DISSERTAÇÃO DE MESTRADO PROFISSIONAL

Manutenção de Sistemas de Detecção de Intrusão Baseados em Algoritmos de Aprendizado de Máquina Utilizando *Benchmarks* e *Software* de Extração de *Features*

Luiz Augusto dos Santos Pires

Programa de Pós-Graduação Profissional em Engenharia Elétrica
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
FACULDADE DE TECNOLOGIA
UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASÍLIA
Faculdade de Tecnologia

DISSERTAÇÃO DE MESTRADO PROFISSIONAL

Manutenção de Sistemas de Detecção de Intrusão Baseados em Algoritmos de Aprendizado de Máquina Utilizando *Benchmarks* e *Software* de Extração de *Features*

Luiz Augusto dos Santos Pires

Dissertação de Mestrado Profissional submetida ao Departamento de Engenharia Elétrica como requisito parcial para obtenção do grau de Mestre em Engenharia Elétrica

Banca Examinadora

Prof. Georges Daniel Amvame Nze, Ph.D, FT/UnB _____
Orientador

Felipe Barreto de Oliveira, Mestre, FT/UnB _____
Coorientador

Prof. Fábio Lúcio Lopes de Mendonça, Ph.D _____
Examinador interno

Prof. Laerte Peotta de Melo, Ph.D _____
Examinador Externo

Prof. Robson de Oliveira Albuquerque, Ph.D, FT/UnB _____
Suplente

FICHA CATALOGRÁFICA

PIRES, LUIZ AUGUSTO DOS SANTOS
Manutenção de Sistemas de Detecção de Intrusão Baseados em Algoritmos de Aprendizado de Máquina Utilizando *Benchmarks* e *Software* de Extração de *Features* [Distrito Federal] 2024.
xvi, 65 p., 210 x 297 mm (ENE/FT/UnB, Mestre, Engenharia Elétrica, 2024).
Dissertação de Mestrado Profissional - Universidade de Brasília, Faculdade de Tecnologia.
Departamento de Engenharia Elétrica

1. *Benchmarks*
2. Aprendizado de máquina
3. *Features*
4. Sistemas de detecção de intrusão

I. ENE/FT/UnB
II. Título (série)

PUBLICAÇÃO: PPEE.MP.073

REFERÊNCIA BIBLIOGRÁFICA

PIRES, L.A.D.S (2024). *Manutenção de Sistemas de Detecção de Intrusão Baseados em Algoritmos de Aprendizado de Máquina Utilizando Benchmarks e Software de Extração de Features*. Dissertação de Mestrado Profissional, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 65 p.

CESSÃO DE DIREITOS

AUTOR: Luiz Augusto dos Santos Pires
TÍTULO: Manutenção de Sistemas de Detecção de Intrusão Baseados em Algoritmos de Aprendizado de Máquina Utilizando *Benchmarks* e *Software* de Extração de *Features*.
GRAU: Mestre em Engenharia Elétrica ANO: 2024

Luiz Augusto dos Santos Pires
Depto. de Engenharia Elétrica (ENE) - FT
Universidade de Brasília (UnB)
Campus Darcy Ribeiro
CEP 70919-970 - Brasília - DF - Brasil

AGRADECIMENTOS

Agradeço a Deus por ter me dado a sabedoria e paciência necessária durante o trabalho e por ter colocado as pessoas certas para me ajudar em sua elaboração.

Agradeço aos meus familiares por terem me incentivado e tranquilizado nos momentos mais difíceis.

Agradeço ao meu orientador, Prof.Georges Daniel Amvame Nze, e ao meu coorientador, Felipe Barreto de Oliveira, por todo o apoio durante a realização do trabalho.

RESUMO

Com o aumento de ataques em rede, impulsionado pelo avanço da inteligência artificial e da Internet das Coisas (IoT), que muitas vezes utiliza dispositivos com segurança frágil, torna-se crucial desenvolver mecanismos para a manutenção de Sistemas de Detecção de Intrusão (IDS). Uma forma de possibilitar isso, é utilizar estratégias para atualizar de forma constante algoritmos de aprendizado de máquina que classificam o tráfego de rede. Um dos pontos mais críticos para isso, é a extração de *features* da rede, pois são usadas como entrada para o treinamento de modelos de aprendizado de máquina. Assim, este trabalho teve por objetivo criar um *software* para extrair de forma automatizada 26 *features* do *dataset* NSL-KDD e analisar os principais fatores e *benchmarks* utilizados na literatura para treinar modelos de *machine learning* (ML), com o intuito de se obter um algoritmo ótimo de classificação a partir do conjunto de *features* extraídas pela ferramenta proposta. O algoritmo ótimo resultante a partir das 26 *features* extraídas pelo *software* foi obtido ao se mesclar o *dataset* do NSL-KDD com o Kyoto 2006 +, tendo valores de acurácia em 83,81%, *recall* em 75,65%, *precision* em 94,87% e *f1-score* em 84,17 % no arquivo de teste do NSL-KDD. Ainda foi utilizado o *software* e algoritmo de ML propostos para avaliar o desempenho no *dataset* CIC-IoT2023, que envolve grande quantidade de tráfego IoT, percebendo-se, após os testes no *dataset*, a necessidade de realizar a manutenção do algoritmo de ML para melhorar a classificação no CIC-IoT2023. Como último resultado, o *software* para extração de *features* foi registrado no INPI (Instituto Nacional da Propriedade Industrial) na categoria de programa de computador (BR512023001038-3).

ABSTRACT

With the increase in network attacks, driven by the advancement of artificial intelligence and the Internet of Things (IoT), which often uses devices with weak security, it becomes crucial to develop mechanisms for the maintaining of Intrusion Detection Systems (IDS). One way to make this possible is to use strategies to constantly update machine learning algorithms that classify network traffic. One of the most critical points is the extraction of features from the network, as they are used as input for training machine learning models. Thus, this work aimed to create a software to automatically extract 26 features from the dataset NSL-KDD and analyze the main factors and benchmarks used in the literature to train machine learning (ML) models, with the aim of obtaining an optimal classification algorithm from the set of features extracted by the proposed tool. The resulting optimal algorithm from the 26 features extracted by the software was obtained by merging the NSL-KDD dataset with Kyoto 2006 +, having accuracy values of 83.81%, recall of 75.65%, precision of 94.87% and f1-score of 84.17% in the NSL-KDD test file.The proposed software and ML algorithm were also used to evaluate the performance in the CIC-IoT2023 dataset, which involves a large amount of IoT traffic. After testing on the dataset, it was noted that there was a need to maintain the ML algorithm to improve the classification in CIC-IoT2023. As a final result, the software for extracting features was registered with the INPI (National Institute of Industrial Property) in the computer program category (BR512023001038-3).

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE TERMOS E SIGLAS

API	*Application Programming Interface*
AUC	*Area Under the Curve*
AWS	*Amazon Web Service*
CSV	*Comma-Separated Values*
DB	*Database*
DDoS	*Distributed Denial-of-Service*
DL	*Deep Learning*
DNN	*Deep Neural Networks*
DoS	*Denial of Service Attack*
DR	*Detection Rate*
DT	*Decision Tree*
ETL	*Extract, Transform and Load*
FNR	*False Negative Rate*
FPR	*False Positive Rate*
GAN	*Generative Adversarial Networks*
HIDS	*Host-Based Intrusion Detection System*
ICITS	*International Conference on Information Technology & Systems*
IDS	*Intrusion Detection Systems*
IG	*Information Gain*
INPI	Instituto Nacional da Propriedade Industrial
IoT	*Internet of Things*
JSON	*JavaScript Object Notation*
KNN	*K-Nearest Neighbors*
LR	*Logistic Regression*
MFO	*Moth Flame Optimization*

ML	*Machine Learning*
NB	*Naive Bayes*
NIDS	*Network-Based Intrusion Detection System*
NoSQL	*Not Only Structured Query Language*
PCAP	*Packet Capture*
R2L	*Remote to Local Attack*
RF	*Random Forest*
RISTI	Revista Ibérica de Sistemas e Tecnologias de Informação
ROC	*Receiver Operating Characteristic*
SVM	*Support Vector Machine*
TNR	*True Negative Rate*
U2R	*User to Root Attack*

1 INTRODUÇÃO

Em resposta à crescente onda de ataques cibernéticos, um arsenal de ferramentas se tornou necessário para proteger os sistemas de computação. Cita-se o uso de *firewalls*, *softwares* de antivírus e *Intrusion Detection Systems* (IDS) como alguns exemplos dessas ferramentas, cada uma desempenhando papel crucial na defesa contra ameaças cibernéticas.

Entre as possíveis categorias de um IDS estão o *Host-Based Intrusion Detection System* (HIDS) , que monitora as atividades de um *host* em específico, e o *Network-Based Intrusion Detection System* (NIDS) , que monitora de maneira constante o tráfego da rede em busca de comportamentos maliciosos.

