.np_utils import to_categorical
from sklearn.metrics import (precision_score, recall_score,f1_score,
accuracy_score,mean_squared_error,mean_absolute_error)
from sklearn import metrics
from sklearn.preprocessing import Normalizer
import h5py
import keras
from keras import callbacks
from keras.callbacks import ModelCheckpoint, EarlyStopping, ReduceLROnPlateau,
CSVLogger
import datetime

currentDT = datetime.datetime.now()
print (str(currentDT))

filename="MatrixResults/TFM_Matrix_"+currentDT.strftime("%Y%m%d")+"_"+currentDT.str
ftime("%H%M%S")
filenameTxt="MatrixResults/TFM_Matrix_"+currentDT.strftime("%Y%m%d")+"_"+currentDT.
strftime("%H%M%S")+".txt"
filenameCsv="MatrixResults/TFM_Matrix_"+currentDT.strftime("%Y%m%d")+"_"+currentDT.
strftime("%H%M%S")+".csv"

matrixFile=open(filenameTxt,"x")
matrixFile.write("WELCOME to Matrix ----------------------------------------------
-------------\n")
matrixFile.write(str(filenameTxt))
matrixFile.write(str(currentDT))
matrixFile.write("\nWELCOME to Matrix ----------------------------------------------
---------------\n")

matrixCsvFile=open(filenameCsv,"x")
matrixCsvFile.write("DataSetSize, NumberOfEpochs, LayerDimension, Normalized,
ActivationFuntion, TestLoss, TestAccuracy, WastedLoss, WastedAccuracy")

traindata = pd.read_csv('kdd/binary/Training.csv', header=None)
testdata = pd.read_csv('kdd/binary/Testing.csv', header=None)

listOfEpochs=[3,5,10]
# listOfEpochs=[1,2] # test

hiddenLayerDimension=[4,10,25]
# hiddenLayerDimension=[2,4] # test

hiddenLayerActivations=["relu","linear"] # check API
# hiddenLayerActivations=["relu","linear"] # test
```

```
hidenLayerNumber=[1,2,3]
lossFuntions=["mean_squared_error","binary_crossentropy"]
dropoutValues=[0.3,0.1,0.01]
sizesDataset=[0.03,0.2, 0.75]
optimizersChosen=["adam","SGD"]
isNormalized=[0,1]



X = traindata.iloc[:,1:42]
Y = traindata.iloc[:,0]
C = testdata.iloc[:,0]
T = testdata.iloc[:,1:42]

# Reserve 10,000 samples for validation
x_val = X[-10000:]
y_val = Y[-10000:]
X= X[:-10000]
Y= Y[:-10000]

for whichSize in sizesDataset:
    print("\nDataset Size : "+str(whichSize))
    matrixFile.write("\nDataset Size : "+str(whichSize))

    X, X_wasted, Y, y_wasted = train_test_split(X, Y, test_size=whichSize,
random_state=42)

    trainX = np.array(X)

    testT = np.array(T)

    for numberOfEpochs in listOfEpochs:
        print("\nNumber of Epochs? : " + str(numberOfEpochs))
        matrixFile.write("\nNumber of Epochs? : " + str(numberOfEpochs))

        for layerDimension in hiddenLayerDimension:
            print("\nHidden Layer Dimension? : " + str(layerDimension))
            matrixFile.write("\nHidden Layer Dimension? : " + str(layerDimension))

            for normalized in isNormalized:
                print("\nNormalizado? : " + str(normalized == True))
                matrixFile.write("\nNormalizad? : " + str(normalized == True))

                for activationFunction in hiddenLayerActivations:
                    print("\nActivation Function? : " + str(activationFunction))
                    matrixFile.write("\nActivation Function? : " +
str(activationFunction))

                    trainX.astype(float)
                    testT.astype(float)

                    if normalized:
                        scaler = Normalizer().fit(trainX)
                        trainX = scaler.transform(trainX)

                        scaler = Normalizer().fit(testT)
                        testT = scaler.transform(testT)

                    y_train = np.array(Y)
                    y_test = np.array(C)

                    X_train = np.array(trainX)
                    X_test = np.array(testT)

                    # exit(0)

                    batch_size = 64
```

```
                    # 1. define the network
                    model = Sequential()
                    model.add(Dense(layerDimension, input_dim=41,
activation=activationFunction))
                    model.add(Dropout(0.01))
                    model.add(Dense(1))
                    model.add(Activation('sigmoid'))

                    # try using different optimizers and different optimizer
configs
                    model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])
                    checkpointer =
callbacks.ModelCheckpoint(filepath="Data/AVP_checkpoint-{epoch:02d}.hdf5",
verbose=1,
                                                              save_best_only=True,
monitor='loss')
                    csv_logger = CSVLogger(
                        filename
                        + "_NR" + str(normalized == True)
                        + "_LSZ" + str(layerDimension)
                        + "_NE" + str(numberOfEpochs)
                        + "_AF" + str(activationFunction)
                        + '_Training_set.csv', separator=',', append=False)

                    model.fit(X_train, y_train,
                              batch_size=batch_size,
                              nb_epoch=numberOfEpochs,
                              validation_data=(x_val, y_val),
                              callbacks=[checkpointer, csv_logger])

                    # Evaluate the model on the test data using `evaluate`
                    print(

"\n_____________________________________________________________________________
_____________")
                    print('\n# Evaluate on test data')
                    results = model.evaluate(X_test, y_test,
                                             verbose=0,
                                             batch_size=128)
                    print('\ntest loss, test acc:', results)
                    print(

"\n_____________________________________________________________________________
_____________\n")

                    # Evaluate the model on the test data using `evaluate`
                    print(

"\n_____________________________________________________________________________
_____________")
                    print('\n# Evaluate on wasted data')
                    resultsWasted = model.evaluate(X_wasted, y_wasted,
                                             verbose=0,
                                             batch_size=128)
                    print('\ntest loss, test acc:', resultsWasted)
                    print(

"\n_____________________________________________________________________________
_____________\n")

                    matrixFile.write(

"\n_____________________________________________________________________________
_____________")
                    matrixFile.write('\n# Evaluate on test data')
                    matrixFile.write('\ntest loss, test acc:')
```

```
                    matrixFile.write(str(results))
                    matrixFile.write(

"\n_____________________________________________________________________________
___________\n")

                    matrixFile.write(

"\n_____________________________________________________________________________
___________")
                    matrixFile.write('\n# Evaluate on wasted data')
                    matrixFile.write('\ntest loss, test acc:')
                    matrixFile.write(str(resultsWasted))
                    matrixFile.write(

"\n_____________________________________________________________________________
___________\n")

                    matrixCsvFile.write("\n" + str(whichSize) + ", ")
                    matrixCsvFile.write(str(numberOfEpochs) + ", ")
                    matrixCsvFile.write(str(layerDimension) + ", ")
                    matrixCsvFile.write(str(normalized == True) + ", ")
                    matrixCsvFile.write(str(activationFunction) + ", ")
                    matrixCsvFile.write(str(results[0]) + ", ")
                    matrixCsvFile.write(str(results[1]) + ", ")
                    matrixCsvFile.write(str(resultsWasted[0]) + ", ")
                    matrixCsvFile.write(str(resultsWasted[1]))



                    model.save(filename + str(normalized == True) + "_" +
str(layerDimension) + "_" + str(
                        numberOfEpochs) + "model.hdf5")

                    keras.backend.clear_session()




matrixCsvFile.close()

matrixFile.write("\nFinished Matrix ------------------------------------------------
-------------\n")
matrixFile.write("\nElapsed Time\n")
finishedDT = datetime.datetime.now()
matrixFile.write(str(finishedDT-currentDT))
matrixFile.write("\nFinished Matrix ------------------------------------------------
-------------\n")

matrixFile.close()
```

La ventaja de este script es que generará un fichero .txt y .csv de forma automatizada que nos permitirá visualizar de forma fácil todo lo que podamos averiguar acerca de la combinación de diferentes hiperparámetros.

Recordemos que iteraremos con el siguiente conjunto:

