Análisis de Datos para Ciberseguridad: Trabajo práctico 2

Alonso Araya Calvo
Pedro Soto
Sofia Oviedo
Instituto Tecnológico de Costa Rica,
Escuela de Ingeniería en Computación,
Programa de Maestría en Ciberseguridad

14 de setiembre de 2025

Escribir la introduccion aqui al final de escribir todo el trabajo...

1. Implementacion de un arbol de decision y random forests para clasificar todos los tipos de ataques

En esta sección se desarrolló la implementación de un árbol de decisión y un random forest para clasificar todos los tipos de ataques encontrados en el dataset KDD99.

Para este efecto se utilizó la libreria Scikit Learn que contiene la funcionalidad necesaria para entrenar estos modelos sin tener que implementarlos desde cero. Asimismo, se utilizo la libreria Optuna que permite optimizar los hiperparámetros de los modelos facilmente.

En las siguientes subsecciones se detallan los resultados obtenidos y los pasos realizados para generar estos modelos.

1.1. Generación de las particiones del dataset

Para poder particionar el set de datos correctamente se desarrollo una función llamada 'split_dataset' que se encarga de partir el dataset en tres conjuntos: entrenamiento, validación y prueba.

Para ello se utiliza la función 'train_test_split' de Scikit Learn que permite particionar el dataset en dos conjuntos: entrenamiento y prueba. Despues se utiliza la función 'train_test_split' de Scikit Learn nuevamente para particionar el conjunto de entrenamiento en dos conjuntos: entrenamiento y validación, por medio del calculo del tamaño de la validación respecto al conjunto restante.

Tambien se utilizo la estratificacion para poder garantizar que la distribución de las clases en cada conjunto sea la misma que en el conjunto completo y la habilidad de poder enviar por parametro una semilla para reproducir los resultados.

Esta función 'split_dataset' recibe ciertos parametros que son:

- df: DataFrame de pandas con los datos del dataset
- target_column: Nombre de la columna de las clases en string
- test_size: Proporción numerica del conjunto de prueba
- val_size: Proporción numerica del conjunto de validación
- random_state: Seed para poder reproducir los resultados si se desea

Y lo que finalmente retorna es una tupla con todas las particiones del dataset con la forma (X_train, X_val, X_test, y_train, y_val, y_test).

Esta salida significa:

- X_train: DataFrame con las caracteristicas de entrenamiento
- X_val: DataFrame con las caracteristicas de validación
- X_test: DataFrame con las caracteristicas de prueba
- y_train: Series con las clases de entrenamiento
- y_val: Series con las clases de validación
- y_test: Series con las clases de prueba

El tamaño de cada conjunto se muestra en el Cuadro 1, mostrado a continuación:

Conjunto	Tamaño	Porcentaje
Completo	145586	100 %
Entrenamiento	101910	70 %
Validación	21838	15 %
Prueba	21838	15 %

Cuadro 1: Tamaño del conjunto y particiones de entrenamiento, validación y prueba.

Como se muestra en el Cuadro 1, el conjunto fue particionado correctamente, cumpliendo con el porcentaje esperado.

Para que las particiones fueran uniformes, se utilizo la estratificacion para poder garantizar la distribución uniforme de las clases.

Como es posible observar en el Cuadro 2, la distribucion de clases de manera uniforme hace que se pueda validar de una mejor manera los modelos, ya que se puede ayudar a que el modelo no se entrene con una inclinacion injustificada hacia cierto ataque y los splits sean mas justos para evaluar el modelo en todas sus clases.

Clase	Completo	Entrenamiento	Validación	Prueba
back.	0.006649	0.006653	0.006640	0.006640
buffer_overflow.	0.000206	0.000206	0.000183	0.000229
ftp_write.	0.000055	0.000059	0.000046	0.000046
guess_passwd.	0.000364	0.000363	0.000366	0.000366
imap.	0.000082	0.000079	0.000092	0.000092
ipsweep.	0.004472	0.004465	0.004488	0.004488
land.	0.000131	0.000128	0.000137	0.000137
loadmodule.	0.000062	0.000069	0.000046	0.000046
multihop.	0.000048	0.000049	0.000046	0.000046
neptune.	0.355941	0.355942	0.355939	0.355939
nmap.	0.001085	0.001079	0.001099	0.001099
normal.	0.603300	0.603297	0.603306	0.603306
perl.	0.000021	0.000020	0.000046	0.000000
phf.	0.000027	0.000029	0.000000	0.000046
pod.	0.001415	0.001413	0.001420	0.001420
portsweep.	0.002857	0.002865	0.002839	0.002839
rootkit.	0.000069	0.000069	0.000092	0.000046
satan.	0.006223	0.006221	0.006228	0.006228
smurf.	0.004403	0.004406	0.004396	0.004396
spy.	0.000014	0.000020	0.000000	0.000000
teardrop.	0.006306	0.006300	0.006319	0.006319
warezclient.	0.006134	0.006133	0.006136	0.006136
warezmaster.	0.000137	0.000137	0.000137	0.000137

Cuadro 2: Distribución de clases en los conjuntos: completo, entrenamiento, validación y prueba.