Para construir NIDS mais eficientes, pesquisadores estão explorando o potencial de algoritmos de *Machine Learning* (ML) e *Deep Learning* (DL) . Esses algoritmos são capazes de extrair informações relevantes do tráfego de rede, permitindo que o NIDS sejam treinados para identificar e classificar com precisão os fluxos de rede, distinguindo entre atividades normais e comportamentos maliciosos.

Numerosos estudos na literatura, citando-se os contidos no *survey* [19] e nos *reviews* de [20] e [1], utilizam um conjuntos de dados elaborados especificamente para treinar algoritmos de ML e DL, os chamados *benchmarks*, para classificar os fluxos de rede.

De acordo com [1], entre os conjuntos de dados mais empregados para essa finalidade estão o NSL-KDD, o KDDCUP 99, o UNSW-NB15, o CIC-IDS2017 e o CSE-CIC-IDS2018. Além disso, o NSL-KDD é destacado em [1] como o principal *dataset* para avaliar a performance de algoritmos de ML e DL em trabalhos científicos entre o período de 2017 até 2020. A Figura 1.1 mostra a distribuição de uso de *benchmarks* na literatura durante o período citado.

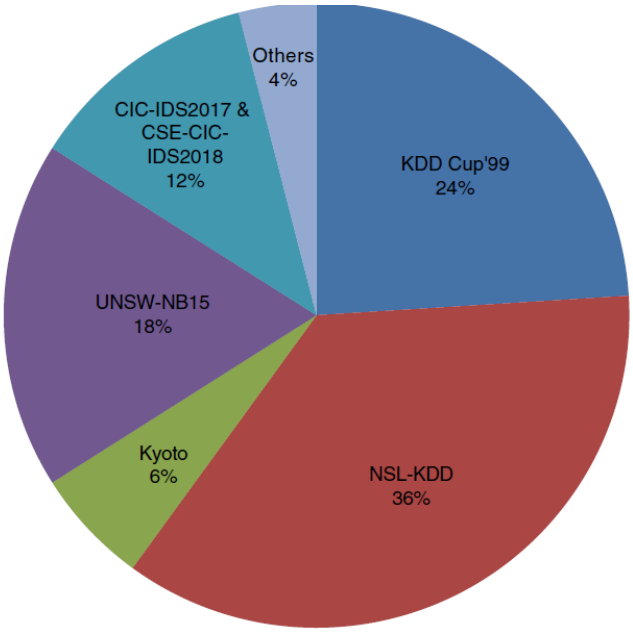

Figura 1.1: Gráfico que mostra uso de *benchmarks* na literatura retirado de [1]

Entre os pontos que tornam o *dataset* NSL-KDD atraente para os pesquisadores é a quantidade de resultados relacionados a testes neste *dataset* e o fato de ser uma versão refinada do KDD99 removendo amostras que traziam viés a modelos de ML [1].

Entretanto, mesmo sendo um *dataset* bastante utilizado ele apresenta como ponto negativo o fato de ser relativamente antigo e não ter sofrido atualizações significativas da sua base de dados como aponta [1] ao afirmar que a maioria dos trabalhos que eles analisaram, que utilizam em grande parte o NSL-KDD como base de treino, falham em detectar novos tipos de ataques por conta da falta de atualização em suas bases de dados.

No esforço de melhorar os classificadores de tráfego, que utiliza a base do NSL-KDD para treino de algoritmos de ML, os trabalhos na literatura tem realizado a manutenção dos algoritmos a partir: de modificações no modelo de ML, no conjunto de *features* (como no *survey*, [19] e o *review* [20]) e no conjunto de dados de treinamento [21] [22].

Uma ponto a ser ressaltado sobre o NSL-KDD nos trabalhos citados é o fato dos algoritmos de ML treinados com ele realizam quase que unanimemente o treino e teste no mesmo *dataset*. Excetuando-se pelos trabalhos encontrados de [22], que mescla durante o treinamento o NSL-KDD com o UNSW-NB15, e [8], que avalia sua própria base de teste proveniente de coleta em rede, não encontrou-se nenhum trabalho que realiza-se treino ou teste com datasets mesclados junto ao NSL-KDD. Tal fato vai de encontro com o que é dito por [1] que relata a escassez de atualizações na base de dados de *benchmarks*, sendo que o principal *dataset* exposto pelos autores o NSL-KDD.

O principal motivo notado para que isto ocorra é o fato dos trabalhos citados acabarem dando enfoque na exploração de técnicas e modelos para melhorar as métricas de performance no conjunto de teste do NSL-KDD que é um das principais formas de comparação de performance na comunidade científica como exposto por [1].

É fundamental experimentar a alteração de tais fatores, entretanto é mister realizar a manutenção do conjunto de dados, de forma a garantir que a base de dados esteja com dados atualizados e assim garantir a detecção de novos padrões de ataques em IDS como aponta [1].

Para realizar as etapas que envolvem a manutenção dos algoritmos de ML (que inclui a manutenção de dados e manutenção do modelo de ML) é importante implementar as etapas do ciclo de ML mostrado na Figura 1.2.

No contexto de algoritmos de ML para IDS, pode-se descrever as etapas do ciclo de dados da Figura 1.2 da seguinte forma:

- *Collect* (Coleta): Envolve a coleta de dados diversas fontes, isto é de diversos *datasets* que coletam tráfego de rede e os usam para treinar o IDS;
- *Curate* (Curadoria): a seleção dos dados relevantes para a solução do problema, que no caso do NSL-KDD, por exemplo, envolve a seleção de pacotes que usam o protocolo TCP, UDP e ICMP [21];
- *Transform* (Transformação): Para IDS que usam ML envolve o processamento para extrair as *features* relevantes para o problema, que são efetivamente as entradas dos algoritmos de ML, a partir dos

dados brutos vindos da rede;

- *Validation* (Validação): Envolve a criação de mecanismos para o controle de qualidade dos dados, garantindo que as amostras adicionadas na base de treinamento sejam relevantes. Por exemplo, pode-se filtrar amostras repetidas com em [21].

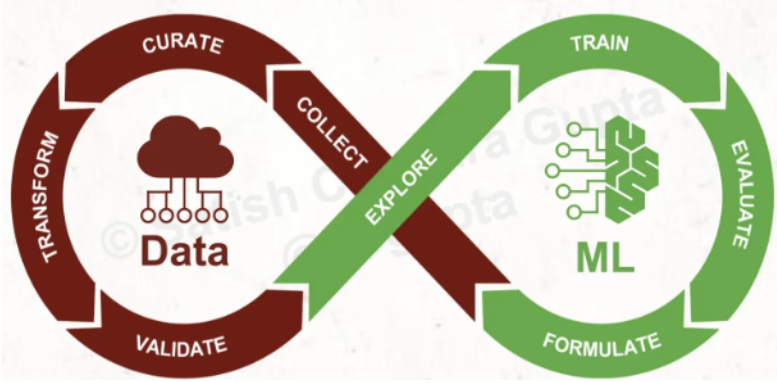

Figura 1.2: Ciclo de vida de ML retirado de [2]

Ao recorrer a literatura, existem alguns trabalhos preocupados com a manutenção dos dados do IDS e que, portanto, desenvolvem ferramentas para extração de *features* no NSL-KDD, as quais vão permitir a manutenção de tal base de treino, realizando um esforço a nível de ETL (*Extract, Transform and Load*) para converter os dados brutos da rede (provenientes da interface de rede ou mesmo de arquivos PCAP (*Packet Capture*))) em *features* do NSL-KDD. Como é o caso de [12], [13] e [8], porém não descrevem em detalhes a sequência de passos específica usada em cada arquitetura para o processo de extração de *features*.

Ainda em relação a manutenção dos dados, alguns trabalhos reconhecem que existe uma lacuna em relação ao processamento de *datasets* em tempo real, como [19] e [1]. Acredita-se, conforme descrito por [15], que a escassez de trabalhos nesse contexto também decorre da falta de detalhamento do algoritmo e da arquitetura associados ao módulo responsável pela extração de *features*, sobretudo, em *datasets* como NSL-KDD, que não possuíam, até onde se tem conhecimento, um *software* específico para fazer isso de forma automatizada até ser proposto o trabalho de [15] (que é uma das contribuições do presente trabalho).

O fato do NSL-KDD não possuir *softwares* para extrair *features* gera ainda outras lacunas, como a dificuldade de se testar a eficácia das suas *features* em outros *benchmarks* que não possuem suas mesmas *features*. Também dificulta a manutenção de um IDS a partir do acréscimo de dados para serem usados no treinamento dos algoritmos de ML.