```
listOfEpochs=[3,5,10]
hiddenLayerDimension=[4,10,25]
hiddenLayerActivations=["relu","linear"]
sizesDataset=[0.03,0.2, 0.75]
isNormalized=[0,1]
```

Y no variaremos ni el número de capas -por el coste en tiempo- ni el *DropOut*.

Obteniendo:

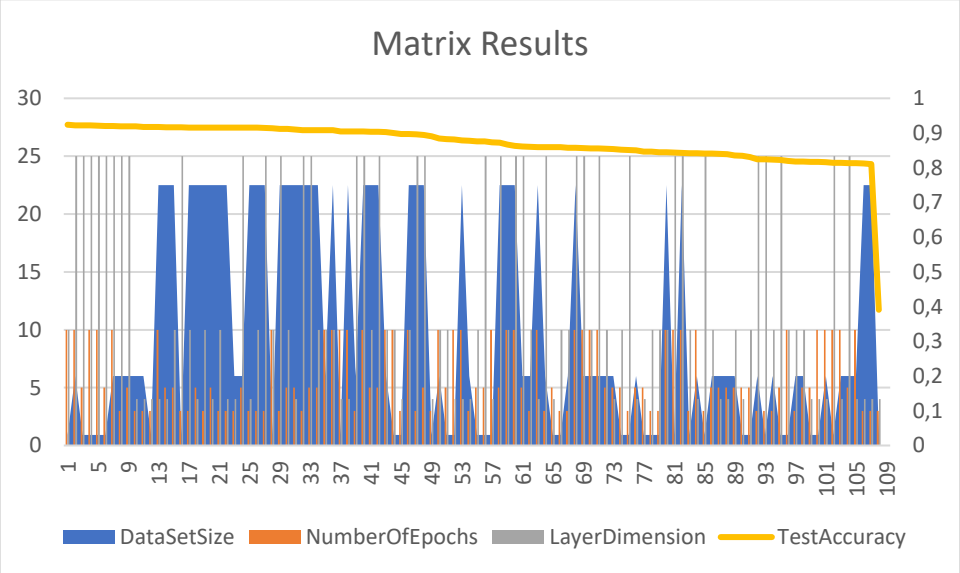
DataSetSize	NumberOfEpochs	LayerDimension	Normalized	ActivationFuntion	TestLoss	TestAccuracy
0,03	10	10	TRUE	relu	0,62688725	92,397493%
0,2	10	25	TRUE	relu	0,611601936	92,232236%
0,03	5	25	TRUE	relu	0,563611908	92,196548%
0,03	10	25	FALSE	relu	0,535784517	92,176292%
0,03	10	25	TRUE	relu	0,667754967	92,142533%
0,03	5	25	FALSE	relu	0,671066214	92,048973%
0,2	10	25	FALSE	relu	0,599017033	91,985956%
0,2	3	25	FALSE	relu	0,569325252	91,974703%
0,2	5	25	TRUE	relu	0,61265118	91,927441%
0,2	3	4	FALSE	relu	1,24076571	91,893360%
0,2	3	4	FALSE	linear	1,184571367	91,787583%
0,03	3	4	FALSE	linear	1,204960138	91,774079%
0,75	10	4	FALSE	relu	0,441433233	91,742249%
0,75	5	4	FALSE	relu	0,363524531	91,698202%
0,75	5	10	TRUE	relu	0,48480004	91,664443%
0,03	3	25	TRUE	relu	0,665157382	91,636793%
0,75	3	10	TRUE	relu	0,363458176	91,590816%
0,75	5	4	TRUE	linear	0,390667142	91,590173%
0,75	3	10	FALSE	relu	0,384424945	91,586958%
0,75	5	4	FALSE	linear	0,384949208	91,582135%
0,75	3	10	TRUE	linear	0,363081738	91,575062%
0,75	3	4	TRUE	relu	0,338398605	91,571204%
0,2	3	4	TRUE	relu	0,401070015	91,567346%
0,2	5	25	FALSE	relu	0,580264438	91,546126%
0,75	3	4	TRUE	linear	0,343701839	91,541946%
0,75	3	10	FALSE	linear	0,35651744	91,538731%
0,75	3	25	TRUE	linear	0,413626468	91,452566%
0,03	10	10	FALSE	relu	0,640207614	91,374116%
0,75	3	25	FALSE	linear	0,435864762	91,162882%
0,75	5	10	FALSE	linear	0,43206277	91,157738%
0,75	5	4	TRUE	relu	0,400894832	90,999232%
0,75	3	25	TRUE	relu	0,441609312	90,871912%
0,75	5	25	TRUE	relu	0,518516045	90,814040%
0,75	5	10	FALSE	relu	0,423053234	90,809860%
0,2	10	10	FALSE	relu	0,619352414	90,793785%
0,75	10	10	TRUE	relu	0,481322734	90,793141%
0,03	10	4	FALSE	relu	0,687193897	90,483524%
0,75	10	4	TRUE	linear	0,463343845	90,474200%
0,2	3	25	TRUE	relu	0,492100817	90,452980%
0,75	10	25	TRUE	relu	0,506737888	90,442049%
0,75	5	10	TRUE	linear	0,44405818	90,343666%
0,75	5	25	FALSE	relu	0,468862575	90,335306%
0,2	10	10	TRUE	relu	0,503536655	90,254928%
0,03	10	10	FALSE	linear	0,560798988	89,951741%
0,03	3	4	TRUE	linear	0,499951781	89,727324%
0,75	10	10	TRUE	linear	0,505992129	89,720573%
0,75	3	25	FALSE	relu	0,424583339	89,634407%
0,75	5	25	FALSE	linear	0,470669988	89,446000%
0,03	3	4	TRUE	relu	0,482435311	89,081082%
0,2	10	10	FALSE	linear	0,574359198	88,408476%
0,03	5	10	FALSE	relu	0,526261398	88,269904%
0,03	10	4	TRUE	linear	0,571245321	88,161233%
0,75	10	4	FALSE	linear	0,49201244	87,843899%

0,2	3	4	TRUE	linear	0,516211979	87,825573%
0,03	5	10	TRUE	relu	0,580239721	87,602121%
0,03	5	25	FALSE	linear	0,585334225	87,579615%
0,03	10	4	TRUE	relu	0,579998879	87,360343%
0,75	5	25	TRUE	linear	0,500830208	87,272248%
0,75	10	10	FALSE	linear	0,539624748	86,698346%
0,75	10	25	FALSE	relu	0,672321746	86,343717%
0,2	5	25	TRUE	linear	0,57475775	86,159168%
0,2	3	10	TRUE	relu	0,502903072	86,014487%
0,75	10	4	TRUE	relu	0,370001283	85,961759%
0,2	3	25	FALSE	linear	0,568287252	85,957258%
0,03	5	4	FALSE	linear	0,569063894	85,956293%
0,03	3	10	TRUE	linear	0,553519487	85,938932%
0,2	3	10	TRUE	linear	0,550204648	85,761778%
0,75	10	25	TRUE	linear	0,545631804	85,757920%
0,2	10	25	FALSE	linear	0,611669127	85,648284%
0,2	10	10	TRUE	linear	0,585693054	85,613560%
0,2	10	25	TRUE	linear	0,621139847	85,611953%
0,2	5	10	TRUE	relu	0,605193624	85,517106%
0,2	5	4	FALSE	linear	0,557495056	85,397825%
0,03	5	10	TRUE	linear	0,593452799	85,184983%
0,03	3	25	FALSE	linear	0,572227029	85,066666%
0,2	5	4	TRUE	relu	0,596316142	85,042552%
0,03	5	4	TRUE	relu	0,492217786	84,663488%
0,03	3	10	FALSE	linear	0,569621101	84,658022%
0,03	3	10	TRUE	relu	0,539107023	84,459005%
0,75	10	10	FALSE	relu	0,520992114	84,443573%
0,03	10	25	TRUE	linear	0,651476169	84,395346%
0,75	10	25	FALSE	linear	0,577141606	84,283459%
0,03	3	10	FALSE	relu	0,713503163	84,189256%
0,2	10	4	TRUE	linear	0,609550756	84,145851%
0,03	3	25	TRUE	linear	0,584153885	84,141029%
0,2	5	10	FALSE	linear	0,600115043	84,135563%
0,2	5	4	FALSE	relu	0,556538091	83,964196%
0,2	5	4	TRUE	linear	0,586289318	83,943619%
0,2	5	10	FALSE	relu	0,602684595	83,526938%
0,03	5	4	TRUE	linear	0,603185291	83,464886%
0,03	5	10	FALSE	linear	0,594119275	83,069424%
0,2	3	25	TRUE	linear	0,606272791	82,473338%
0,03	3	25	FALSE	relu	0,655762183	82,419967%
0,2	3	10	FALSE	linear	0,588572068	82,383636%
0,03	5	25	TRUE	linear	0,629424188	82,213234%
0,03	10	10	TRUE	linear	0,654386956	81,970813%
0,2	3	10	FALSE	relu	0,604195542	81,770510%
0,2	5	10	TRUE	linear	0,642430197	81,763115%
0,03	5	4	FALSE	relu	0,644792302	81,675985%
0,03	10	4	FALSE	linear	0,641439464	81,666983%
0,2	10	4	FALSE	linear	0,648103851	81,632902%
0,03	10	25	FALSE	linear	0,737902882	81,417488%
0,2	10	4	TRUE	relu	0,548351409	81,407200%
0,2	5	25	FALSE	linear	0,662009413	81,355436%
0,2	10	4	FALSE	relu	0,595068922	81,292420%
0,75	3	4	FALSE	relu	0,906976845	81,200788%
0,75	3	4	FALSE	linear	3,01506848	81,012703%
0,03	3	4	FALSE	relu	9,722336817	39,089603%

4.4. Interpretación de los resultados.

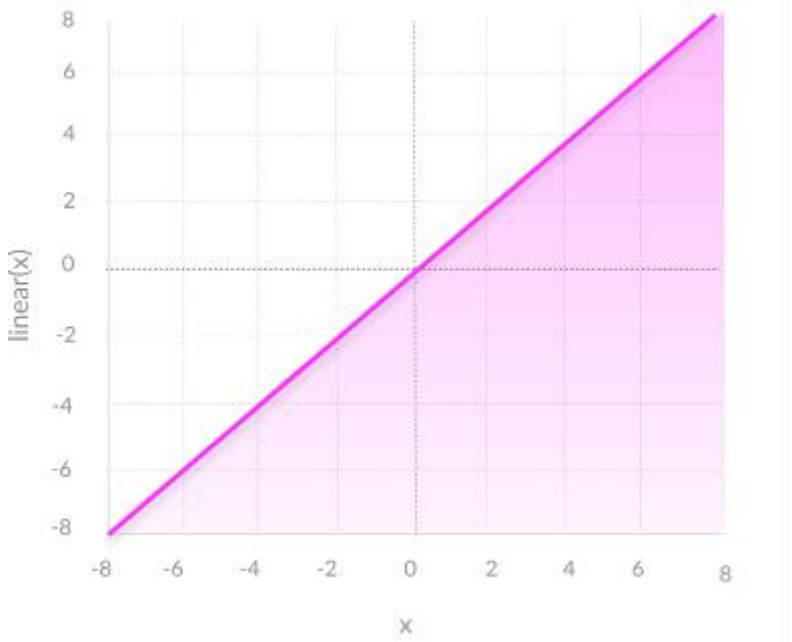
En función de este Conjunto de resultados con 108 configuraciones diferentes de 5 variables cada una y un resultado, podemos empezar a determinar varias cosas:

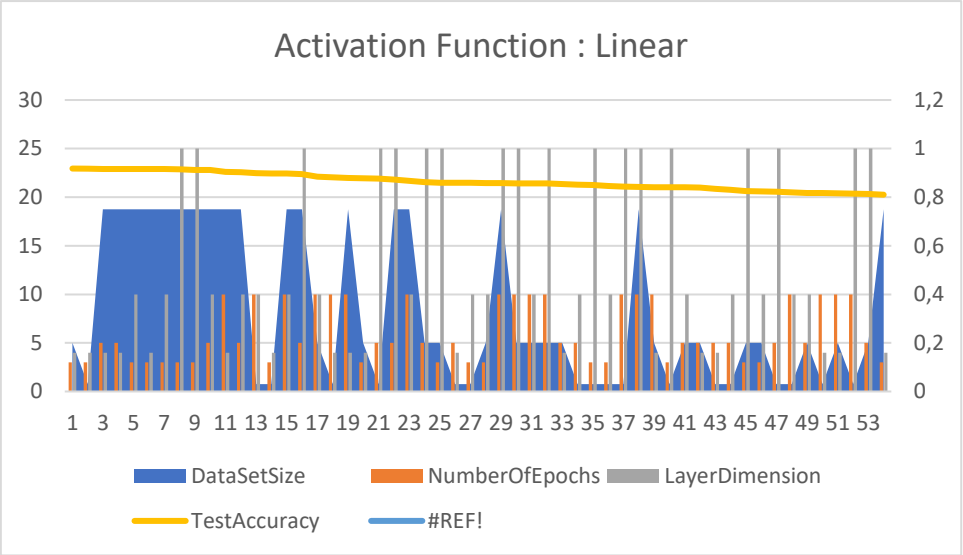
La primera y más importante es que el límite de precisión parecer estar realmente determinado por los conjuntos de datos suministrado para la generación del modelo y su evaluación. Sin importar que modelo de ML escojamos o cuan compleja sea la red neuronal implementada, parece haber un límite en torno al 93% de precisión. La media de precisión obtenida es de un 87,127150%



Pasemos ahora a discutir como parecen afectar los diferentes hiperparámetros a la precisión obtenida.

4.4.1 Resultados para la función de activación lineal:



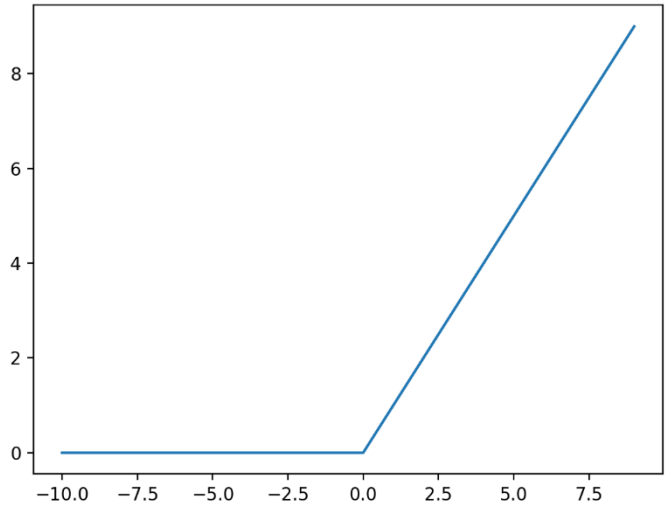


DataSetSize	NumberOfEpochs	LayerDimension	Normalized	ActivationFuntion	TestLoss	TestAccuracy
0,2	3	4	FALSE	linear	1,184571367	91,787583%
0,03	3	4	FALSE	linear	1,204960138	91,774079%
0,75	5	4	TRUE	linear	0,390667142	91,590173%
0,75	5	4	FALSE	linear	0,384949208	91,582135%
0,75	3	10	TRUE	linear	0,363081738	91,575062%
0,75	3	4	TRUE	linear	0,343701839	91,541946%
0,75	3	10	FALSE	linear	0,35651744	91,538731%
0,75	3	25	TRUE	linear	0,413626468	91,452566%
0,75	3	25	FALSE	linear	0,435864762	91,162882%
0,75	5	10	FALSE	linear	0,43206277	91,157738%

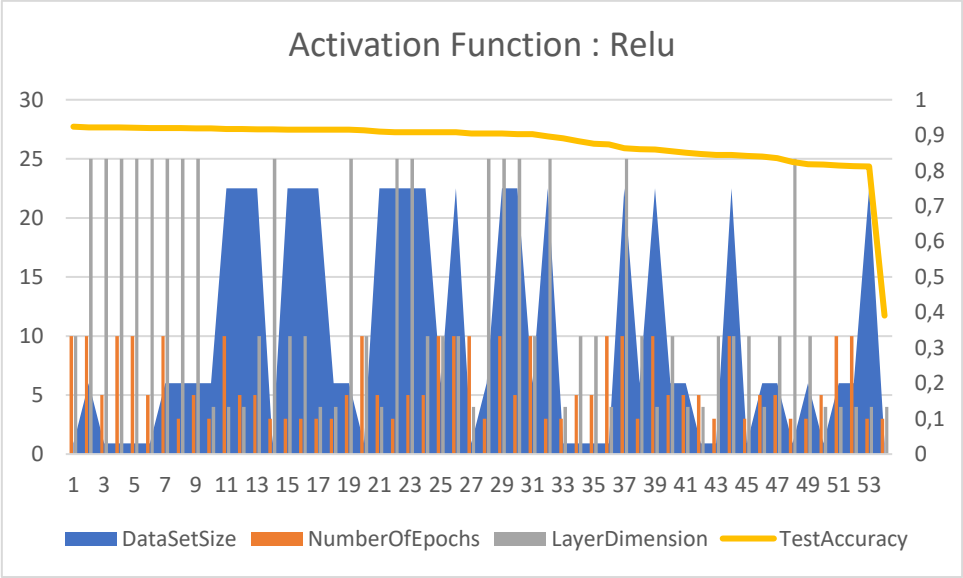
Donde tenemos 10 valores por encima de un 91%, con una media total de un 86,496245%. Esto está por debajo del valor medio de la matriz de configuraciones y concuerda con nuestra expectativa de que al utilizar una función de activación lineal en las neuronas la red neuronal funciona de manera subóptima, dificultando su capacidad de generalizar conocimiento. Como hemos discutido anteriormente, el utilizar funciones lineales como activaciones produce un comportamiento lineal en la generación del clasificador. Mas rigurosamente: la función lineal ofrece como derivada una constante, no una función, lo que inhibe el mecanismo de retro propagación que involucra utilizar la regla de la cadena en cada paso hacia atrás para cada neurona a la hora de utilizar el gradiente de la función de perdida para converger hacia un mínimo.

4.4.2. Función de activación relu (Rectified Linear Unit):

La función de activación lineal rectificada es una función lineal definida por partes que emitirá la entrada directamente si es positiva; de lo contrario, emitirá cero. Se ha convertido en la función de activación predeterminada para muchos tipos de redes neuronales porque un modelo que la usa es más fácil de entrenar y, a menudo, logra un mejor rendimiento. Para redes neuronales profundas con un numero superior de capas, elimina el problema de la difusión del gradiente, aunque esto no es observable para los modelos aquí empleados.



Asimismo, ReLu es menos costoso computacionalmente que tanh y sigmoide porque involucra operaciones matemáticas más simples. Ese es un buen punto a considerar cuando estemos diseñando redes neuronales profundas.



Con una media de un 87,758055% a función de activación relu ofrece, como esperamos, una precisión por encima de la media.

DataSetSize	NumberOfEpochs	LayerDimension	Normalized	ActivationFuntion	TestLoss	TestAccuracy
0,03	10	10	TRUE	relu	0,62688725	92,397493%
0,2	10	25	TRUE	relu	0,611601936	92,232236%
0,03	5	25	TRUE	relu	0,563611908	92,196548%
0,03	10	25	FALSE	relu	0,535784517	92,176292%
0,03	10	25	TRUE	relu	0,667754967	92,142533%
0,03	5	25	FALSE	relu	0,671066214	92,048973%
0,2	10	25	FALSE	relu	0,599017033	91,985956%
0,2	3	25	FALSE	relu	0,569325252	91,974703%
0,2	5	25	TRUE	relu	0,61265118	91,927441%
0,2	3	4	FALSE	relu	1,24076571	91,893360%
0,75	10	4	FALSE	relu	0,441433233	91,742249%
0,75	5	4	FALSE	relu	0,363524531	91,698202%
0,75	5	10	TRUE	relu	0,48480004	91,664443%
0,03	3	25	TRUE	relu	0,665157382	91,636793%
0,75	3	10	TRUE	relu	0,363458176	91,590816%
0,75	3	10	FALSE	relu	0,384424945	91,586958%
0,75	3	4	TRUE	relu	0,338398605	91,571204%
0,2	3	4	TRUE	relu	0,401070015	91,567346%
0,2	5	25	FALSE	relu	0,580264438	91,546126%
0,03	10	10	FALSE	relu	0,640207614	91,374116%

4.4.3 Numero de epochs = 3:

Los resultados obtenidos para tres iteraciones de entrenamiento de la red neuronal ofrecen una precisión media de un 86,577341% y solamente 14 resultados por encima de 91%

DataSetSize	NumberOfEpochs	LayerDimension	Normalized	ActivationFuntion	TestLoss	TestAccuracy
0,2	3	25	FALSE	relu	0,569325252	91,974703%
0,2	3	4	FALSE	relu	1,24076571	91,893360%
0,2	3	4	FALSE	linear	1,184571367	91,787583%
0,03	3	4	FALSE	linear	1,204960138	91,774079%
0,03	3	25	TRUE	relu	0,665157382	91,636793%
0,75	3	10	TRUE	relu	0,363458176	91,590816%
0,75	3	10	FALSE	relu	0,384424945	91,586958%
0,75	3	10	TRUE	linear	0,363081738	91,575062%
0,75	3	4	TRUE	relu	0,338398605	91,571204%
0,2	3	4	TRUE	relu	0,401070015	91,567346%
0,75	3	4	TRUE	linear	0,343701839	91,541946%
0,75	3	10	FALSE	linear	0,35651744	91,538731%
0,75	3	25	TRUE	linear	0,413626468	91,452566%
0,75	3	25	FALSE	linear	0,435864762	91,162882%

Aquí tenemos que hacer una pausa y tener en cuenta que al haber reservado un subconjunto de datos de entrenamiento para realizar la validación no deberemos preocuparnos por el overfitting y podemos añadir epochs sin perder precisión en el conjunto de datos de evaluación, pero en condiciones en las que esto no se cumpliese, seria necesario valorar a partir de que punto la red neuronal empieza a dejar de generalizar y procede a memorizar el dataset, si es que tiene capacidad estructural para ello.

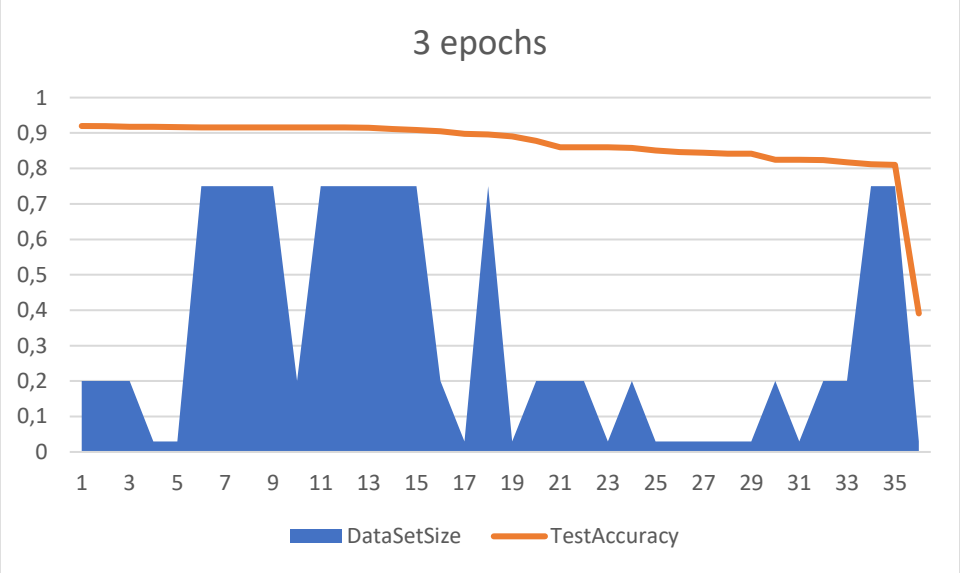
Aun así, en condiciones en las que el overfitting no es el problema a resolver, seguirá siendo importante valorar a partir de cuantas iteraciones el entrenamiento de la red llega a un máximo de precisión, dado que dichas iteraciones tienen un coste computacional lineal a la hora de generar el

modelo. Aun mas importante en coste, al no ser lineal, es el añadido de neuronas por capa y el de capas, pudiendo ambos generar overfitting.

Lo fundamental en esta discusión es que, si bien no podemos identificar solamente un hiperparámetro aislado que pueda por si solo generar overfitting, una combinación entre ellos si lo hará.

Por ejemplo, si tenemos una red neuronal que se entrenara con un número de muestras *S* con un par de capas de *n* neuronas, es cierto que si aumento el numero de neuronas a *10n* favoreceremos la posibilidad de que, para minimizar la función perdida, la red neuronal utilice los valores *w* de las conexiones entre neuronas como una memoria en lugar de como los descriptores de una función clasificadora compleja. Pero tan pronto como variemos otro de los hiperparámetros, reduciendo el numero de capas a una (disminuyendo la capacidad de memoria de la red) o aumentando el numero de muestras *S* a *10S*, esto dejara de producirse.

Veremos esto con mas facilidad analizando los subsiguientes ejemplos. Volvamos al análisis del número de iteraciones:

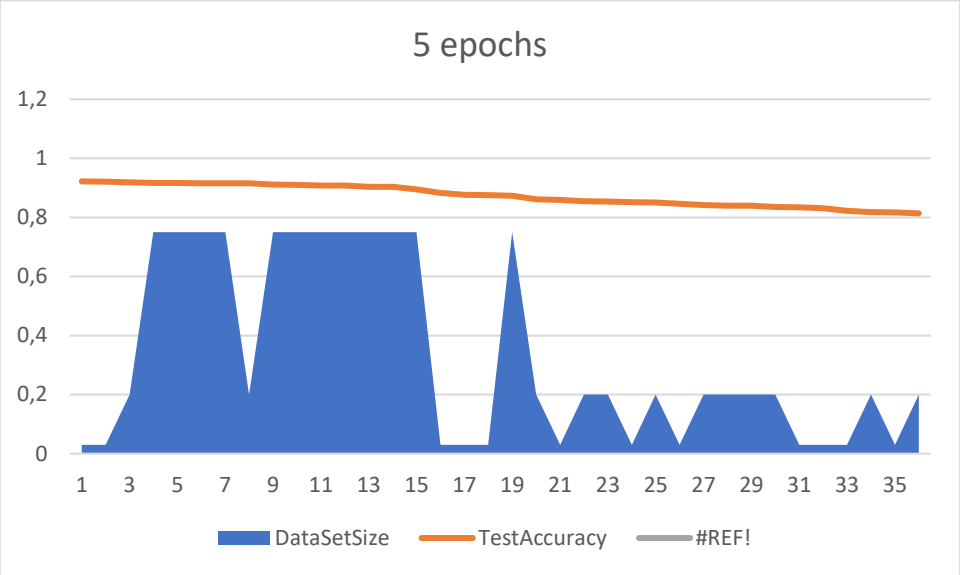


4.4.4 Numero de epochs = 5:

DataSetSize	NumberOfEpochs	LayerDimension	Normalized	ActivationFuntion	TestLoss	TestAccuracy
0,03	5	25	TRUE	relu	0,563611908	92,196548%
0,03	5	25	FALSE	relu	0,671066214	92,048973%
0,2	5	25	TRUE	relu	0,61265118	91,927441%
0,75	5	4	FALSE	relu	0,363524531	91,698202%
0,75	5	10	TRUE	relu	0,48480004	91,664443%
0,75	5	4	TRUE	linear	0,390667142	91,590173%
0,75	5	4	FALSE	linear	0,384949208	91,582135%
0,2	5	25	FALSE	relu	0,580264438	91,546126%
0,75	5	10	FALSE	linear	0,43206277	91,157738%

Con un numero de iteraciones igual a cinco, la precisión media obtenida es de un 87,386600%, lo que es superior a las tres anteriores como era de esperar, pero solo obtendríamos 9 configuraciones

con una precisión superior al 91%. Podemos interpretar esto como el resultado de que el modelo empiece a acercarse al limite en el cual en condiciones normales se produciría overfitting. Aunque este no tiene lugar debido al uso de datos de evaluación que son comprobados tras cada iteración para ajustar el modelo frenando la memorización, se produce en algunas configuraciones una perdida de precisión. Si tenemos razón, en el siguiente apartado esta tendencia se mantendrá: mejor precisión media, mejor precisión máxima pero menos resultados por encima del rango escogido. ¿Cuáles son estas configuraciones? Aquellas que añaden complejidad a la red disminuyendo el numero de muestras de aprendizaje, como podemos ver en el siguiente y subsiguientes gráficos.



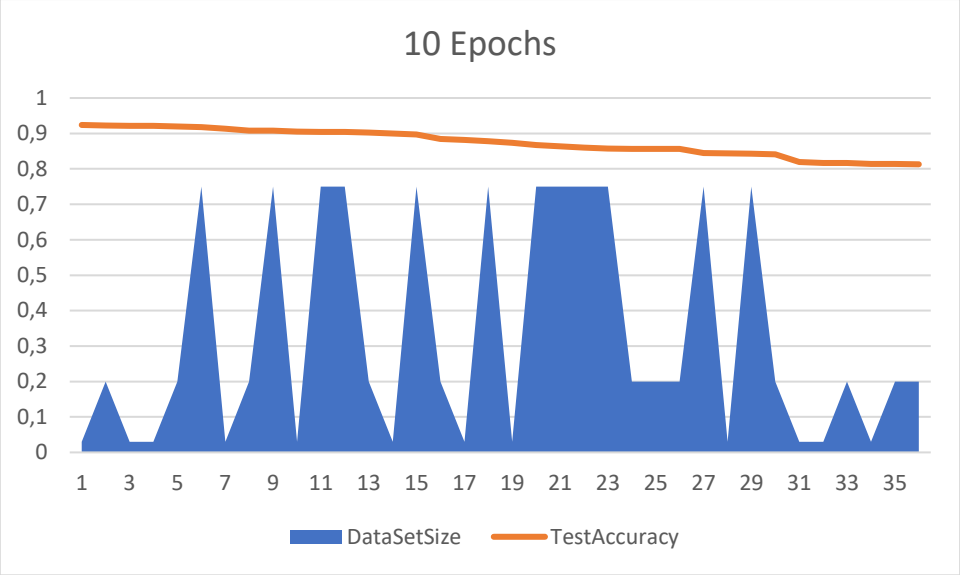
Importante de nuevo recalcar lo que ya hemos repetido anteriormente: No estamos observando overfitting.

Estamos viendo como en la generación de modelos que potencialmente debido a su complejidad puedan producir overfitting, tiene lugar un comportamiento que parece uniforme y repetido con respecto a las precisiones obtenidas -que mejoran- y al número de configuraciones superior a un valor dado -que empeoran-. Nuestra hipótesis es que, al evaluarse el modelo obtenido con el conjunto de datos de validación, acaba detectándose que pierde capacidad de generalización, pudiendo producirse el ligero frenado en el entrenamiento del modelo que posibilita el comportamiento observado.