1.2. Entrenamiento, Optimización y Evaluación del Árbol de Decisión

1.2.1. Optimización de Hiperparámetros con Optuna

Como parte del trabajo se realizo la optimizacion de varios hiperparametros para el arbol de decision implementado con Scikit Learn. Para este efecto se utilizo la libreria Optuna en la cual se optimizaron los parametros necesarios de la funcion 'DecisionTreeClasifier' en un estudio de 100 pruebas de optimizacion, se entreno con la particion de entrenamiento y se evaluo las predicciones con la particion de validacion utilizando como resultado el F1-Score promedio macro en base a la prediccion con el conjunto de validacion.

Para ello se creo la funcion 'optimize_decision_tree' que se encarga de recibir por parametro un objeto de estudio de Optuna por medio de la función 'create_study' y dentro de la funcion de optimizacion se definen los rangos de los parametros a optimizar, se define la funcion de arbol de decision de Scikit Learn y se entrena con la particion de entrenamiento y se evalua con la particion de validacion por medio de las funciones 'predict', 'fit' y para generar la puntuacion se utilizo 'f1_score' de Scikit Learn que tambien es utilizada como salida de la funcion de optimizacion.

Se optimizaron los siguientes hiperparametros:

- Profundidad maxima del arbol (max_depth)
- Cantidad minima de observaciones por particion (min_samples_split)
- Cantidad minima de observaciones por hoja (min_samples_leaf)
- Criterio de pureza (criterion)

Para el parametro de profundidad maxima del arbol se optimizo el valor en un rango de 3 a 20. Este rango se escogio debido a las siguientes razones encontradas [8, 7, 2]:

- Valores muy bajos podrian causar que el arbol no capturen los patrones del dataset correctamente y no genere un buen resultado, por lo que empezar con 3 es un buen valor inicial.
- Valores muy altos podrian permitir que el arbol se sobreajuste y no generalice bien, por lo que 20 es un valor optimo.
- El valor limite de 20 es suficiente ya que mas profundidad no se observa que mejoren el resultado y seria un desperdicio de computo.
- El dataset tiene clases desbalanceadas, por lo que profundidades muy altas podrian crear una inclinacion hacia estas clases grandes.

En cuanto al parametro de cantidad minima de observaciones por particion se optimizo el valor en un rango de 2 a 50. Las consideraciones fueron [5, 4]:

- Los balores muy bajos como 2 podrian generar que el arbol se divida excesivamente y que se ajusta al ruido del dataset.
- Se deja el limite en 50 dado que valores bajos dan paso a mas varianza y posibles sobreajustes, mientras que los valores altos podrian permitir una mejor generalizacion, sin llegar al exceso que el arbol casi no se divida.
- El rango de de 2 a 50 es lo suficientemente moderado para poder generar splits con buen soporte, pero tampoco tan pequeño como para que se generen splits debiles como serian con un valor bajo, evitando un arbol muy simple o especifico.

En el caso de la cantidad minima de observaciones por hoja se optimizo el valor en un rango de 1 a 25. Los puntos a tomar fueron [3, 9]:

- Valores bajos pueden crear hojas con muy pocos ejemplos, dando paso a que se sobreajuste a cierto patrones especificos del dataset.
- Los valores altos podrian dar paso a poder generalizar mejor, ya que contiene un numero mas alto de muestras, reduciendo la varianza y posiblemente evitando el sobreajuste.

- Si se obtienen hojas con mas datos es posible que los patrones aprendidos sean mas estables.
- El valor limite de 25 permite suficientes hojas y reducir los nodos del arbol que podria mantener el arbol mas pequeño y eficiente.

Por ultimo, se optimizo el criterio de pureza en un rango de 'gini' y 'entropy'. El criterio para este rango fue [6, 1]:

- Se utiliza solo gini o entropy y no se incluye 'log_loss' debido a que es mas costoso computacionalmente y puede no dar una mejora significativa en contraste con los otros dos.
- Se incluye gini ya que es un algoritmo robusto y rapido computacionalmente.
- En el caso de entropy este es un poco mas costoso que gini dado a que realiza el calculo de logaritmos, pero puede llegar a ser un poco mas preciso.

Para poder ejecutar esta función se creo un estudio de Optuna con la direccion de maximizar la funcion objetivo, en este caso al escogerse como metrica el F1-Score promedio macro se busco mejorar esta puntuacion en todos los estudios.

Se realizaron 100 pruebas de Optuna, suficientes para poder encontrar las mejores tres arquitecturas en un tiempo adecuado, su duracion de ejecucion en Google Colab fue de alrededor de un poco mas de dos minutos, mayores numeros de pruebas realmente no lo mejoraron significativamente por lo que no se justifica la computacion y tiempo mayor de estudio.

Se utilizo la metrica a maximizar lo que es el F1-Score macro, debido a que el dataset KDD99 tiene clases desbalanceadas, por lo que esta metrica permite dar importancia a todas las clases y no solo a algunas que son muy representativase en este conjunto. Otras metricas como el accuracy podrian creer que es un buen modelo para cierto tipo de ataques pero podria estar sesgado y otras clases podrian no ser detectadas correctamente con un rendimiento pobre.

Proceso de Optimizacion Para documentar el proceso de optimizacion se generaron distintos graficos que permiten ver la evolucion de cada estudio realizado y la evolucion de los parametros, en especial de la metrica maximizada que fue el F1-Score.

Para ello se generaron cuatro graficos mostrados en la Figura X, que serian la historia de optimizacion del F1-Score, la distribucion de las F1-Score, importancia de los hiperparametros y una comparacion de las tres mejores arquitecturas por F1-Score.