Em suma, ao não se preocupar devidamente com o ciclo de vida dos dados acaba-se por se prejudicar as experimentações relativas aos algoritmos de *machine learning* no NSL-KDD, já que não será possível o incremento adequado da base de treino e avaliação de performance a partir de dados de outros *benchmarks*, sendo fundamental propor mecanismos que solucionem esses problemas.

1.1 OBJETIVOS E CONTRIBUIÇÕES

O presente trabalho tem o objetivo desenvolver e implementar uma solução integradora para a manutenção de um IDS baseado em algoritmos de ML, utilizando o *dataset* NSL-KDD como banco de dados de referência. Sendo a manutenção do IDS feita a partir: de técnicas de extração de *features*; do uso de diferentes modelos de ML; da combinação de dados de diferentes *benchmarks*; e da avaliação de desempenho a partir do arquivo de teste do NSL-KDD e de amostras de outros *benchmarks* (Kyoto 2006 + e CICIoT2023 [4]).

Para cumprir o objetivo exposto foi necessário antes cumprir os seguintes objetivos específicos:

- Desenvolvimento de *Software*: Criar um *software* especializado para a extração em tempo quase real de 26 *features* do *dataset* NSL-KDD a partir de dados de rede, incluindo arquivos PCAP, facilitando a formatação dos dados brutos de diversos *benchmarks* em um formato utilizável para algoritmos de ML.
- Implementação de Algoritmo de Amostragem: Introduzir um algoritmo de amostragem, inspirado no trabalho de [21], que assegura a manutenção efetiva dos algoritmos de ML, através da incorporação criteriosa de amostras de outros *benchmarks* ao conjunto de treino.
- Melhoria de Classificação de Tráfego: Propor um algoritmo de ML que utilize exclusivamente as *features* extraídas pelo *software* desenvolvido, com o objetivo de otimizar a classificação do tráfego de rede e avaliar o desempenho do IDS em testes subsequentes, tanto no arquivo de teste do NSL-KDD quanto em amostas de outras bases de dados, como Kyoto 2006 + e CICIoT2023.

Como contribuições importantes do trabalho, ressalta-se que foi produzido um *software* registrado no Instituto Nacional da Propriedade Industrial (INPI) na categoria de programa de computador (BR51202300 1038-3), que permite a extração de *features* em tempo quase real (adaptado posteriormente para permitir a extração também a partir de arquivos PCAP).

Isso possibilitou a geração de mais amostras a partir das *features* geradas pelo *software* para serem agregados em *datasets* de treino ou teste e permitiu solucionar a lacuna envolvendo testes da eficácia das *features* do NSL-KDD em outros conjuntos de dados e adição de amostras para manutenção de algoritmos de ML.

Cita-se também, que foi produzido o artigo [15], apresentado na conferência *International Conference on Information Technology & Systems 2024 (ICITS'24)* e publicado na Revista Ibérica de Sistemas e Tecnologias de Informação (RISTI), que detalha a arquitetura do *software* de extração de *features* do NSL-KDD proposto, bem como a metodologia empregada para se extrair as *features* a partir da referida arquitetura.

1.2 ORGANIZAÇÃO DO TRABALHO

O trabalho está organizado em cinco capítulos, cada um com um foco específico que contribui para a compreensão e a execução das pesquisas e inovações apresentadas. A divisão ocorre da seguinte forma: O

Capítulo 2 apresenta os conceitos e informações fundamentais em relação aos *datasets* que serão utilizados no trabalho. Apresenta também as ferramentas utilizadas para construir a arquitetura do *software* que extrai as *features*. Além disso, expõe conceitos relacionados aos modelos de ML utilizados, de técnicas para extração de *features* e de métodos para se treinar o NSL-KDD. Ainda, o capítulo contém uma revisão de trabalhos relativos a IDS que fazem uso dos *datasets* analisados e que dão ênfase ao processo de extração de *features* nos algoritmos de ML para classificação do tráfego da rede; o Capítulo 3 descreve o método proposto neste trabalho que consiste na descrição das etapas necessárias para confecção do *software* de extração de *features*, os passos necessários para construção do algoritmo ótimo, que usa um modelo de ML para classificar o tráfego a partir das *features* extraídas pelo *software* proposto, e, por último, as etapas para avaliar o algoritmo de ML na classificação das amostras do CIC-IoT *dataset* 2023; o Capítulo 4 apresenta os resultados obtidos, mostrando o funcionamento do *software* para extração de *features*. Também revela o resultado dos experimentos a partir da variação de fatores nos algoritmos de ML para se chegar ao algoritmo ótimo, e os resultados após os testes no CIC-IoT *dataset* 2023. Ainda apresenta uma breve discussão, relatando os desafios e cuidados que foram tomados durante o trabalho, e; por último, o Capítulo 5 apresenta a conclusão do trabalho, propondo possíveis trabalhos futuros.

2 REFERENCIAL TEÓRICO

2.1 FUNDAMENTOS

Para compreender o *software* proposto e seu processo de extração de *features* é essencial entender os detalhes do NSL-KDD e os programas que compõe o *software* feito. Além disso, é crucial dominar os algoritmos de aprendizado de máquina (ML) utilizados no trabalho. Para isso, é necessário conhecer os modelos de classificação aplicados, bem como as técnicas de experimentação e avaliação de desempenho dos algoritmos.

2.1.1 Datasets

Os *datasets* escolhidos se baseiam em conexões. Uma conexão (ou fluxo) é formada a partir do agrupamento de pacotes que possuem os mesmos elementos de uma das duas tuplas: con_ida ou con_volta , descritas na equação 2.1.

$$\begin{aligned} con\_ida &= (ip_{origem}, porta_{origem}, ip_{destino}, porta_{destino}, protocolo) \\ con\_volta &= (ip_{destino}, porta_{destino}, ip_{origem}, porta_{origem}, protocolo) \end{aligned} \tag{2.1}$$

Sendo:

- ip_{origem} : Ip de origem do pacote no cabeçalho ip
- $porta_{origem}$: Porta de origem que está no cabeçalho da camada de transporte
- $ip_{destino}$: Ip de destino do pacote no cabeçalho ip
- $porta_{destino}$: Porta de destino que está no cabeçalho da camada de transporte
- $protocolo$: Protocolo utilizado. Pode ser: TCP, UDP ou ICMP.
- con_ida: Tupla em que os pacotes que vão da origem ao destino se enquadram
- con_volta: Tupla em que os pacotes que vão do destino para origem se enquadram

Simplificadamente, uma conexão é formada por um conjunto de pacotes que saem de um mesmo ip_{origem} e $porta_{origem}$ e vão para o mesmo $ip_{destino}$ e $porta_{destino}$ durante um intervalo de tempo (ou, no caso do TCP, quando a conexão é encerrada). Leva-se em consideração que tanto os pacotes enviados da origem para o destino quanto os pacotes retornados do destino para a origem pertencem à mesma conexão.

2.1.1.1 NSL-KDD

O NSL-KDD é um *dataset* que é resultado de um refinamento de outro *dataset*, o KDD99. O processo de sua confecção começa com seu tráfego de rede sendo coletado no programa de avaliação de IDS chamado 1999 DARPA *Intrusion Detection Evaluation Dataset* [23]. Contendo 7 semanas de tráfego, que foram separadas em semanas de dados para treino e testes (sendo 2 semanas de testes), conforme explica [21]. Esse tráfego do DARPA é gerado por simulação em rede virtual, o que é considerado como um dos pontos negativos, pois acaba por não refletir totalmente o comportamento de conexões em ambientes reais de rede de computadores.

A partir desse conjunto de dados foi formado o *dataset* KDD99, que contém aproximadamente 4.900.000 instâncias (amostras), sendo cada uma composta por 41 *features* relacionadas a conexão de rede. Cada amostra ainda contém um rótulo que identifica o tipo de tráfego da conexão, ou seja, se é um tráfego malicioso ou normal.

O KDD99 contém 22 tipos de ataque no conjunto de treino e mais 17 adicionais no conjunto de teste. Como mostrado no diagrama na Figura 2.1.