4.4.5. Numero de epochs = 10:

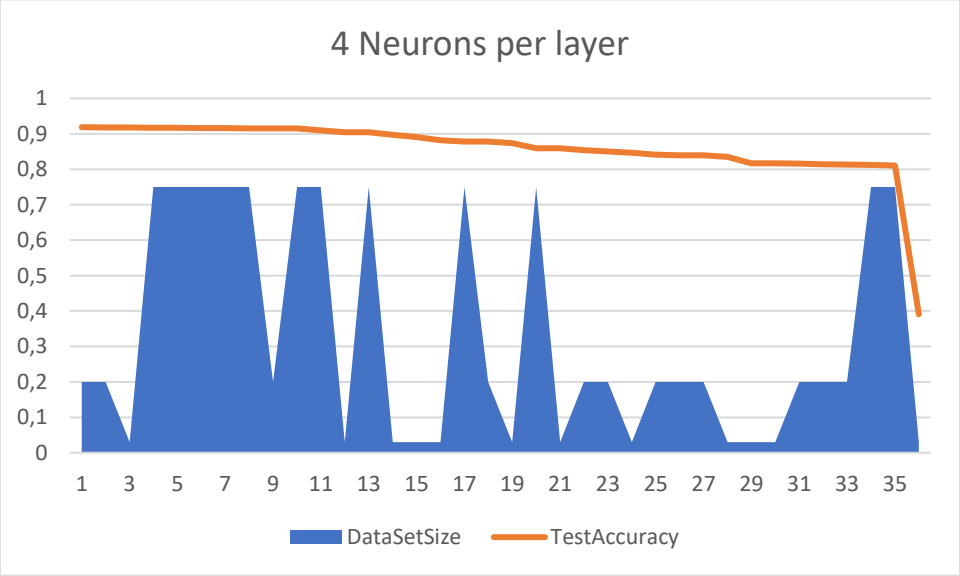
DataSetSize	NumberOfEpochs	LayerDimension	Normalized	ActivationFuntion	TestLoss	TestAccuracy
0,03	10	10	TRUE	relu	0,62688725	92,397493%
0,2	10	25	TRUE	relu	0,611601936	92,232236%
0,03	10	25	FALSE	relu	0,535784517	92,176292%
0,03	10	25	TRUE	relu	0,667754967	92,142533%
0,2	10	25	FALSE	relu	0,599017033	91,985956%
0,75	10	4	FALSE	relu	0,441433233	91,742249%
0,03	10	10	FALSE	relu	0,640207614	91,374116%

Podemos observar que la hipótesis antes descrita parece cumplirse: mejor precisión media, mejor precisión máxima, menos configuraciones (solamente 7) por encima del 91%



4.4.6. Numero de neuronas de la capa oculta = 4 :

DataSetSize	NumberOfEpochs	LayerDimension	Normalized	ActivationFuntion	TestLoss	TestAccuracy
0,2	3	4	FALSE	relu	1,24076571	91,893360%
0,2	3	4	FALSE	linear	1,184571367	91,787583%
0,03	3	4	FALSE	linear	1,204960138	91,774079%
0,75	10	4	FALSE	relu	0,441433233	91,742249%
0,75	5	4	FALSE	relu	0,363524531	91,698202%
0,75	5	4	TRUE	linear	0,390667142	91,590173%
0,75	5	4	FALSE	linear	0,384949208	91,582135%
0,75	3	4	TRUE	relu	0,338398605	91,571204%
0,2	3	4	TRUE	relu	0,401070015	91,567346%
0,75	3	4	TRUE	linear	0,343701839	91,541946%

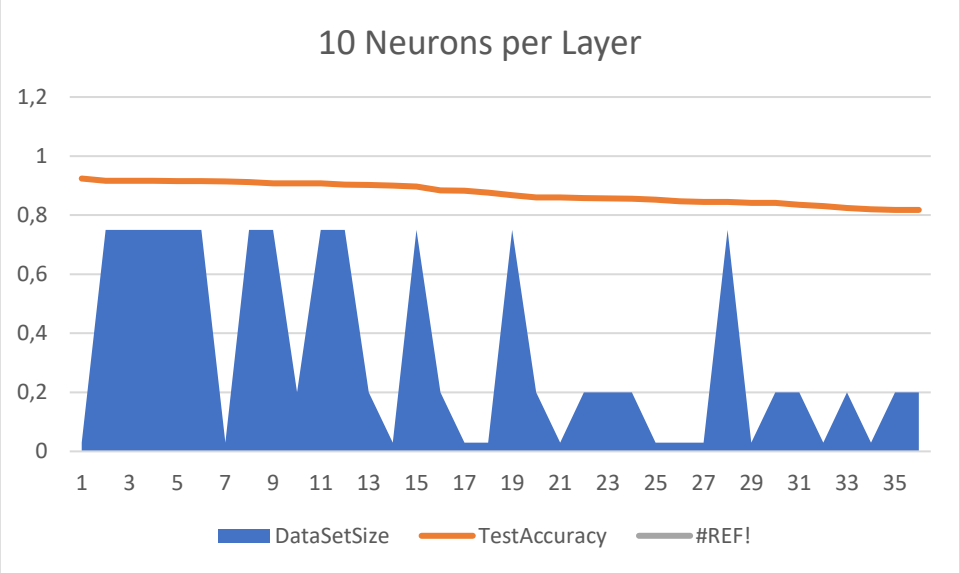


Con cuatro neuronas en la capa oculta, la precisión media es de 85,666582% y el numero de configuraciones con precisiones superiores al 91% es de 10 (casi 11).

4.4.7. Numero de neuronas en la capa Oculta = 10

Con diez neuronas en la capa oculta, la precisión media es de 87,272432%y el numero de configuraciones con precisiones superiores al 91% es de 8.

DataSetSize	NumberOfEpochs	LayerDimension	Normalized	ActivationFuntion	TestLoss	TestAccuracy
0,03	10	10	TRUE	relu	0,62688725	92,397493%
0,75	5	10	TRUE	relu	0,48480004	91,664443%
0,75	3	10	TRUE	relu	0,363458176	91,590816%
0,75	3	10	FALSE	relu	0,384424945	91,586958%
0,75	3	10	TRUE	linear	0,363081738	91,575062%
0,75	3	10	FALSE	linear	0,35651744	91,538731%
0,03	10	10	FALSE	relu	0,640207614	91,374116%
0,75	5	10	FALSE	linear	0,43206277	91,157738%



4.4.8. Numero de neuronas en la capa Oculta = 25

Finalmente, con veinticinco neuronas en la capa oculta, la precisión media es de 88,127107% y el número de configuraciones con precisiones superiores al 91% es de 12.

DataSetSize	NumberOfEpochs	LayerDimension	Normalized	ActivationFuntion	TestLoss	TestAccuracy
0,2	10	25	TRUE	relu	0,611601936	92,232236%
0,03	5	25	TRUE	relu	0,563611908	92,196548%
0,03	10	25	FALSE	relu	0,535784517	92,176292%
0,03	10	25	TRUE	relu	0,667754967	92,142533%
0,03	5	25	FALSE	relu	0,671066214	92,048973%
0,2	10	25	FALSE	relu	0,599017033	91,985956%
0,2	3	25	FALSE	relu	0,569325252	91,974703%
0,2	5	25	TRUE	relu	0,61265118	91,927441%
0,03	3	25	TRUE	relu	0,665157382	91,636793%
0,2	5	25	FALSE	relu	0,580264438	91,546126%
0,75	3	25	TRUE	linear	0,413626468	91,452566%
0,75	3	25	FALSE	linear	0,435864762	91,162882%

Del mismo modo que en el caso anterior con el sucesivo aumento de epochs, el aumento de neuronas en la capa oculta produce mejores resultados de media y mejores precisiones máximas para determinadas configuraciones, aunque en este caso el número de configuraciones con precisiones superiores a 91% no decrece uniformemente.

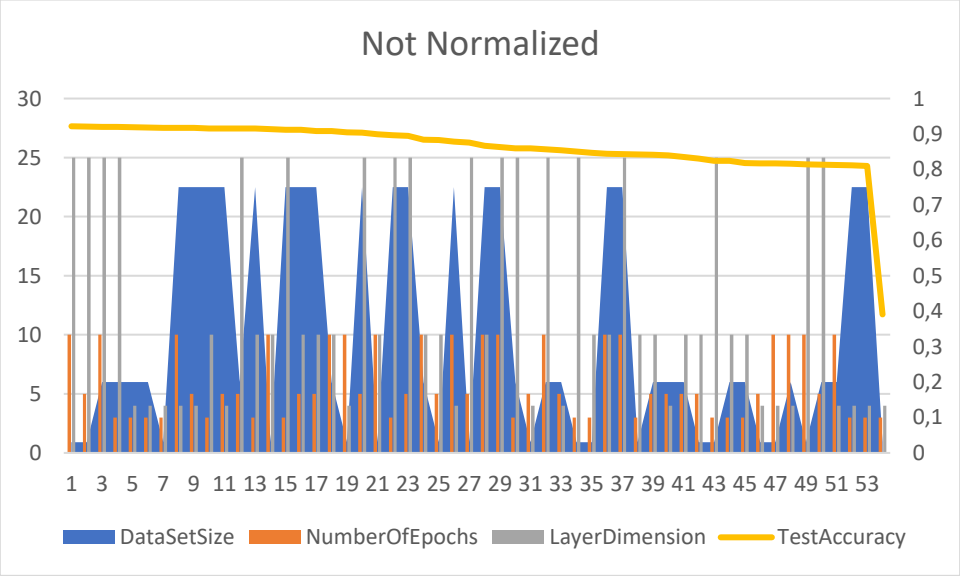
Al respecto de estos resultados podemos realizar dos precisiones:

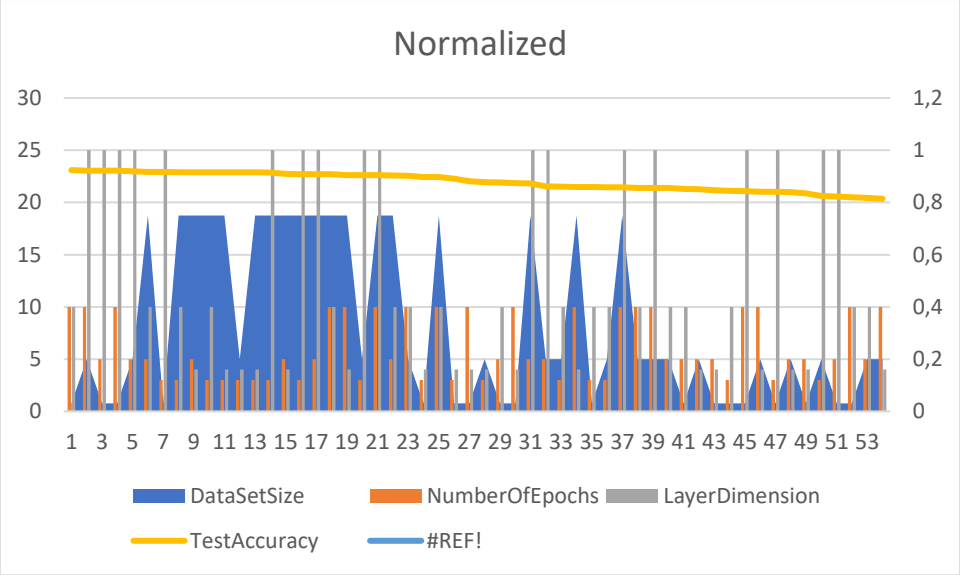
La primera es que si bien aumentar el numero de epochs o el numero de neuronas en la capa oculta produce de media mejores resultados así como mejores precisiones absolutas, es la combinacion de los diferentes hiperparámetros la que produce unos resultados mejores o peores, o lo que es lo mismo: la maximización de dichos hiperparámetros (25 neuronas, 10 epochs, relu y Normalizacion) no ofrece la precisión máxima absoluta obtenida en todas las configuraciones probadas, dándonos unos resultados por encima del 92% para Datasets de 0,03 y 0,2 de tamaño pero quedándose en un 90% para el dataset de 0'75.

La segunda es que naturalmente todas las configuraciones que añadan complejidad requerirán de un coste de computación mas elevado. Este valor, que no forma parte del espacio de hiperparámetros ni de los resultados de precisión obtenidos, no debe desdeñarse ya que en casos mas complejos que el tratado puede determinar la posibilidad de generar un modelo ajustándose a determinadas condiciones. Naturalmente en los casos en los que la computación fuera demasiado costosa, otras técnicas de feature engineering podrían entrar en juego para reducirla en la medida de lo posible.

4.9. Normalizacion del conjunto de features

Por último, contrastaremos los resultados obtenidos en configuraciones normalizadas versus no normalizadas:





Obteniendo de media un 86,20% vs un 87,85% a favor del conjunto de configuraciones normalizables, como era de esperar, estando las precisiones máximas absolutas a favor asimismo de los dataset previamente tratados por un pequeño margen.

De forma significativa, si observamos las precisiones mas altas no normalizadas veremos que existe cierta uniformidad en las configuraciones que las producen:

DataSetSize	NumberOfEpochs	LayerDimension	Normalized	ActivationFuntion	TestLoss	TestAccuracy
0,03	10	25	FALSE	relu	0,535784517	92,176292%
0,03	5	25	FALSE	relu	0,671066214	92,048973%
0,2	10	25	FALSE	relu	0,599017033	91,985956%
0,2	3	25	FALSE	relu	0,569325252	91,974703%

En efecto, los cuatro primeros valores están copados por configuraciones con el máximo de 25 neuronas por capa y función de activación relu. Nuestra conjetura al observar dicho comportamiento es que si bien la normalización del conjunto de datos utilizado para el entrenamiento del modelo optimiza la generación de dicho modelo (a misma complejidad y topología de la red neuronal), dicha normalización no será estrictamente necesaria en el caso de que la red neuronal disponga de los recursos suficientes. Ofrecerá un modelo menos preciso a igualdad de condiciones o un modelo mas costoso de generar si aumentamos su complejidad para obtener precisiones comparables, pero podremos llegar a resultados comparables.

O dicho de otra manera, podemos integrar una parte importante de nuestro Feature engineering en la propia generación del modelo si estamos dispuestos a asumir el coste computacional asociado. No se trata de que la red normalizará el dataset, sino que aprenderá como relacionar la disparidad en ordenes de magnitud entre diferentes features. No puede no obstante ignorarse que la normalización será a priori siempre nuestra primera opción, pero ilustra la propiedad intrínseca de los modelos NN de ingerir datos y generalizar conocimiento. Por otra parte, si utilizamos técnicas de regularización de coeficientes, su utilización será prácticamente necesaria.

6. Conclusiones

La precisión máxima obtenida en este proyecto se ha alcanzado generando un modelo mediante las técnicas de árbol de decisión, concretamente con un 92.987%. Comparativamente, la mejor precisión obtenida por un modelo generado a partir de una red neural ha sido de un 92.397%. Si al hecho de aportar una mayor decisión añadimos el que la interpretabilidad de dicho árbol de decisión siempre será posible, en contraposición a la opacidad de las técnicas de Deep learning, pareciera que a priori la utilización de redes neuronales añada una capa de complejidad innecesaria sin repercutir en unos mejores resultados.

La diferencia radica en que para conseguir los resultados obtenidos mediante el citado árbol de decisión ha existido un trabajo previo en el feature engineering del dataset. Este ha sido previamente realizado al aglutinar y reducir los diferentes componentes de un datagrama a 41 valores, que es lo que posibilita obtener resultados tan positivos. Y -y esto es la clave- este feature engineering requiere de un conocimiento previo del problema, relativamente profundo. Si suministrásemos al algoritmo de modelización basado en árbol de decisiones los datagramas correspondientes al conjunto de datos de entrenamiento en bruto -como una serie de octetos-, este sería totalmente ineficiente, tanto por la dimensionalidad del vector de entrada como por la imposibilidad de generar distribuciones que representasen conocimiento. Si, por el contrario, suministráramos datagramas en bruto a una red neuronal lo suficientemente compleja y profunda, esta sí que sería capaz de generar un conocimiento a partir de un conjunto de datos en este formato. De forma muchos menos eficiente, utilizando más recursos y posiblemente con menores tasas de éxito, pero la red neuronal, vista como una generalización de las técnicas de ML, no tiene unos requerimientos de feature engineering tan estrictos como estas, y nos permite abordar problemas con un nivel de comprensión inicial inferior al de los métodos de Machine Learning.

Sobre la consideración acerca de Whitebox vs. Blackbox, si bien es cierto que una red neural podrá ser a priori menos interpretable -o directamente no interpretable- en su elaboración del modelo, hay que tener en cuenta dos consideraciones: la primera, que no todas las redes neurales ofrecen una interpretabilidad nula. De hecho, en redes neurales convolucionales comúnmente utilizadas para la resolución de problemas relacionados con la visión por ordenador, cada una de las capas de la topología utilizada puede mostrar características directamente relacionadas con la imagen analizada, como contornos, distribuciones de color o mapas de luminancia, de forma similar a como la señal podría expresarse. La segunda, que si hubiéramos suministrado a la capa de entrada el datagrama en bruto como un vector de octetos, y diseñado convenientemente la red, eventualmente veríamos como diferentes capas representarían combinaciones de octetos correspondientes a protocolos, llamadas a OS, y otros descriptores de alto nivel del paquete analizado.

Hemos discutido y analizado también como las diferentes configuraciones de hiperparámetros a la hora de definir la red neural producen resultados consecuentes con el trasfondo matemático de estas entidades, si bien hemos podido demostrar que incluso aislando los hiperparámetros que pueden beneficiar la obtención de precisiones superiores, estos no pueden manipularse sin tener en cuenta la configuración de forma global. El hecho de que mas neuronas por capa, o un numero de epochs mayor puedan producir resultados beneficiosos para nuestro modelo, no quiere decir que debamos limitarnos a maximizar todo aquel hiperparámetro que se demuestre beneficioso, debiendo seleccionarse siempre la configuración como un conjunto.

7. Siguientes pasos

Si bien todos los objetivos planteados al inicio de este Trabajo de Fin de Master han podido llevarse a cabo, debido a las habituales limitaciones de tiempo relacionadas con el desempeño de una actividad profesional más o menos intensa durante la implementación del proyecto hay varias cuestiones que no han sido desarrolladas como hubiera sido deseable. Pasemos a hacer un breve repaso de cuales son estas.

El limite de ese 93% en las tasas de precisión de los modelos obtenidos con el segundo dataset utilizado ha sido interpretado como la existencia de datos en el conjunto de evaluacion que no estaban presentes en el conjunto de entrenamiento. De hecho, la descripción del NSLKDD ya hace referencia -y así se citó en la presentación del dataset- a que de los 38 tipos de ataque recogidos, en el data set solo se mostraban un subconjunto de ellos. El hecho es que nos hubiera gustado demostrarlo y para ello en la implementación de la matriz de configuraciones introdujimos un tratamiento matemático para poder confirmar si esto era cierto. Al particionar el conjunto de datos según los valores escogidos, procederíamos a realizar una segunda evaluación con el remanente de dicho conjunto de datos de entrenamiento -que naturalmente nunca se hubieran utilizado para entrenar la red-. Al haber un fallo en la implementación, dichos conjuntos wasted_X y Waste_y no estaban correctamente normalizados ni vectorizados, cono lo que no podemos confirmar fehacientemente este punto, aunque naturalmente estamos convencidos de que obtendríamos precisiones cercanas al 100% de proceder así. Queda pendiente.

En segundo lugar, nos hubiera gustado poder realizar una matriz de configuraciones con no 108 sino 1080 datapoints diferentes. Nuestra intención al proceder así hubiera sido generar una red neuronal adicional con los datos de dicha matriz que nos hubiera proporcionado la elección de hiperparámetros optima para el modelo estudiado. Naturalmente el coste de computación de hacia prohibitivo, llegando a cálculos que hubieran durado mas de 30 días, pero la idea nos parece atractiva y no la hemos visto recogida en la bibliografía. Entendemos naturalmente que se estarían malgastando recursos para solucionar un problema concreto, pero si lo reformulásemos orientando la situación al cálculo de anomalías con un clasificador binario -cosa que si hicimos- entendemos que el resultado obtenido seria generalizable al conjunto de problemas de detección de anomalías basados en el análisis de trafico de red representado mediante un esquema basado en el OSI model. Toso esto queda a día de hoy muz lejos de nuestras capacidades e intenciones, por lo que necesitaremos de mas practica y tiempo para siquiera poder plantear el problema de una manera formalmente correcta, si es que tiene sentido en la forma que lo hemos visualizado.

8. Correcciones

Existe un error en la implementación del script tester que afecta a la manera en como se realiza la particion del conjunto de datos de entrenamiento para escoger un subconjunto de este.

Concretamente la línea:

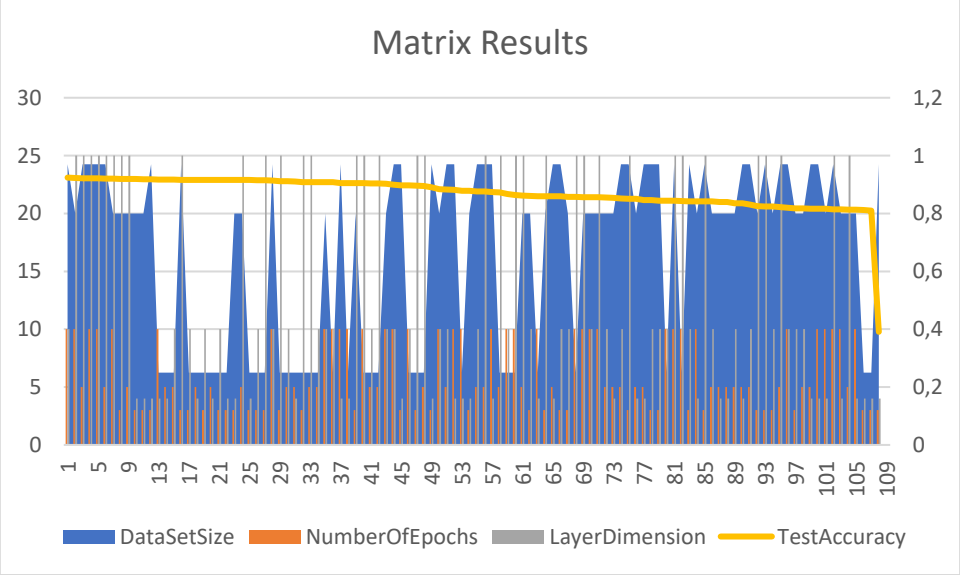
```
X, X_wasted, Y, y_wasted = train_test_split(X, Y, test_size=(whichSize), random_state=42)
```

Debería ser:

```
X, X_wasted, Y, y_wasted = train_test_split(X, Y, test_size=(1-whichSize), random_state=42)
```

Y esto tiene como resultado que la discusión sobre como afecta el conjunto quede parcialmente invalidada. Si bien las conclusiones acerca de que la maximización del numero de muestras no siempre produce los resultados óptimos, podemos observar con claridad con la implementación correcta como éste es un factor decisivo en el éxito de la red neuronal, y que, a mas muestras de entrenamiento, generalmente obtendremos mejores precisiones, lo que esta de acuerdo con toda la bibliografia y cuerpo de conocimiento consultado.

La nueva tabla queda entonces:



Donde puede observarse como si bien las configuraciones con mejores precisiones están copadas por Datasets de tamaño máximo, no parece observarse un sesgo en la distribución de dichos valores de tamaños. ¿Por qué?

Es fácil ver que hay un motivo para ello. Si en el primer planteamiento del estudio escogíamos tres tamaños relativos del Dataset de entrenamiento para iterar sobre ellos [0.03,0.20,0.75] que resultaban, especialmente en el primer caso, en un numero de muestras muy significativamente inferior al del conjunto, de aproximadamente 14.000 sobre 480.000.

El conjunto real de muestras de entrenamiento utilizadas realmente debido al error deviene [0.97,0.80,0.25] lo que en el peor de los casos significa disponer 125.000 muestras de

entrenamiento, no constituyendo un factor decisivo a la hora de observar un sesgo decidido en las precisiones obtenidas.

De este modo, la tabla correcta quedaría:

DataSetSize	1- DataSetSize	NumberOfEpochs	LayerDimension	Normalized	ActivationFuntion	TestLoss	TestAccuracy
0,97	0,03	10	10	TRUE	relu	0,62688725	92,397493%
0,8	0,2	10	25	TRUE	relu	0,611601936	92,232236%
0,97	0,03	5	25	TRUE	relu	0,563611908	92,196548%
0,97	0,03	10	25	FALSE	relu	0,535784517	92,176292%
0,97	0,03	10	25	TRUE	relu	0,667754967	92,142533%
0,97	0,03	5	25	FALSE	relu	0,671066214	92,048973%
0,8	0,2	10	25	FALSE	relu	0,599017033	91,985956%
0,8	0,2	3	25	FALSE	relu	0,569325252	91,974703%
0,8	0,2	5	25	TRUE	relu	0,61265118	91,927441%
0,8	0,2	3	4	FALSE	relu	1,24076571	91,893360%
0,8	0,2	3	4	FALSE	linear	1,184571367	91,787583%
0,97	0,03	3	4	FALSE	linear	1,204960138	91,774079%
0,25	0,75	10	4	FALSE	relu	0,441433233	91,742249%
0,25	0,75	5	4	FALSE	relu	0,363524531	91,698202%
0,25	0,75	5	10	TRUE	relu	0,48480004	91,664443%
0,97	0,03	3	25	TRUE	relu	0,665157382	91,636793%
0,25	0,75	3	10	TRUE	relu	0,363458176	91,590816%
0,25	0,75	5	4	TRUE	linear	0,390667142	91,590173%
0,25	0,75	3	10	FALSE	relu	0,384424945	91,586958%
0,25	0,75	5	4	FALSE	linear	0,384949208	91,582135%
0,25	0,75	3	10	TRUE	linear	0,363081738	91,575062%
0,25	0,75	3	4	TRUE	relu	0,338398605	91,571204%
0,8	0,2	3	4	TRUE	relu	0,401070015	91,567346%
0,8	0,2	5	25	FALSE	relu	0,580264438	91,546126%
0,25	0,75	3	4	TRUE	linear	0,343701839	91,541946%
0,25	0,75	3	10	FALSE	linear	0,35651744	91,538731%
0,25	0,75	3	25	TRUE	linear	0,413626468	91,452566%
0,97	0,03	10	10	FALSE	relu	0,640207614	91,374116%
0,25	0,75	3	25	FALSE	linear	0,435864762	91,162882%
0,25	0,75	5	10	FALSE	linear	0,43206277	91,157738%
0,25	0,75	5	4	TRUE	relu	0,400894832	90,999232%
0,25	0,75	3	25	TRUE	relu	0,441609312	90,871912%
0,25	0,75	5	25	TRUE	relu	0,518516045	90,814040%
0,25	0,75	5	10	FALSE	relu	0,423053234	90,809860%
0,8	0,2	10	10	FALSE	relu	0,619352414	90,793785%
0,25	0,75	10	10	TRUE	relu	0,481322734	90,793141%
0,97	0,03	10	4	FALSE	relu	0,687193897	90,483524%
0,25	0,75	10	4	TRUE	linear	0,463343845	90,474200%
0,8	0,2	3	25	TRUE	relu	0,492100817	90,452980%
0,25	0,75	10	25	TRUE	relu	0,506737888	90,442049%
0,25	0,75	5	10	TRUE	linear	0,44405818	90,343666%
0,25	0,75	5	25	FALSE	relu	0,468862575	90,335306%
0,8	0,2	10	10	TRUE	relu	0,503536655	90,254928%
0,97	0,03	10	10	FALSE	linear	0,560798988	89,951741%
0,97	0,03	3	4	TRUE	linear	0,499951781	89,727324%
0,25	0,75	10	10	TRUE	linear	0,505992129	89,720573%
0,25	0,75	3	25	FALSE	relu	0,424583339	89,634407%
0,25	0,75	5	25	FALSE	linear	0,470669988	89,446000%
0,97	0,03	3	4	TRUE	relu	0,482435311	89,081082%
0,8	0,2	10	10	FALSE	linear	0,574359198	88,408476%
0,97	0,03	5	10	FALSE	relu	0,526261398	88,269904%

0,97	0,03	10	4	TRUE	linear	0,571245321	88,161233%
0,25	0,75	10	4	FALSE	linear	0,49201244	87,843899%
0,8	0,2	3	4	TRUE	linear	0,516211979	87,825573%
0,97	0,03	5	10	TRUE	relu	0,580239721	87,602121%
0,97	0,03	5	25	FALSE	linear	0,585334225	87,579615%
0,97	0,03	10	4	TRUE	relu	0,579998879	87,360343%
0,25	0,75	5	25	TRUE	linear	0,500830208	87,272248%
0,25	0,75	10	10	FALSE	linear	0,539624748	86,698346%
0,25	0,75	10	25	FALSE	relu	0,672321746	86,343717%
0,8	0,2	5	25	TRUE	linear	0,57475775	86,159168%
0,8	0,2	3	10	TRUE	relu	0,502903072	86,014487%
0,25	0,75	10	4	TRUE	relu	0,370001283	85,961759%
0,8	0,2	3	25	FALSE	linear	0,568287252	85,957258%
0,97	0,03	5	4	FALSE	linear	0,569063894	85,956293%
0,97	0,03	3	10	TRUE	linear	0,553519487	85,938932%
0,8	0,2	3	10	TRUE	linear	0,550204648	85,761778%
0,25	0,75	10	25	TRUE	linear	0,545631804	85,757920%
0,8	0,2	10	25	FALSE	linear	0,611669127	85,648284%
0,8	0,2	10	10	TRUE	linear	0,585693054	85,613560%
0,8	0,2	10	25	TRUE	linear	0,621139847	85,611953%
0,8	0,2	5	10	TRUE	relu	0,605193624	85,517106%
0,8	0,2	5	4	FALSE	linear	0,557495056	85,397825%
0,97	0,03	5	10	TRUE	linear	0,593452799	85,184983%
0,97	0,03	3	25	FALSE	linear	0,572227029	85,066666%
0,8	0,2	5	4	TRUE	relu	0,596316142	85,042552%
0,97	0,03	5	4	TRUE	relu	0,492217786	84,663488%
0,97	0,03	3	10	FALSE	linear	0,569621101	84,658022%
0,97	0,03	3	10	TRUE	relu	0,539107023	84,459005%
0,25	0,75	10	10	FALSE	relu	0,520992114	84,443573%
0,97	0,03	10	25	TRUE	linear	0,651476169	84,395346%
0,25	0,75	10	25	FALSE	linear	0,577141606	84,283459%
0,97	0,03	3	10	FALSE	relu	0,713503163	84,189256%
0,8	0,2	10	4	TRUE	linear	0,609550756	84,145851%
0,97	0,03	3	25	TRUE	linear	0,584153885	84,141029%
0,8	0,2	5	10	FALSE	linear	0,600115043	84,135563%
0,8	0,2	5	4	FALSE	relu	0,556538091	83,964196%
0,8	0,2	5	4	TRUE	linear	0,586289318	83,943619%
0,8	0,2	5	10	FALSE	relu	0,602684595	83,526938%
0,97	0,03	5	4	TRUE	linear	0,603185291	83,464886%
0,97	0,03	5	10	FALSE	linear	0,594119275	83,069424%
0,8	0,2	3	25	TRUE	linear	0,606272791	82,473338%
0,97	0,03	3	25	FALSE	relu	0,655762183	82,419967%
0,8	0,2	3	10	FALSE	linear	0,588572068	82,383636%
0,97	0,03	5	25	TRUE	linear	0,629424188	82,213234%
0,97	0,03	10	10	TRUE	linear	0,654386956	81,970813%
0,8	0,2	3	10	FALSE	relu	0,604195542	81,770510%
0,8	0,2	5	10	TRUE	linear	0,642430197	81,763115%
0,97	0,03	5	4	FALSE	relu	0,644792302	81,675985%
0,97	0,03	10	4	FALSE	linear	0,641439464	81,666983%
0,8	0,2	10	4	FALSE	linear	0,648103851	81,632902%
0,97	0,03	10	25	FALSE	linear	0,737902882	81,417488%
0,8	0,2	10	4	TRUE	relu	0,548351409	81,407200%
0,8	0,2	5	25	FALSE	linear	0,662009413	81,355436%
0,8	0,2	10	4	FALSE	relu	0,595068922	81,292420%
0,25	0,75	3	4	FALSE	relu	0,906976845	81,200788%
0,25	0,75	3	4	FALSE	linear	3,01506848	81,012703%
0,97	0,03	3	4	FALSE	relu	9,722336817	39,089603%

La implementación definitiva del script quedaría como:

```
from __future__ import print_function
from sklearn.model_selection import train_test_split
import pandas as pd
import numpy as np
np.random.seed(2041)  # for reproducibility
from keras.preprocessing import sequence
from keras.utils import np_utils
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Embedding
from keras.layers import LSTM, SimpleRNN, GRU
from keras.datasets import imdb
from keras.utils.np_utils import to_categorical
from sklearn.metrics import (precision_score, recall_score,f1_score,
accuracy_score,mean_squared_error,mean_absolute_error)
from sklearn import metrics
from sklearn.preprocessing import Normalizer
import h5py
import keras
from keras import callbacks
from keras.callbacks import ModelCheckpoint, EarlyStopping,
ReduceLROnPlateau, CSVLogger
import datetime

currentDT = datetime.datetime.now()
print (str(currentDT))

filename="TestResults/TFM_Tester_"+currentDT.strftime("%Y%m%d")+"_"+current
DT.strftime("%H%M%S")
filenameTxt="TestResults/TFM_Tester_"+currentDT.strftime("%Y%m%d")+"_"+curr
entDT.strftime("%H%M%S")+".txt"
filenameCsv="TestResults/TFM_Tester_"+currentDT.strftime("%Y%m%d")+"_"+curr
entDT.strftime("%H%M%S")+".csv"

matrixFile=open(filenameTxt,"x")
matrixFile.write("WELCOME to Tester ----------------------------------------
--------------------\n")
matrixFile.write(str(filenameTxt))
matrixFile.write(str(currentDT))
matrixFile.write("\nWELCOME to Tester ----------------------------------
----------------------\n")

matrixCsvFile=open(filenameCsv,"x")
matrixCsvFile.write("DataSetSize, NumberOfEpochs, LayerDimension,
Normalized, ActivationFuntion, TestLoss, TestAccuracy, WastedLoss,
WastedAccuracy")

traindata = pd.read_csv('kdd/binary/Training.csv', header=None)
testdata = pd.read_csv('kdd/binary/Testing.csv', header=None)