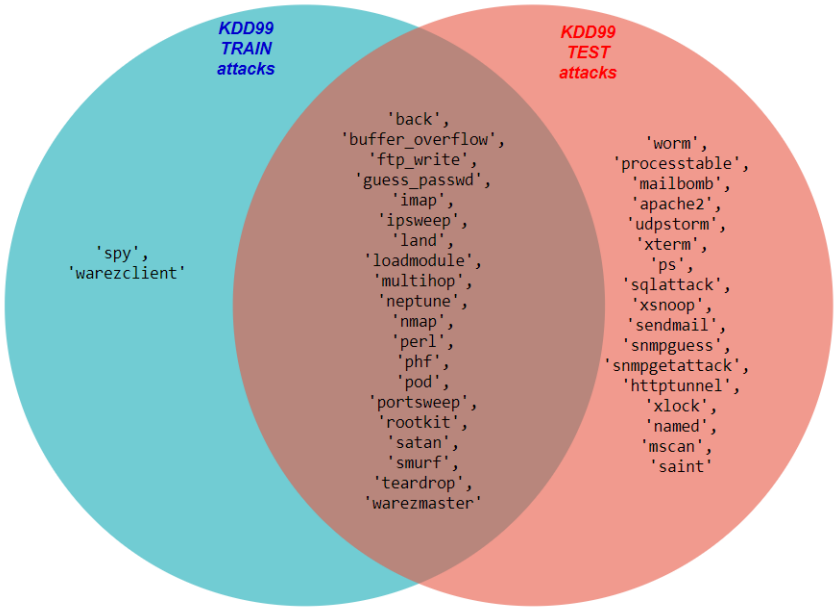

Figura 2.1: Diagrama de Venn mostrando a distribuição dos ataques do KDD 99 no conjunto de treino e teste

Os ataques no *dataset* podem ser enquadrados, em uma das 4 categorias descritas:

- *Denial of Service Attack* (DoS) : ataque que sobrecarrega um sistema com uma grande quantidade de solicitações falsas, tornando os serviços do sistema indisponíveis para um usuário legítimo.
- *User to Root Attack* (U2R): ataque que começa com o atacante conseguindo informações para acessar uma conta normal de usuário, e a partir disso, explora formas de conseguir acesso *root* ao sistema.
- *Remote to Local Attack* (R2L): acontece quando um atacante usa da capacidade de enviar pacotes a uma máquina na rede para explorar vulnerabilidades que o permitam conseguir acesso como usuário a ela mesmo sem ter autorização.

- *Probing Attack*: Tentar coletar informações importantes de uma rede a partir de ferramentas e métodos para comprometê-la posteriormente.

A Tabela 2.1 mostra os ataques específicos de cada uma das categorias explicitadas.

Tabela 2.1: Ataques contidos em cada categoria.

Dos	mailbomb neptune	worm smurf	apache2 pod	back teardrop	land processtable	udpstorm
Probe	ipsweep	portsweep	nmap	mscan	saint	satan
R2L	ftp_write guess_passwd snmpguess	warezclient httptunnel xsnoop	spy phf	imap multihop snmpgetattack	named sendmail	xlock warezmaster
U2R	ps	perl	xterm	loadmodule	sqlattack	
	rootkit			buffer_overflow		

É importante também relatar sobre as *features* deste *dataset*. Estas podem ser divididas em 4 categorias, como no quadro mostrado na Figura 2.2. A seguir explica-se os critérios para se dividir as categorias.

Category I	duration (0–to–54451)	protocol type (1, 2, 3)	service (1–to–70)	src_bytes (0–to– 1379963888)	dst_bytes (0–to– 309937401)
	Flag (1–to–11)	Land (0, 1)	wrong_fragment (0, 1, 3)	Urgent (0–to–3)	
Category II	Hot (0–to–101)	num_failed_logins (0–to–4)	logged_in (0,1)	num_compromised (0–to–7479)	root_shell (0, 1)
	su_attempted (0, 1)	num_root (0–to–7468)	num_file_creations (0–to–100)	num_shells (0–2)	num_access_files (0–9)
	is_guest_login (0, 1)	is_hot_logins (0, 1)	num_outbound_cmds (0)		
Category III	Count (0–511)	srv_count (0–511)	serror_rate (0–to–1)	srv_serror_rate (0–to–1)	rerror_rate (0–to–1)
	srv_rerror_rate (0–to–1)	same_srv_rate (0–to–1)	diff_srv_rate (0–to–1)	srv_diff_host_rate (0–to–1)	
Category IV	dst_host_count (0–to–255)	dst_host_srv_count (0–to–255)	dst_host_same_srv_rate (0–to–1)	dst_host_diff_srv_rate (0–to–1)	dst_host_same_src_port_rate (0–to–1)
	dst_host_srv_diff_host_rate (0–to–1)	dst_host_serror_rate (0–to–1)	dst_host_srv_serror_rate (0–to–1)	dst_host_rerror_rate (0–to–1)	dst_host_srv_rerror_rate (0–to–1)
	class_label (0–to–1)				

Figura 2.2: Quadro retirado de [3] mostrando as *features* do *dataset* KDD-CUP 99 separadas por categoria.

- Categoria I: São as *features* relacionadas a conexão em andamento, com informações retiradas do próprio cabeçalho TCP/IP
- Categoria II: São as *features* que estão contidas nas porções de dados dos pacotes. Elas podem ser complexas de serem obtidas por conta de criptografia e outros complicadores.

- Categoria III: São as *features* que geram estatísticas das conexões dos últimos 2 segundos com a conexão atual.
- Categoria IV: São as *features* que geram estatísticas das últimas 100 conexões com a conexão atual.

Uma *feature* que carrega uma complexidade adicional é a *feature flag* da categoria I. Tal complexidade advém da análise das *flags* do cabeçalho do protocolo TCP nos pacotes do fluxo (conexão), avaliando se a conexão foi iniciada normalmente (por meio do *3-way handshake*) e como foi encerrada (se houve uma interrupção abrupta por parte do cliente ou servidor ou se terminou graciosamente). Considerando o exposto, os possíveis valores desta *feature* são mostrados na Figura 2.3.

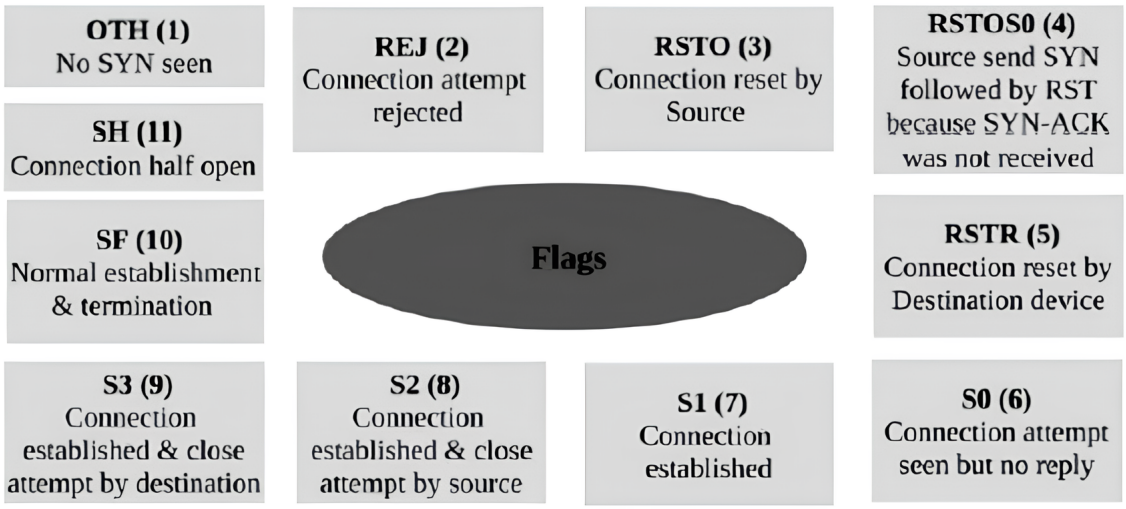

Figura 2.3: Valores possíveis da *feature* flag [3]

O KDD 99 *dataset* possui alguns problemas, como suas amostras repetidas, que fazem com que os modelos criem um viés para as amostras mais frequentes. A partir disso, o *dataset* NSL-KDD (que é um refinamento do KDD99) realiza a remoção de amostras repetidas.

Após remover as amostras repetidas, é utilizado outro critério para remoção de amostras, que consta com as etapas a seguir:

- Dividir o *dataset* KDD99 em 3 partes e utilizando em cada parte 7 modelos de ML diferentes, ou seja, treinando 21 modelos no total.
- Após o treinamento as amostras são agrupadas com base na quantidade de modelos que previram seu rótulo corretamente.
- Em seguida ocorre a remoção de amostras, de forma que as amostras que estão em grupos com mais acertos serão proporcionalmente mais removidas do que as que tiveram menos acertos.

Os procedimentos completos para gerar o NSL-KDD a a partir do KDDCUP99 estão mapeados em [21]. O resultado desse processamento é um *dataset* com um conjunto de amostras de treino e teste como explicitado na Tabela 2.2.

Tabela 2.2: Distribuição de instâncias no conjunto de treino e teste no NSL-KDD. Adaptada de [17]

Class	**Training Instances**	**Testing Instances**
Normal	67343	9711
DoS	45927	7458
Probe	11656	2421
R2L	995	2754
U2R	52	200
Total	**125973**	**22544**

2.1.1.2 Kyoto 2006+ Dataset

Diferente do NSL-KDD, seu tráfego é gerado a partir da coleta de 3 anos de tráfego real. Começando a coleta em Novembro de 2006 até Agosto de 2009. Cada amostra deste *dataset* consiste em 14 *features* estatísticas que são derivadas do KDD 99, e portanto do NSL-KDD, e de 10 *features* adicionais. O conjunto completo das *features* do Kyoto *dataset* é mostrado na Tabela 2.3. As informações a respeito das *features* exclusivas do Kyoto *Dataset* são mostradas na Tabela 2.4.

Tabela 2.3: Features do Kyoto 2006+ *Dataset* [17].

Feature Type	**Feature**
Conventional features	Duration, Service, Source_bytes, Destination_bytes, Count, Same_srv_rate, Serror_rate, Srv_serror_rate, Dst_host_count, Dst_host_srv_count, Dst_host_same_src_port_rate, Dst_host_serror_rate, Dst_host_srv_serror_rate, Flag (14)
Additional features	IDS detection, Malware detection, Ashula detection, Label, Source_IP_Address, Source_Port_Number, Destination_IP_Address, Destination_Port_Number, Start Time, Duration (10)

Tabela 2.4: Features adicionais do Kyoto 2006+ Dataset [18].

IDS_detection	Utiliza o Symantec IDS para detectar um ataque. '0' indica que nenhum alarme foi disparado, outros números indicam um ataque.
Malware_detection	Utiliza o software clamav software para detectar um malware. '0' indica que nenhum ataque foi identificado e uma *string* indica o tipo de ataque
Ashula_detection	Utiliza um software dedicado chamado Ashula para identificar se códigos *shell* e códigos de exploração foram usados na conexão. '0' indica que não foram usados, outros números indicam que houve uso.
Label	Indica se a conexão é normal ('1'), um ataque conhecido ('-1') ou um ataque desconhecido ('-2')
Source IP Address	Indica o ip de origem mascarado usado na conexão
Source Port Number	Indica a porta de origem usada na conexão
Destination IP Address	Indicao ip de destino mascarado usado na conexão
Destination Port Number	Indica a porta de destino usada na conexão
Start Time	Indica quando a conexão iniciou
Duration	Indica o tempo que a conexão ficou estabelecida

Esse *dataset* contém uma grande quantidade de instâncias, mais de 93 milhões (como expõe [17]), sendo que a lista completa de instâncias (amostras) separadas entre tráfego normal e malicioso (conhecidos e desconhecidos).

No presente trabalho considerou-se seu uso para verificar sua eficácia em complementar o *dataset* NSL-KDD. Isto por conta de dois motivos, expostos a seguir:

- O primeiro motivo é o fato de terem um subconjunto de *features* significativo em comum. Até onde se tem conhecimento, é o que tem mais *features* em comum com o NSL-KDD (fora o KDD99 que deu origem ao NSL-KDD). Este motivo é importante, pois ao terem *features* em comum não é necessário processamento a nível de ETL para serem usadas junto ao NSL-KDD em algoritmos de ML.
- O segundo motivo é o fato de serem bases de dados distintas, o que contribui para a assimilação de novos padrões de ataques pelos modelos de ML e como consequência uma melhor generalização pelos algoritmos de ML.

Como observação, cita-se que, até onde se tem conhecimento, não foi encontrado trabalho que realizasse a combinação destes dois*datasets*, sendo somente visto uma tentativa de fusão, de forma a acrescentar *features* e não amostras, do NSL-KDD com o UNSW-NB 15 em [22] como já explicitado.

2.1.1.3 CIC-IoT *dataset* 2023

Com o intuito de testar a eficiência das *features* e do conjunto de dados do NSL-KDD na classificação em ambientes de *Internet of Things* (IoT), foi usado o *dataset* CIC-IoT *dataset* 2023 [4]. O *dataset* apresenta como vantagem o fato de conter tráfego recente e real de dispositivos em redes.

Este *dataset* é feito a partir de uma rede de dispositivos IoT, composta por 105 dispositivos. Neste *dataset* dispositivos IoT lançam ataques a outros dispositivos IoT. No total o *dataset* possui 33 ataques, que são classificados em 7 categorias. São elas: *Distributed Denial-of-Service* (DDoS), DoS, *Recon*, *Web-based*, *brute force*, *spoofing*, e *Mirai*.

Como observação em relação aos ataques desse *dataset*, cita-se que o *Mirai* envolve uma série de outros ataques. Isto porque este tem por objetivo encontrar máquinas vulneráveis (usando de *probe attacks*), ganhar o controle delas (atráves de *Brute Force*, por exemplo) e assim lançar ataques em outros dispositivos (no caso do Mirai, os bots são principalmente usados para lançar ataques DDoS), usando as máquinas escravizadas como intermediários.

O *dataset* consta com 548 GB de tráfego coletado em arquivos PCAP. Cada categoria de ataque, bem como o tráfego normal, são separados em arquivos PCAP diferentes, o que permite a adição imediata de rótulos de tráfego por terceiros nas conexões de rede (sendo que para adicionar um rótulo basta colocar, por exemplo, todos os fluxos de um arquivo com tráfego malicioso com o rótulo "ataque"e os de tráfego legítimo com rótulo "normal").

Em [4] é relatada a estratégia usada para processar os arquivos PCAP, convertendo-os em 47 *features*, que tem relação com conexões de rede. O princípio consiste em fracionar os arquivos PCAP grandes em

vários pequenos de 10 MB para então realizar o processamento em paralelo dos mesmos a fim de obter as 47 *features*. Por último as *features* de cada arquivo são mescladas em um arquivo *Comma-Separated Values* (CSV). Tal procedimento é mostrado na Figura 2.4.

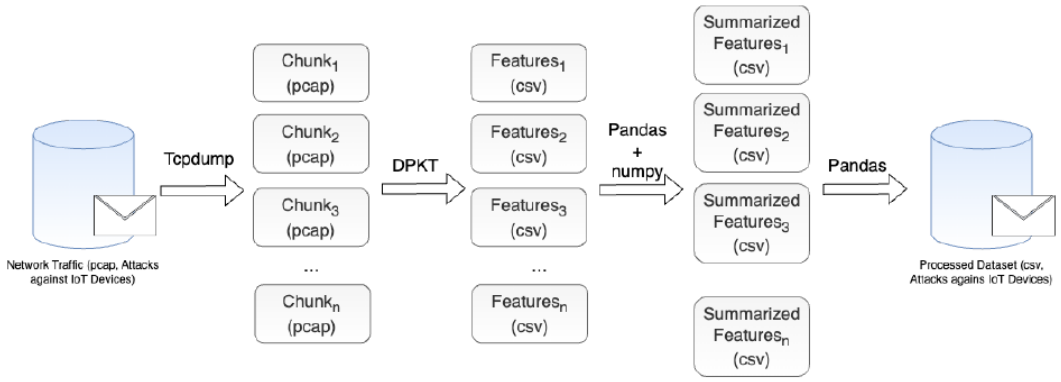

Figura 2.4: Processamento para obter as *features* no CIC-IoT *dataset* 2023 retirado de [4].

Vale ressaltar que as *features* presentes neste trabalho são em sua maioria diferentes das presentes no NSL-KDD podendo ser consultadas em [4].

2.1.2 Ferramental utilizado

Para construir o software que extrai as *features* do NSL-KDD foi necessário utilizar de diversos programas que possibilitassem gerar uma arquitetura de *software* simples e concisa.

2.1.2.1 Mongo DB

Mongo DB é um banco de dados de documentos de uso geral. O Mongo DB guarda os dados no formato de documentos JSON (*JavaScript Object Notation*), o que representa uma vantagem para os desenvolvedores de aplicativo, pois os documentos são facilmente mapeados em objetos da aplicação.

Os campos em um documento podem variar. Sendo que, os campos em um documento podem ser comparados as colunas em um banco de dados tradicional. A diferença é que uma coluna, neste caso poderia guardar tipos de dados muito mais genéricos como dicionários e *arrays*.

Um conjunto de documentos é chamado de coleção, sendo equiparável a uma tabela em um banco de dados relacionais. É interessante ressaltar que documentos em uma mesma coleção podem ter campos diferentes, ou seja, não tem um esquema fixo como em um banco relacional. Ainda, cada coleção está associada a um banco de dados Mongo DB.

Entre as principais vantagens em se utilizar o mongo DB estão a sua alta disponibilidade, pois mantém mais de uma cópia dos dados de forma automática ao se criar um banco de dados. Um conjunto replicado vai permitir a proteção contra tempo de inatividade em caso de falha do sistema ou manutenção planejada como explicado por [24].

Ainda, outra vantagem é a utilização de índices para permitir a execução de consultas no banco de

forma eficiente, sendo que a escolha adequada de índices pode tornar as consultas mais rápidas como explica [24].

Por último, cita-se também seu suporte em diferentes linguagens de programação, sendo que as bibliotecas têm desenvolvimento ativo para implementar novas funcionalidades emergentes, *patches* de segurança e concertando eventuais *bugs*.