X = traindata.iloc[:,1:42]
Y = traindata.iloc[:,0]
C = testdata.iloc[:,0]
T = testdata.iloc[:,1:42]

# Reserve 10,000 samples for validation
x_val = X[-10000:]
y_val = Y[-10000:]
X= X[:-10000]
```

```
Y= Y[:-10000]



whichSize = float(input("Chose the size of the Dataset [0.01-0.95] : "))
print("\nDataset Size : "+str(whichSize))
matrixFile.write("\nDataset Size : "+str(whichSize))

X, X_wasted, Y, y_wasted = train_test_split(X, Y, test_size=(1-whichSize),
random_state=42)

trainX = np.array(X)
wasteX = np.array(X_wasted)
testT = np.array(T)

numberOfEpochs = int(input("Choose number of Epochs [1-99] : "))
print("\nNumber of Epochs? : " + str(numberOfEpochs))
matrixFile.write("\nNumber of Epochs? : " + str(numberOfEpochs))

layerDimension = int(input("Choose hidden layer's Dimension [1-100] : "))
print("\nHidden Layer Dimension? : " + str(layerDimension))
matrixFile.write("\nHidden Layer Dimension? : " + str(layerDimension))

numberOfHiddenLayers = int(input("Choose hidden layer's number [1-10] : "))
print("\nHidden Layer Number? : " + str(layerDimension))
matrixFile.write("\nHidden Layer Number? : " + str(layerDimension))

normalized = bool(int(input("Choose if DataSet will be normalized [0/1]")))
print("\nNormalizado? : " + str(normalized == True))
matrixFile.write("\nNormalizad? : " + str(normalized == True))

activationFunction = input("Choose the activation funtion [linear, relu,
tanh, sigmoid...] : ")
print("\nActivation Function? : " + str(activationFunction))
matrixFile.write("\nActivation Function? : " + str(activationFunction))

dropoutValue = float(input("Choose dropout Value [0.001 - 0.9] : "))
print("\nDropout Value? : " + str(layerDimension))
matrixFile.write("\nDropout Value? : " + str(layerDimension))

trainX.astype(float)
testT.astype(float)
wasteX.astype(float)

if normalized:
    scaler = Normalizer().fit(trainX)
    trainX = scaler.transform(trainX)

    scaler = Normalizer().fit(wasteX)
    wasteX = scaler.transform(wasteX)

    scaler = Normalizer().fit(testT)
    testT = scaler.transform(testT)

y_train = np.array(Y)
y_test = np.array(C)
y_waster = np.array(y_wasted)

X_train = np.array(trainX)
X_test = np.array(testT)
X_waster = np.array(wasteX)
```

```
keras.backend.clear_session()

batch_size = 64

# 1. define the network

model = Sequential()
model.add(Dense(layerDimension, input_dim=41,
activation=activationFunction))
print ("Adding First Hidden Layer\n")
if numberOfHiddenLayers > 1:
    for addLayer in range(2,numberOfHiddenLayers+1):
        model.add(Dense(layerDimension, activation=activationFunction))
        model.add(Dropout(dropoutValue))
        print("Adding hidden layer number : "+str(addLayer)+"\n")

model.add(Dense(1))
model.add(Activation('sigmoid'))



model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])
checkpointer = callbacks.ModelCheckpoint(filepath="Data/AVP_checkpoint-
{epoch:02d}.hdf5", verbose=1,

save_best_only=True, monitor='loss')
csv_logger = CSVLogger(
    filename
    + "_NR" + str(normalized == True)
    + "_LSZ" + str(layerDimension)
    + "_NE" + str(numberOfEpochs)
    + "_AF" + str(activationFunction)
    + '_Training_set.csv', separator=',', append=False)

model.fit(X_train, y_train,
    batch_size=batch_size,
    nb_epoch=numberOfEpochs,
    validation_data=(x_val, y_val),
    callbacks=[checkpointer, csv_logger])

# Evaluate the model on the test data using `evaluate`
print("\n_____________________________________________________________________
_____________________")
print('\n# Evaluate on test data')
results = model.evaluate(X_test, y_test, verbose=0, batch_size=128)
print('\ntest loss, test acc:', results)
print("\n_____________________________________________________________________
_____________________________\n")



matrixFile.write("\n_____________________________________________________________
_____________________________________")
matrixFile.write('\n# Evaluate on test data')
matrixFile.write('\ntest loss, test acc:')
matrixFile.write(str(results))
matrixFile.write("\n_____________________________________________________________
_____________________________________\n")
```

```
# Evaluate the model on the wasted data using `evaluate`
print("\n_____________________________________________________________________
_____________________")
print('\n# Evaluate on wasted data')
results = model.evaluate(X_waster, y_waster, verbose=0, batch_size=128)
print('\ntest loss, test acc:', results)
print("\n_____________________________________________________________________
_____________________\n")



matrixFile.write("\n_____________________________________________________________
_____________________________")
matrixFile.write('\n# Evaluate on wasted data')
matrixFile.write('\ntest loss, test acc:')
matrixFile.write(str(results))
matrixFile.write("\n_____________________________________________________________
_____________________________\n")


matrixCsvFile.write("\n" + str(whichSize) + ", ")
matrixCsvFile.write(str(numberOfEpochs) + ", ")
matrixCsvFile.write(str(layerDimension) + ", ")
matrixCsvFile.write(str(normalized == True) + ", ")
matrixCsvFile.write(str(activationFunction) + ", ")
matrixCsvFile.write(str(results[0]) + ", ")
matrixCsvFile.write(str(results[1]))



model.save(filename + str(normalized == True) + "_" + str(layerDimension) +
"_" + str(numberOfEpochs) + "model.hdf5")

matrixCsvFile.close()

matrixFile.write("\nFinished Matrix ------------------------------------
--------------------\n")
matrixFile.write("\nElapsed Time\n")
finishedDT = datetime.datetime.now()
matrixFile.write(str(finishedDT-currentDT))
matrixFile.write("\nFinished Matrix ------------------------------------
--------------------\n")