2.1.2.2 Redis

O Redis é um banco de dados *open source* que permite o armazenamento de um conjunto de dados composto por chave/valor *Not Only Structured Query Language* (NoSQL) em memória. O fato de ser um banco que guarda todos os dados em memória permite uma alta performance para ler e escrever dados como explica [25].

Ele é muito utilizado quando se precisa de uma resposta rápida na aplicação, sendo ideal para ser usado em processos que necessitam de análise em tempo real, como, por exemplo, processos de *machine learning* que exigem processamento com baixa latência como aponta [25].

2.1.2.3 Dialog

O Dialog é um programa que permite criar interfaces gráficas a partir de um terminal. É possível criar caixas de diálogo a partir de simples *scripts* em *bash*. Essas caixas de diálogo interagem com o usuário de diferente formas. Entre as principais caixas de diálogo do Dialog, pode-se citar:

- Msgbox: Mostra uma mensagem ao usuário e espera o OK dele.
- Inputbox: Permite ao usuário inserir um valor.
- Menu: Apresenta uma lista de opções para o usuário selecionar.
- Checklist: Um menu em que o usuário pode selecionar várias opções.

Para entender mais sobre as funcionalidades do Dialog recomenda-se consultar em [26], que fornece uma visão mais aprofundada das caixas de diálogo e de seus atributos.

2.1.2.4 Docker

O Docker é uma plataforma aberta para desenvolver e rodar aplicações como explica [5]. A plataforma permite a separação entre aplicação e infraestrutura, garantindo uma entrega de *software* mais rápida. Ele ainda permite empacotar e executar aplicativos em ambientes isolados chamados *containers*. O isolamento apresenta a vantagem de poder executar vários *containers* ao mesmo tempo em um mesmo *host*.

É possível instalar todos os elementos de uma aplicação dentro de um *container*. Assim, não é necessário instalar nada no *host*. Ainda os *containers* podem ser usados para compartilhar uma aplicação de

maneira rápida e fácil com outros usuários. Utilizou-se o docker como ferramenta neste trabalho, sobretudo por considerar que tem um alta capacidade de portabilidade, garantindo fácil instalação do *software* de extração de *features* proposto pelo usuário em diferentes máquinas.

O Docker é composto por uma arquitetura cliente e servidor, em que um Docker *client* fala com um Docker *daemon* utilizando uma *Application Programming Interface* (API), encima dos *sockets* do UNIX ou de uma interface de rede. Um outro Docker *client* importante de ser mencionado é o Docker *compose*, que permite trabalhar com aplicações compostas por vários *containers*. Isso se enquadra no caso do *software* desenvolvido, que necessita de vários *containers* interagindo entre si.

Outro elemento importante a ser mencionado na arquitetura são as *images*, que são os *templates* que contém as instruções necessárias para criar o *container*. As imagens Docker são guardadas em Docker *registries*, como o Docker *Hub*, que contém um banco de Docker *images* público. A Figura 2.5 ilustra a arquitetura do Docker.

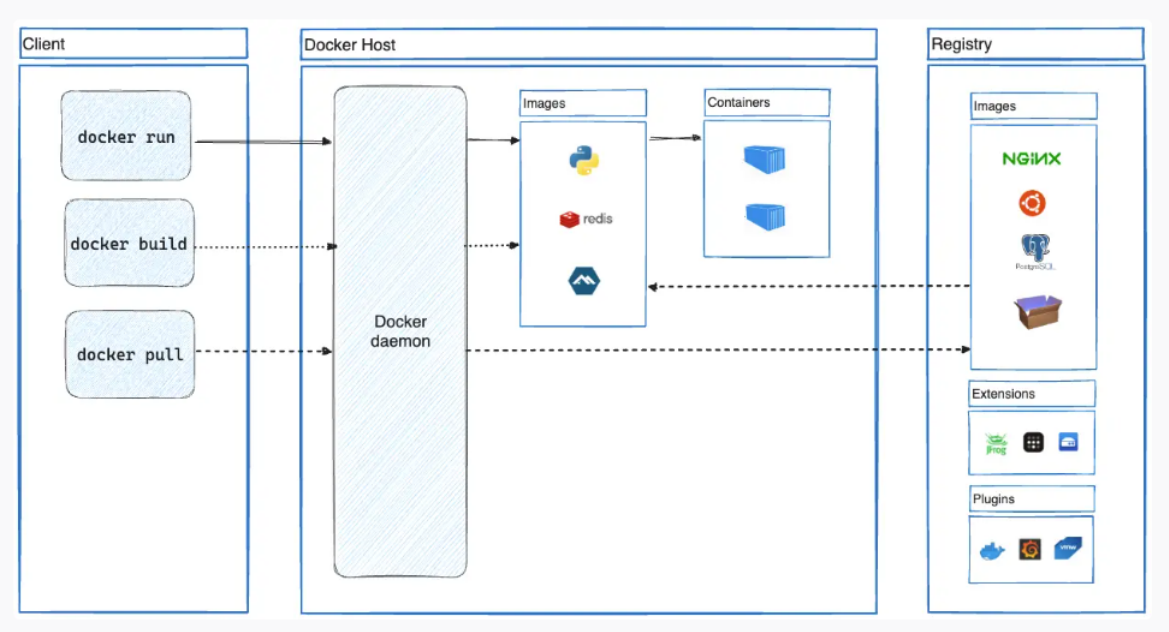

Figura 2.5: Visão geral da arquitetura do Docker retirada de [5]

2.1.2.5 Python

Python é uma linguagem de programação amplamente reconhecida por ser extremamente versátil. A linguagem conta com uma vasta quantidade de bibliotecas. Possuindo bibliotecas para extração de tráfego da rede (como o Pyshark) , bibliotecas para interagirem com banco de dados relacionais e não relacionais e também para rodar algoritmos de ML.

A seguir são listadas as principais bibliotecas utilizadas no trabalho, bem como suas principais aplicações:

- Pyshark: É um *wrapper* que permite extrair informações de um arquivo pcap ou de uma interface de rede.

- PyMongo: É a biblioteca utilizada para trabalhar junto com o MongoDB permitindo criar e alterar instâncias no banco de dados.
- Redis: É a interface Python para o armazenamento de valores-chave do Redis.
- Sklearn: Biblioteca amplamente utilizada para treinar modelos de ML em Python.
- Optuna: Biblioteca em Python para otimização de hiperparâmetros de algoritmos de ML.
- Pydialog: Bilbioteca que permite criar janelas de mensagem no Dialog.

Vale ressaltar que as bibliotecas são feitas para interagir com as aplicações em python, não para as substituir. Sendo assim, necessária a instalação das aplicações em uma máquina ou por meio da utilização de *containers*.

2.1.3 Modelos de aprendizado de máquina

O uso de aprendizado de máquina ocorre para resolver tarefas que geralmente não têm um algoritmo específico para solucioná-las, mesmo com vários estudos relacionados ao tema.

O processo de aprendizagem se dá em ajustar parâmetros de um modelo para se encaixarem em um conjunto de dados visto durante o treinamento. Com isso, é possível resolver uma tarefa específica fundamentada em um conjunto de dados de entrada. O modelo treinado transforma-se, então, em um algoritmo para solução de uma tarefa específica como explica [27].

Os modelos de aprendizado podem ser supervisionados ou não supervisionados. No caso de um modelo supervisionado, o objetivo é mapear os parâmetros de entrada para uma saída considerando que os valores corretos estão sendo fornecidos por um supervisor. Já no caso de um modelo não supervisionado, esse supervisor não existe, há somente os dados de entrada. Esse método pode ser usado para verificar padrões nos conjuntos de dados de entrada para, por exemplo, formar *clusters* [27].

No presente trabalho, o interesse é criar um algoritmo, utilizando-se de aprendizado de máquina, para classificar fluxos na rede entre normais ou anômalos. Para tanto, serão utilizados modelos de aprendizagem supervisionado, que contêm os rótulos corretos das classes, acompanhados das amostras de entrada.

Ao analisar trabalhos anteriores, verificou-se que entre os modelos supervisionados mais utilizados para classificação no *dataset* NSL-KDD estão os modelos relacionados a árvores de decisões como mostrado no *survey* de [19] e no *review* de [1].

Entre os motivos a serem elencados para se utilizar tais algoritmos está o fato de serem facilmente interpretados, sua capacidade em capturar relações não lineares entre variáveis e sua resiliência a *outliers*, que, a concluir dos resultados obtidos nos trabalhos analisados na literatura, são fatores que contribuem positivamente na detecção de anomalias em IDS.

Foram escolhidos 3 modelos baseados em DT para treinamento. Cita-se que, nesses 3 modelos elegeu-se um que usa o método *bagging* (RF), um que usa *boosting* (adaBoost) e outro que usa o próprio modelo de DT.

Usou-se diferentes modelos para comparar o desempenho dos algoritmos treinados levando em conta o fato dos modelos de *bagging* e *boosting* buscarem reduzir o *overfitting* que pode ocorrer em modelos treinados com DT. Ressalta-se que, foram utilizadas técnicas de poda como forma de reduzir o *overfitting* do algoritmo treinado com DT.

2.1.3.1 *Decision Tree* (DT)

Uma árvore de decisão é um modelo hierárquico para aprendizado supervisionado. Uma região local de decisão neste modelo é definida a partir de uma sequência de divisões anteriores. Uma árvore de decisão pode ser usada para classificação (como, por exemplo, classificar entre tráfego de rede normal ou malicioso), sendo chamada de árvore de classificação, ou para regressão, sendo chamada de árvore de regressão (para prever valores numéricos como, por exemplo, o preço de uma casa).

Cada nó de decisão n implementa uma função $f_n(x)$ que vai decidir para qual ramo a amostra vetorial x deve ser encaminhada. A função de cada nó,$f_n(x)$, é aplicada do nó raiz n até atingir um nó chamado folha, sendo que o valor apontado pela folha é a saída, que no caso da classificação é a classe que a amostra x foi classificada. A Figura 2.6 mostra um exemplo simples de árvore de classificação a partir dos conceitos expostos.

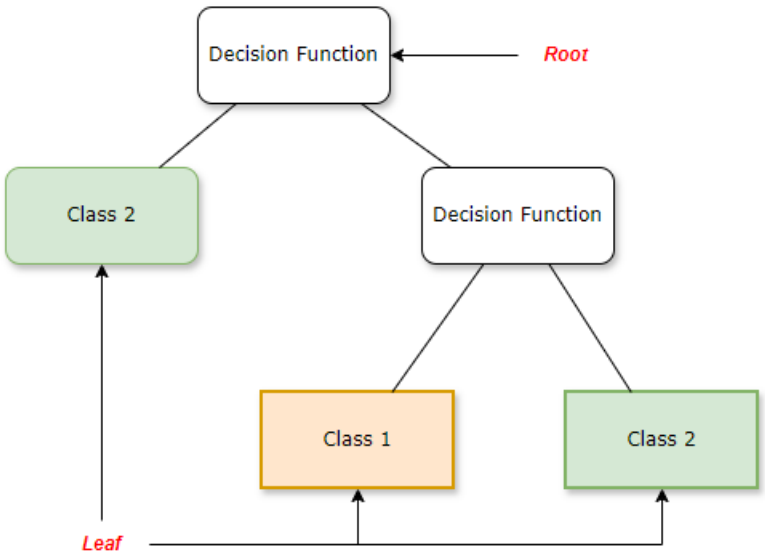

Figura 2.6: Um exemplo de árvore de classificação.

No caso de uma árvore univariada, cada nó interno usa apenas uma das dimensões de entrada de um vetor v para gerar uma função no nó n, $f_n(v)$. Cada função é como um discriminador local.

Para o caso de uma árvore que faz classificação a qualidade de uma divisão é quantificada a partir do grau de impureza. Uma divisão é considerada pura, ou seja, tem grau de impureza quantificado em 0, se todas as instâncias (do treino) após a divisão forem da mesma classe.

Inspirando-se nas definições de [27], se considerarmos I_n a quantidade de instâncias I que estão no nó n, C_i a classe com rótulo i, e I_n^i como a quantidade de instâncias pertencentes a classe i no nó n, pode-se dizer que um nó será puro se tiver valor 1 na Equação 2.2 para uma classe i.

Exemplificando a Equação 2.2, se um nó n tem como probabilidade de encontrar uma amostra da classe a como 1, então $p_n^a = 1$, enquanto que para as classes diferentes de a tal probabilidade será igual a 0 ($i \neq a \Rightarrow p_n^i = 0$), sendo assim um nó puro.

$$P(C_i|x, n) = p_n^i = \frac{I_n^i}{I_n} \tag{2.2}$$

Geralmente, a fórmula que determina o grau de impureza de um nó é dado pela Equação 2.3, que é o cálculo de entropia. Também podem-se usar outros índices análogos como o índice de Gini.

$$\sum_{i=1}^{K} (p_n^i) log_2(p_n^i) \tag{2.3}$$

O objetivo de uma árvore de decisão é reduzir o máximo possível a impureza nos nós realizando o menor número de divisões possíveis [27]. Para isso, procura-se em cada divisão gerar uma função $f_n(v)$ que reduza o máximo a impureza.

Ainda, é importante citar métodos de poda (*pruning*) em DT, que são úteis no caso de haver poucas instâncias no *dataset* de treino, o que pode levar a uma alta variância, e assim, impedir a generalização de um modelo. Tais métodos podem ser utilizados depois que a árvore foi construída (*post-pruning*) ou antes (*pre-pruning*).

Como explicado em [28], as DTs podem acabar criando modelos complexos para os dados, o que resultará em overfitting. Portanto, é fundamental considerar técnicas de poda para evitar que isso ocorra.

Para obter mais informações relacionadas ao algoritmo em questão, recomenda-se a leitura de [27] e [28].

2.1.3.2 *Random Forest* (RF)

Uma *Random Forest* (RF) é um modelo que realiza predições a partir de um conjunto de árvores de decisões. Nesse modelo, cada árvore de decisão é construída a partir de um conjunto de amostras embaralhadas com reposição, método conhecido como *bagging*. Ainda é possível limitar cada árvore na RF a ter um subconjunto aleatório de *features*.

Em relação a classificação, a saída de uma instância v é determinada pela votação majoritária entre as árvores de decisão que compõem a floresta, ou seja, é a classe mais frequentemente predita para a instância v. A Figura 2.7 mostra um exemplo de RF.

Sabe-se que as árvores de decisões são modelos em que pequenas mudanças no *dataset* de treino resultam em grandes diferenças nas predições, sendo um algoritmo com alta variância. Segundo [29], o modelo RF é uma alternativa interessante para ajudar a diminuir a variância de um algoritmo, pois permite

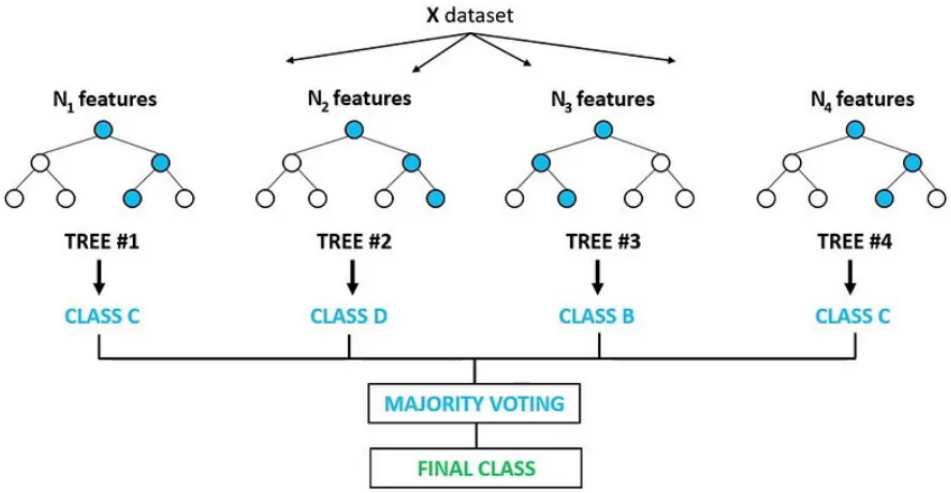

Figura 2.7: Um exemplo de RF de classificação retirado de [6]

a utilização de diferentes subconjuntos de *features* para realizar as divisões em cada árvore.

Entre os principais parâmetros a serem ajustados em uma RF, segundo [29], estão a quantidade de estimadores, isto é, quantidade de árvores de decisões e a cardinalidade do conjunto de *features* a serem usadas durante as divisões da árvore.

Em relação ao conjunto de estimadores, quanto mais estimadores, mais lento o modelo fica, sendo que a partir de um determinado número de estimadores a melhora não será mais significativa na performance do modelo [29]. Já em relação a cardinalidade do conjunto de features, ao diminuir o seu valor ocorre diminuição da variância, entretanto, isso também aumenta o viés do modelo [29].

2.1.3.3 AdaBoost

Esse algoritmo utiliza uma sequência de modelos fracos (modelos de baixa complexidade, como árvores de decisão de pouca profundidade) aplicando-os repetidamente em versões ponderadas do conjunto de treino (isto ocorre de forma que as amostras de treino mais dificilmente classificadas recebem maior peso). As predições de cada modelo são então combinadas, sendo que cada modelo recebe um peso diferente de acordo com a quantidade de acertos que tem no *dataset*, gerando a predição final.