matrixFile.close()
```

En la que ya esta incluida la funcionalidad comentada en el apartado anterior “Siguientes pasos” para utilizar el conjunto sobrante de datos no utilizados para el entrenamiento del modelo como set de evaluación secundario y confirmar así que una vez no se encuentran presentes las tipologías de ataque adicionales en el conjunto de test, las precisiones alcanzan al 99% sin demasiado problema para las configuraciones más exitosas.

Apencice A. Documentación

A.1 Análisis de los recursos online disponibles

https://towardsdatascience.com/machine-learning-for-cybersecurity-101-7822b802790b

Sin duda alguna, se trata del recurso online más importante que hemos encontrado para abordar el desarrollo de este proyecto. En una lectura ligera de apenas 25 minutos, encontramos no solo un núcleo de información básica muy bien presentada y explicada, sino además una serie de vínculos a recursos adicionales para profundizar en cada uno de los conceptos tratados.

Podríamos añadir otros recursos como:

https://www.cylance.com/en-us/index.html

https://www.exabeam.com/

correspondientes a dos vendors fundamentales en la aplicación de Machine Learning para la seguridad en ámbitos empresariales, específicamente en end point el primero y en UEBA el segundo. Debido a mi reciente cambio de trabajo en una consultoría, me estoy certificando en ambos, y la documentación disponible online resulta muy completa e interesante.

Para una introducción más académica a Data Science e independiente del componente de aplicación a la seguridad:

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0002-introduction-to-computational-thinking-and-data-science-fall-2016/

A.2 Análisis de la Bibliografía utilizada.

-Machine Learning and Security. Protecting Systems with Data and Algorithms. Clarence Chio and David Freeman. O'Reilly 2018.

Clarence Chio y David Freeman son especialistas en aprendizaje automático aplicado a la seguridad, y en este volumen brindan un marco para analizar la intersección de estos dos campos, así como un conjunto de herramientas de algoritmos de aprendizaje automático que puede aplicar a una variedad de problemas de seguridad. Este libro nos ha parecido ideal para iniciarnos en el campo, tratando los temas con un nivel de profundidad creciente y sin ignorar las bases matemáticas tras las APIs utilizadas. Si bien es cierto que presupone un nivel de conocimientos básicos -Desarrollo en Python, Algebra Matricial-, dicha información suele poder adquirirse en un tiempo relativamente corto en multitud de recursos online adicionales. Su mayor defecto es que no trata la utilización de redes neuronales, siendo de utilidad limitada para el actual proyecto.

-Introduction to Artificial Intelligence by Security Professionals. Cylance Data Team. Cylance 2017.

Se trata de un libro editado por el desarrollador de CylanceProtect, un antivirus de última generación con un modelo de protección basado en la detección de anomalías mediante técnicas de Machine Learning y Deep Learning. Su porcentaje de efectividad no solo está por encima de productos de otros fabricantes con un modelo de análisis y detección basado en firmas, sino que consigue, de forma esperable, detectar amenazas no registradas hasta el momento.

El libro en sí tiene tres ventajas: es breve -apenas 177 páginas-, es gratuito, y es sorprendentemente ameno e interesante. Dejando de lado que naturalmente se trata de un libro editado por una compañía que vende un producto, el libro recoge varias técnicas y casos de uso y consta de un repositorio git con el código y conjuntos de datos necesario para seguir su lectura.

- Deep Learning with Python. Francois Chollet 2017

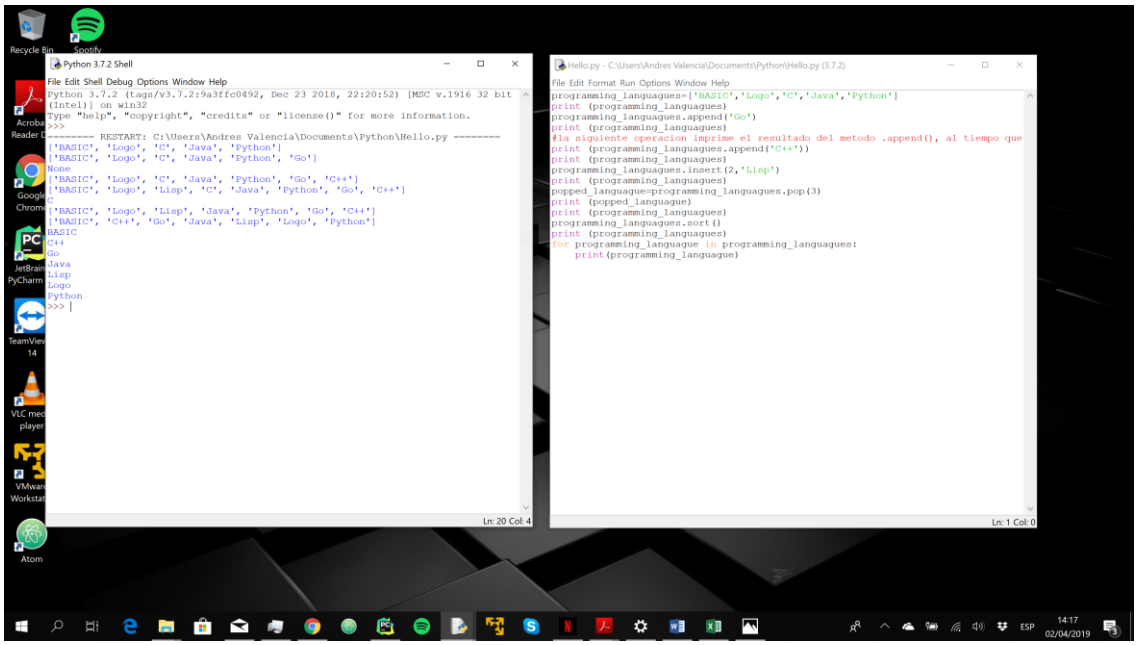
Escrito por el diseñador de Keras, el libro no solo es actual y comprehensivo, sino que está bien escrito y aúna un enfoque practico sin descuidar las base matemática. Aun sin tener ninguna relación con el problema tratado, su explicación de las diferentes clases de red neuronal e implementaciones posibilita acometer cualquier proyecto de inicio a fin con cierta confianza.

A.3 Selección de las herramientas necesarias

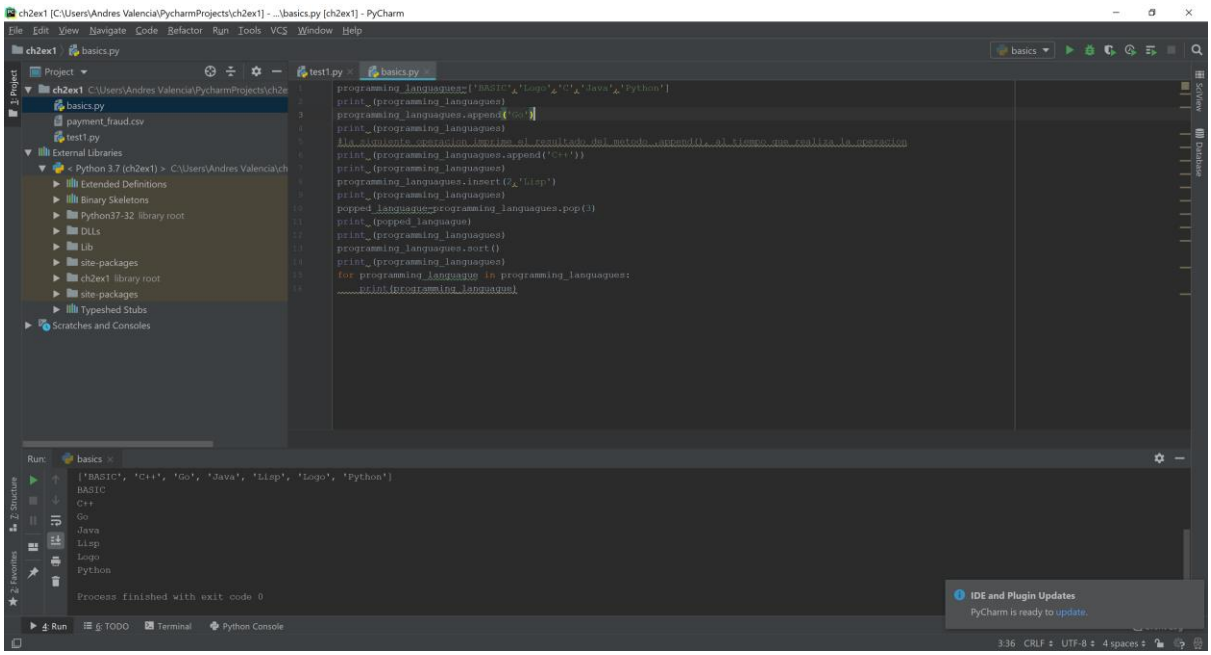
A.3.1 IDE y entorno de programación

IDE PyCharm Professional 2018.3 Education License

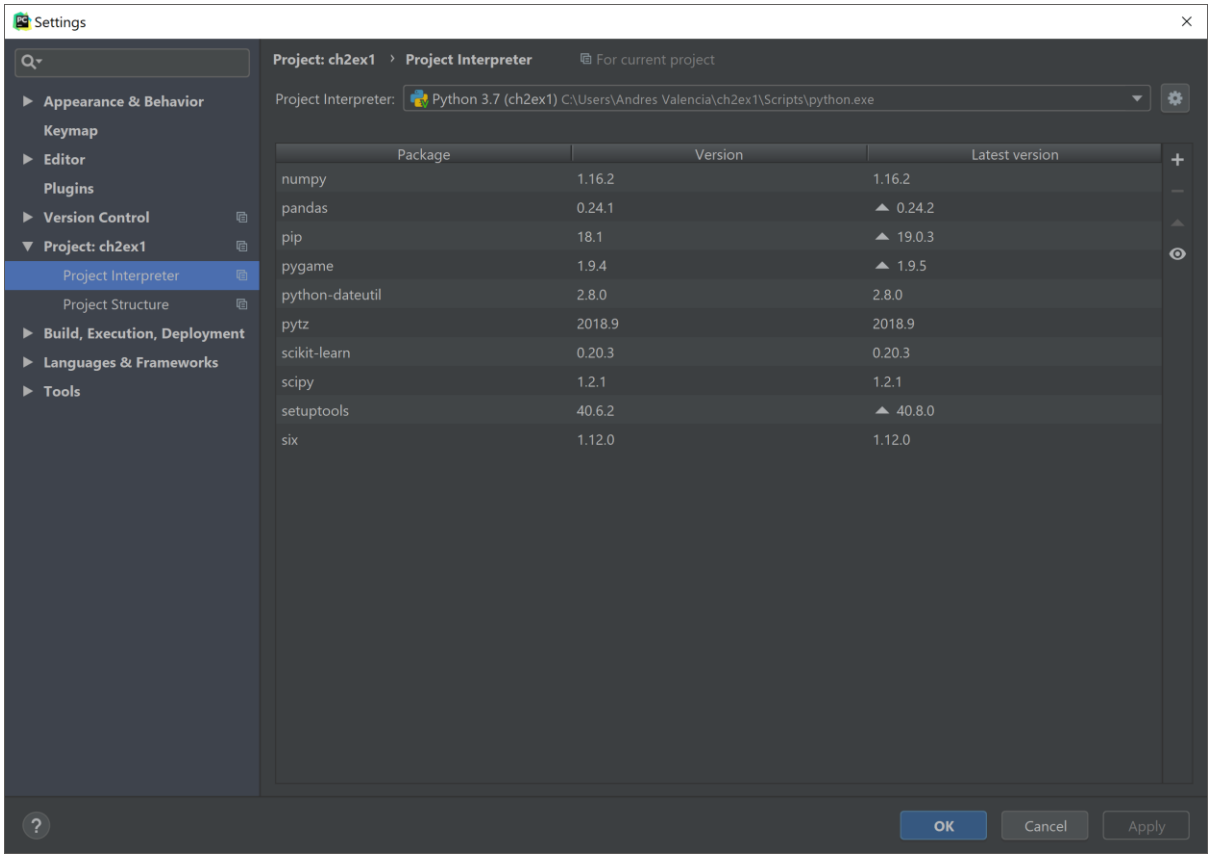
Tras testear instalaciones de Python más básicas, nos hemos decidido por elegir JetBrains Pycharm. Las razones de esta decisión son la facilidad de instalación de paquetes, las funciones de autocorrección integradas y la facilidad de navegación por los proyectos. Aunque la curva de aprendizaje es algo mas pronunciada que la de por ejemplo el Python Shell:



tras un estudio breve de las funcionalidades de la IDE, esta deviene en mucho mas usable y potente tan pronto como es necesario abordar proyectos de más entidad. Asimismo, la facilidad a la hora de buscar e instalar librerías, de forma automatizada con sus posibles dependencias, deviene en una ventaja fundamental para el neófito:



Encontrándose todos los recursos requeridos de manera sumamente fácil y siendo la adquisición e instalación de los paquetes automatizada.



A.4 Aprendizaje de las herramientas necesarias

A.4.1 Aprendizaje del Lenguaje de Programación: Python 3.7

¿Por qué Python para Machine Learning?

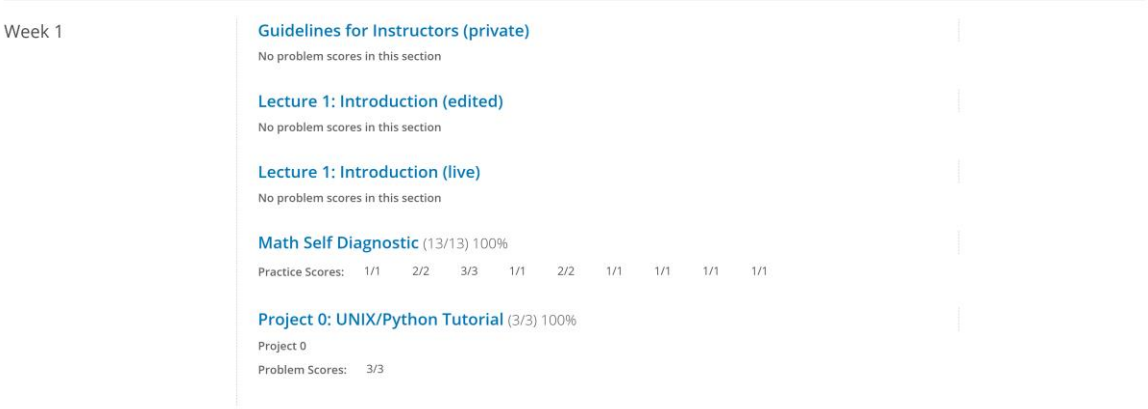
Punto a desarrollar, consultando fuentes como:

https://hackernoon.com/why-is-python-used-for-machine-learning-773aaaadd0ea

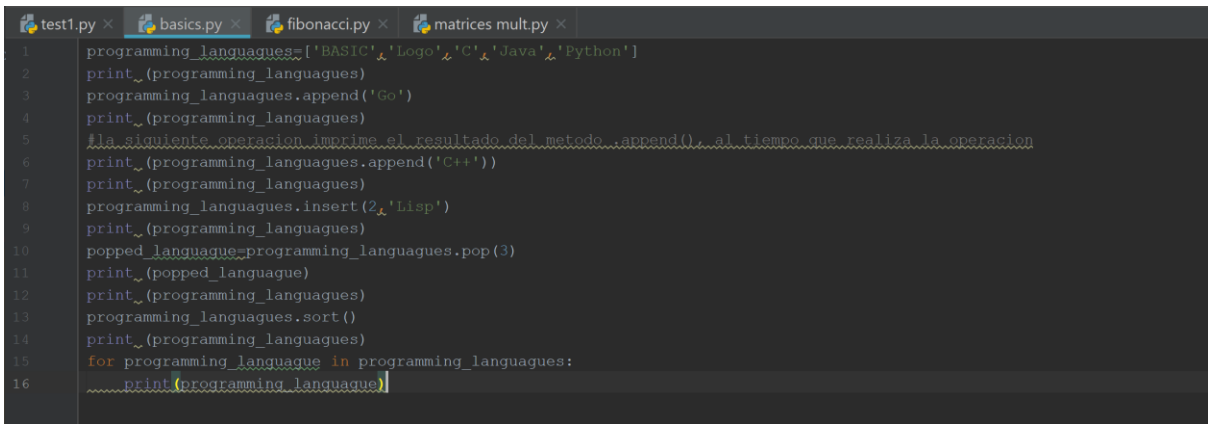
Introducción a Python: Tutorial Básico.

BerkeleyX: CS188x_1**Artificial Intelligence**

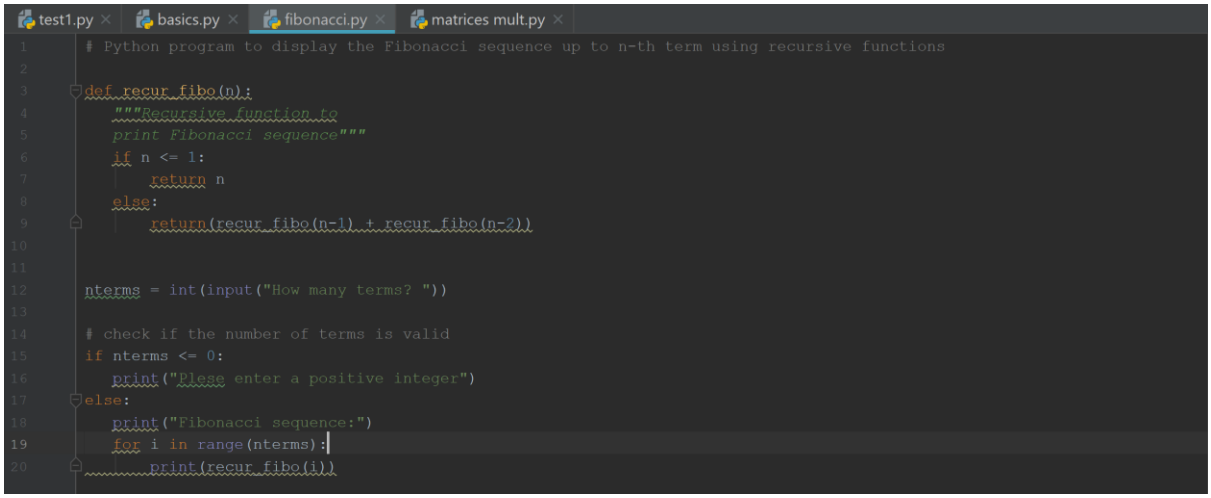
En la plataforma edX existen innumerables recursos para poder familiarizarse con el lenguaje de programación Python. Para el desarrollo de nuestro proyecto, hemos escogido un breve tutorial de introducción al lenguaje que nos introduce de manera sencilla a las particularidades mas importantes de dicho lenguaje de programación. Lo interesante de dicho curso es que al estar orientado a la inteligencia artificial, proporciona desde el inicio una aplicación del uso del lenguaje orientada a esta disciplina.



Tras esto, hemos empezado a desarrollar pequeños snippets de código con funcionalidades limitadas de manera que pudiéramos familiarizarnos con las diferentes posibilidades del lenguaje, como las colecciones:



Funciones y recursividad:



O multiplicación de matrices:

Entendemos en todo momento que el objetivo de este proyecto no es el aprendizaje del lenguaje de programación Python y que el conocimiento que debemos desarrollar sobre este no ha de ser en modo alguno exhaustivo.

A.4.2 Aprendizaje de la API

Siguiendo las pautas marcadas por la referencia Bibliográfica "*Machine Learning and Security. Protecting Systems with Data and Algorithms."* antes citada, usaremos un dataset de ejemplo para empezar a familiarizarnos con las técnicas básicas que deberemos utilizar para la elaboración del proyecto.

Concretamente, utilizaremos un conjunto de datos correspondientes a registros de transacciones online recogido por una tienda online. El conjunto de datos contiene 39,221 transacciones, cada una detallando 5 propiedades -a partir de ahora *features*- que pueden ser utilizadas para describir la transacción. Asimismo, como sexto valor encontramos una etiqueta -*label*- binaria en cada transacción, indicando con un valor de "1" si es fraudulenta y de "0" si no lo es, de modo que el nuestro es un *labeled dataset*. Nuestro objetivo será, utilizando dicho dataset, el de entrenar un modelo de machine learning sencillo que nos permita preveer el valor de dicha etiqueta en transacciones anteriormente nunca encontradas por el modelo.

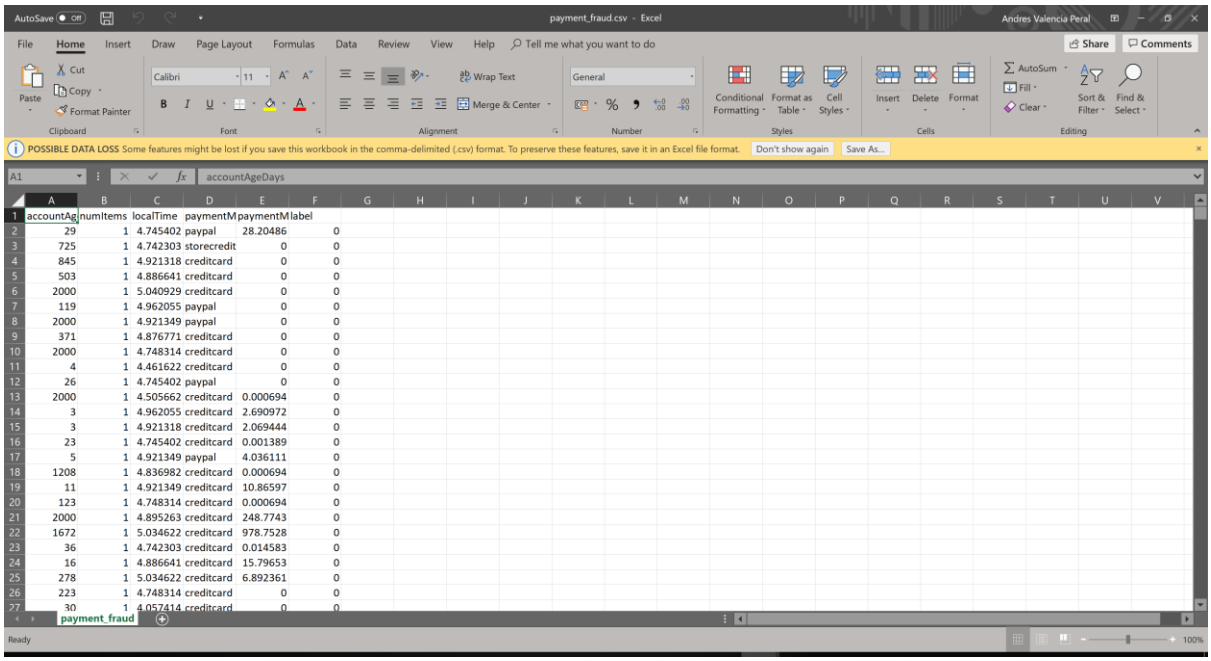
Es obligado detenerse para incidir en que dado el hecho de que tenemos un conocimiento previo acerca de si cada transacción registrada es fraudulenta o no, por lo que podemos aprovechar dicha información para realizar un aprendizaje

automático *supervisado*, porque tenemos una información externa a los datos que los clasifica. De no ser así, un podríamos utilizar dicho dataset para mediante técnicas de clustering intentar obtener información sobre la que clasificar cada transacción.

La URL del repositorio del dataset es:

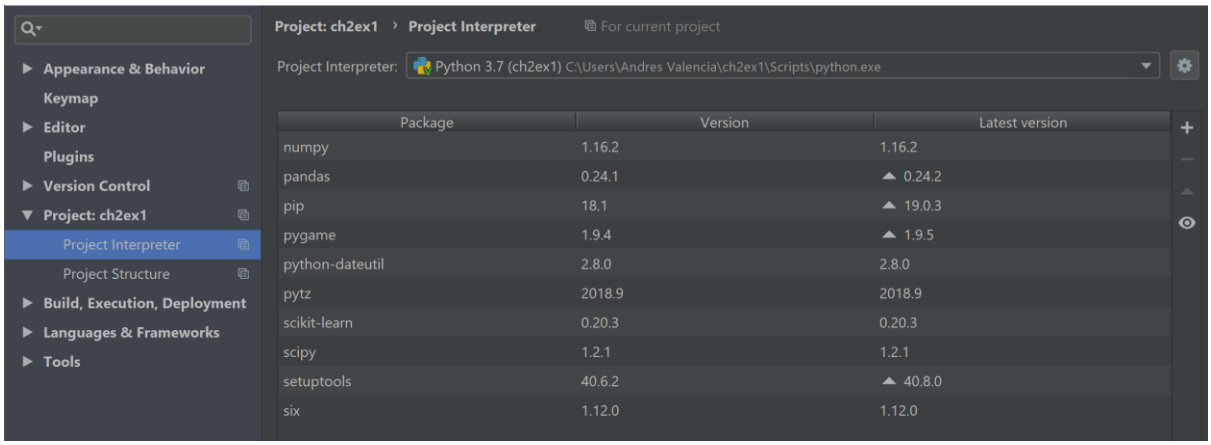
https://github.com/oreilly-mlsec/book-resources/blob/master/chapter2/datasets/payment_fraud.csv

y procedemos a descargarla:



accountAg	numItems	localTime	paymentM	paymentM	label
29	1	4.745402	paypal	28.20486	0
725	1	4.742303	storecredit	0	0
845	1	4.921318	creditcard	0	0
503	1	4.886641	creditcard	0	0
2000	1	5.040929	creditcard	0	0
119	1	4.962055	paypal	0	0
2000	1	4.921349	paypal	0	0
371	1	4.876771	creditcard	0	0
2000	1	4.748314	creditcard	0	0
4	1	4.461622	creditcard	0	0
26	1	4.745402	paypal	0	0
2000	1	4.505662	creditcard	0.000694	0
3	1	4.962055	creditcard	2.690972	0
3	1	4.921318	creditcard	2.069444	0
23	1	4.745402	creditcard	0.001389	0
5	1	4.921349	paypal	4.036111	0
1208	1	4.836982	creditcard	0.000694	0
11	1	4.921349	creditcard	10.86597	0
123	1	4.748314	creditcard	0.000694	0
2000	1	4.895263	creditcard	248.7743	0
1672	1	5.034622	creditcard	978.7528	0
36	1	4.742303	creditcard	0.014583	0
16	1	4.886641	creditcard	15.79653	0
278	1	5.034622	creditcard	6.892361	0
223	1	4.748314	creditcard	0	0
30	1	4.057414	creditcard	0	0

Una vez hemos descargado e instalado mediante los scripts pip de PyCharm todos los paquetes necesarios para empezar a trabajar, comprobamos que leemos correctamente el conjunto de datos.



Package	Version	Latest version
numpy	1.16.2	1.16.2
pandas	0.24.1	▲ 0.24.2
pip	18.1	▲ 19.0.3
pygame	1.9.4	▲ 1.9.5
python-dateutil	2.8.0	2.8.0
pytz	2018.9	2018.9
scikit-learn	0.20.3	0.20.3
scipy	1.2.1	1.2.1
setuptools	40.6.2	▲ 40.8.0
six	1.12.0	1.12.0

Una vez hecho esto y antes de estudiar el caso, comprobemos que podemos acceder al dataset, utilizando:

```
# import Pandas lybrary for data analysis

import pandas as pd

# create variable df (for dataframe) a Pandas data structure that represents a dataset in a two dimensional table

df=pd.read_csv('payment_fraud.csv')

# Print a ramdom sample with n rows

print(df.sample(3))
```

Cuyo resultado en la consola es:

```
"C:\Users\Andres Valencia\ch2ex1\Scripts\python.exe" "C:/Users/Andres Valencia/PycharmProjects/ch2ex1/test1.py"
       accountAgeDays  numItems  ...  paymentMethodAgeDays label
19354             510         1  ...            495.224306     0
19239            1979         1  ...           1236.181250     0
22251             317         1  ...             16.959722     0

[3 rows x 6 columns]
```

Una vez comprobada la funcionalidad el entorno, procedamos a discutir el problema con cierto detalle. En primer lugar, analicemos las features del data set:

accountAgeDays

numItems

localTime

paymentMethod

paymentMethodAgeDays

label

que corresponden, respectivamente, a la antigüedad de la cuenta, el numero de objetos comprados, la hora local de la transacción, el modo de pago, y la antigüedad de el modo de pago para dicha cuenta. “label”, como indicamos con anterioridad, determina si la transacción es fraudulenta o no, basada en la existencia de un proceso de *chargeback* realizado por la entidad emisora.

Uno de estas features destaca debido a que su valor no es numérico. Se trata de *paymentMethod*, que puede tomar solamente tres valores: *creditcard*, *paypal* y *storecredit*. Este tipo de feature se denomina como variable categórica, porque adota el valor correspondiente a la categoría a la que pertenece. Importa esto?

Algunos algoritmos de machine learning requieren que los valores del dataset suministrado sean todos numéricos para poder ser evaluados. Este es el caso del algoritmo de logistic regression, que utilizaremos para esta prueba de concepto. Nótese que esto no es necesario para todos los algoritmos existentes. Por ejemplo, en el algoritmo de arboles de decisión, los valores categóricos son aceptados. En el caso que nos ocupa, convertiremos dichos valores categóricos en numéricos mediante:

```
# Convert cathegorical values to numeric

df = pd.get_dummies(df, columns=['paymentMethod'])

# print all columns of the sample

with pd.option_context('display.max_rows', None, 'display.max_columns', None):
    print(df.sample(3))
```

Obteniendo:

```
[3 rows x 6 columns]
       accountAgeDays  numItems  localTime  paymentMethodAgeDays  label  \
33769             269         2   4.461622             73.830556      0
12029             935         1   4.965339            316.400000      0
23975              68         1   4.962055              0.000000      0

       paymentMethod_creditcard  paymentMethod_paypal  \
33769                         0                     1
12029                         0                     1
23975                         1                     0

       paymentMethod_storecredit
33769                          0
12029                          0
23975                          0
```

Con lo que hemos añadido tres columnas nuevas a la tabla: *paymentMethod_credicard*, *paymentMethod_paypal* y *paymentMethod_storecredit*. Estas tres columnas configuran necesariamente vectores ortogonales, esto es, una y solo una de ellas adoptara un valor de 1, siendo las otras dos de 0. Este método de codificación de variables categóricas se conoce como *one-hot encoding*, y las nuevas variables se definen como *dummy variables* en nuestra terminología.

Hasta ahora nos hemos limitado a editar y preparar el dataset. Pese a la extrema sencillez del ejemplo aquí descrito, la preparación de un dataset no es solamente un paso fundamental en la elaboración del modelo, sino que puede ser en muchos casos la parte fundamental y principal de dicha elaboración. Las técnicas avanzadas, como Principal Component Analysis, nos permitirán obtener un subconjunto limitado de features de un dataset para poder suministrar al modelo para su aprendizaje sin perder prácticamente su poder predictivo. En palabras de Derek Lin, Chief Data Scientist de Exabeam, la preparación de los datos ocupa en su proceso entre un 70% y un 80% del trabajo que realizan en dicho entorno.

Una vez hemos aclarado este punto, procederemos nuestro conjunto de datos en dos: training set and test set. Mediante este *Split*, definimos que parte del dataset principal se utilizara para para entrenar al modelo y que parte se utilizara par comprobar la validez de dicho modelo. Mediante:

```
33  # Split the dataset into training and test sets
34
35  from sklearn.model_selection import train_test_split
36
37  X_train, X_test, y_train, y_test = train_test_split(
38  df.drop('label', axis=1), df['label'],
39  test_size=0.33, random_state=17)
```

Mediante estas instrucciones, la división realizada toma como valor de referencia test_size=0.33 para definir un training set (X_train) de 0.67 y un test set (X_test) de 0.33. Esto significa que utilizaremos dos tercios del conjunto de datos para entrenar al algoritmo, y un tercio para comprobar su eficacia.

Aplicaremos ahora un algoritmo estándar de aprendizaje supervisado, *logistic regression*, a los datos:

```
41  # Apply Logistic Regression
42
43  from sklearn.linear_model import LogisticRegression
44  used_solver='liblinear'
45  clf = LogisticRegression(solver=used_solver)
46  clf.fit(X_train, y_train)
```

Como particularidad, al invocar el constructor en la línea 45, definimos el tipo de solver que utilizaremos para entrenar al modelo. Este valor, sin entrar en profundidad, nos permite definir con exactitud las operaciones matemáticas que realizaremos para entrenar a nuestro modelo. Algunas exigirán mas potencia de calculo y serán idóneas para diferentes clases de datasets, ofreciendo diferentes de eficiencia, por lo que es un valor a tener en cuenta.

Una vez hecho esto, ya tenemos a nuestro modelo entrenado. Solo queda suministrarle el conjunto de datos (X_test) para obtener las predicciones y_pred:

```
48  # Obtain the prediction values (y_pred labels) for test dataset (X_test)
49
50  y_pred = clf.predict(X_test)
```

Para por último poder realizar un análisis acerca de la precisión del modelo mediante:

```
# Test Accuracy

from sklearn.metrics import accuracy_score
print("Accuracy Score with solver: "+used_solver)
print(accuracy_score(y_pred, y_test))
```

Imprimiendo finalmente el resultado en la consola:

```
Accuracy Score with solver: liblinear
0.999922738159623
```

Mediante esta necesariamente sencilla prueba de concepto, hemos establecido el esquema más básico de la que será la aproximación a realizar para poder solucionar el problema que nos ocupa:

a) Ingestión de datos
b) Tratamiento de dichos datos
c) División entre datos de entrenamiento y datos de prueba
d) Elección del modelo de aprendizaje automático y particularidades
e) Entrenamiento del modelo
f) Prueba del modelo

Este esquema es modular, significando que podemos sustituir la funcionalidad de cualquiera de sus partes por una equivalente. Por ejemplo, si en lugar de un archivo .csv tuviéramos que recoger los datos de una base de datos o sensores en tiempo real, solamente es necesario cambiar dicha parte del programa. Si además de eso, deseamos utilizar un modelo de arboles de decisión, esa es la parte del codigo a modificar, sin necesidad de alterar las demás partes, aunque es necesario comprender que la edición de los datos y la elección del modelo de aprendizaje no son decisiones independientes, sino que van íntimamente unidas.

Por ejemplo, utilicemos ahora la técnica de Decission Trees, para lo cual modificaremos únicamente las líneas de codigo 36 a 39, correspondientes a la sección d)

```
# Apply Decission Tree

used_criterion = 'gini'
clf = DecisionTreeClassifier(used_criterion, random_state = 100, max_depth=3, min_samples_leaf=5)
clf.fit(X_train, y_train)

# Obtain the prediction values (y_pred labels) for test dataset (X_test)

y_pred = clf.predict(X_test)

# Test Accuracy

from sklearn.metrics import accuracy_score
print("Accuracy Score with criterion: "+used_criterion)
print(accuracy_score(y_pred, y_test))
```

Obteniendo:

```
Accuracy Score with criterion: gini
1.0

Process finished with exit code 0
```