A modificação nos dados em cada iteração do algoritmo de *boosting* ocorre ao se utilizar pesos para cada uma das amostras do *dataset* utilizado. Na primeira iteração do algoritmo, os pesos das amostras são iguais, sendo que o modelo fraco, que resulta em mais acertos nos dados da primeira iteração, é usado para treinar o *dataset* original.

Já nas iterações seguintes, os pesos das amostras classificadas erroneamente nas iterações anteriores serão maiores do que os das classificadas corretamente, fazendo com que os modelos fracos gerados nas próximas iterações levem mais em conta a influência de amostras que foram classificadas incorretamente por modelos das iterações anteriores.

Ainda vale ressaltar que os modelos fracos gerados em cada uma das iterações terão uma influência

diferente na votação para predição final de acordo com a sua capacidade de acerto no conjunto de dados de sua iteração.

A Figura 2.8 de [7] tem o intuito de mostrar de forma mais clara o funcionamento do algoritmo. Nela é possível ver um conjunto de amostras S em que a i-ésima amostra é dada por (x_i) e i-ésima label por (y_i).

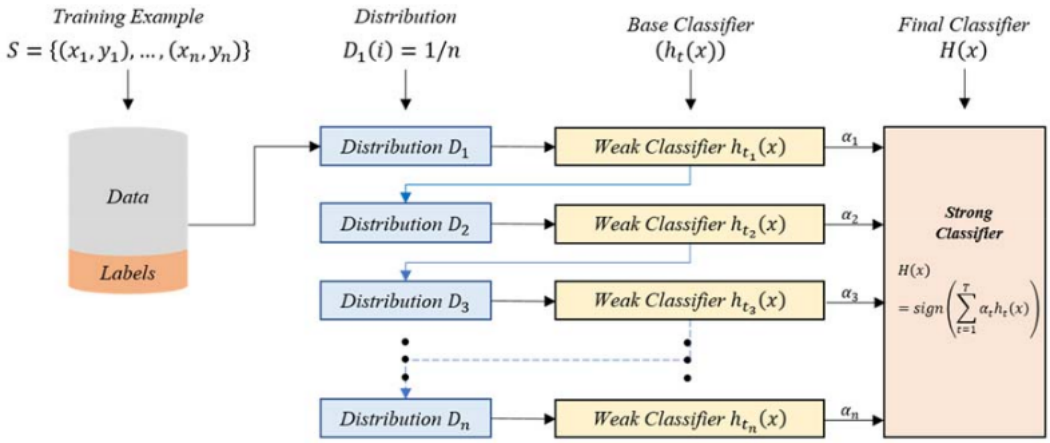

Figura 2.8: Visualização do algoritmo adaBoost retirado de [7]

Ainda cada amostra n em cada iteração $(iter)$ recebe um peso a partir da distribuição $D_{iter}(i)$, (sendo $D_1 = \frac{1}{n}$, ou seja, as amostras começam com mesmo peso) e assim é treinada com um classificador $h_{t_{iter}}(x)$. Ao final de cada iteração os pesos D_{iter} são atualizados e é formado outro classificador fraco.

No fim das iterações é formado um classificador $H(x)$ formado pelo conjunto de classificadores fracos definidos em cada iteração.

2.1.4 Estratégias de experimentação no NSL-KDD

Um experimento consiste em alterar os fatores da entrada de um modelo para verificar o resultado na saída do modelo. Entre alguns dos fatores possíveis de serem alterados na entrada estão o modelo do algoritmo de ML usado, o conjunto de dados de treino e o conjunto de *features*.

A experimentação destes fatores visa encontrar a combinação que produz o melhor resultado de determinada métrica de performance (sendo que no presente trabalho tal métrica é a acurácia no conjunto de validação). No caso do NSL-KDD, partiu-se do pressuposto que encontrar a melhor acurácia no conjunto de validação resultaria na melhor acurácia no arquivo de teste do NSL-KDD.

Reitera-se ainda que, o NSL-KDD é um *dataset* feito com o objetivo de ser mais generalista que seu antecessor, KDD99, isto principalmente por conta dos motivos explicitados a seguir.

O primeiro motivo é a amostragem feita no *dataset* de treino e teste do KDD-CUP 99 para gerar o NSL-KDD, que seleciona proporcionalmente mais as amostras no KDD-CUP 99 que são mais difíceis de serem classificadas corretamente ao longo dos 21 modelos de ML treinados em [21] (21 modelos, por conta dos 3 subconjuntos de dados de treino avaliados em 7 algoritmos de ML distintos). O segundo motivo, é o fato de retirar as amostras redundantes que acabavam por impor um viés durante o treinamento do modelo.

Ainda, um ponto positivo herdado do KDD 99, para previsão de anomalias, é o fato do conjunto de ataques de treino ser diferente do conjunto de teste. Isso exige que o modelo tenha uma capacidade significativa em detectar ataques não vistos a fim de ter uma boa acurácia.

Apesar de ser um *dataset* com muitas melhorias em relação ao seu antecessor, muitos trabalhos ainda buscam técnicas, a partir da alteração de fatores, para reduzir o ajuste excessivo ao conjunto de dados de treino (conhecido como *overfitting*), utilizando-se, sobretudo, de algoritmos para extração de *features*, como mostrado no conjunto de trabalhos contidos no *survey* [19].

A seguir são analisadas algumas estratégias que podem ser usadas para se evitar *overfitting*, e com isso, obter um modelo melhor na predição para um conjunto de dados de teste, isto é, uma base da qual não se tem nenhuma informação prévia.

2.1.4.1 Acréscimo de dados

É possível que acrescentar dados a um *dataset* torne o algoritmo de ML mais generalista, e assim, melhore o seu desempenho em um conjunto de dados não visto (conjunto de teste, e na detecção de anomalias de rede).

Como forma de acrescentar dados, pesquisou-se *datasets* que poderiam extrair *features* semelhantes ao do NSL-KDD e, encontrou-se o Kyoto 2006+ *dataset* que contém 14 *features* iguais ao do NSL-KDD. Outra forma de realizar o acréscimo de dados seria usando o *software* proposto para extrair tráfego de uma interface de rede em tempo real ou de um arquivo PCAP.

2.1.4.2 Extração de *Features*

Como forma de melhorar a performance dos modelos de ML, é comum a utilização de algoritmos que selecionam somente as *features* mais relevantes em um *dataset*, descartando as julgadas irrelevantes.

Entre os algoritmos para extração de *features* utilizados em ML deu-se enfoque em dois. O *Information Gain* (IG), por ser um dos mais utilizados e apresentar resultados expressivos como mostrado por [19], e o *Moth Flame Optimization* (MFO), por ser o trabalho que se encontrou melhor resultado na literatura como mostrado por [8].

O algoritmo IG é utilizado para determinar a quantidade de informação que uma *feature* trás (influencia) no rótulo de classificação, fazendo isso a partir da medida de entropia, como explica [19].

O algoritmo MFO é apresentado no trabalho de [8] como um algoritmo de *feature extraction* que influencia de forma bastante positiva na acurácia do *dataset* de teste do NSL-KDD. Como é um algoritmo menos utilizado na literatura e com implementação menos disseminada, deu-se alguns detalhes a mais sobre o seu funcionamento. Ressalta-se que uma de suas características positivas é tentar encontrar soluções globais ótimas, escapando de soluções locais.

O algoritmo MFO pode se utilizar de várias iterações para encontrar as melhores *features*, como mostrado na Figura 2.9. Entre os elementos mais importantes neste algoritmo estão:

Algorithm 1 Steps of the MFO algorithm for the present work.

Input: Complete feature set of NSL-KDD dataset, $max\_iteration$, n (number of moths) & d (number of dimensions)
Output: The relevant features with their respective fitness values

1. Initializes the position of the moths and flames randomly
2. **for** i in 1 to $max\_iteration$ **do**
3. $Flame\_no \leftarrow round[(N-1)*(N-1)/max\_iteration]$
4. $OM \leftarrow fitnessFunction(M)$
5. **if** i=1 **then**
6. $F \leftarrow sort(M)$
7. $OD \leftarrow sort(OM)$
8. **else**
9. $F \leftarrow sort(M_{t-1}, m_t)$
10. $OF \leftarrow sort(M_{t-1}, m_t)$
11. **end if**
12. **for** i in 1 to n **do**
13. **for** j in 1 to d **do**
14. $[r, t] \leftarrow update(r, t)$
15. $D_i \leftarrow |F_j - M_j|$ ▷ D_i is the calculated distance between ith moth and jth flame
16. $S(M_i, F_j) \leftarrow D_i e^{bt} \cos(2\Pi t) + F_j$
17. $M_i \leftarrow S(M_i, F_j)$
18. **end for**
19. **end for**
20. **end for**
21. End

Figura 2.9: Pseudocódigo para o algoritmo MFO [8]

- Uma matriz aleatória chamada *Moth* ($M_{n,d}$), em que cada linha n representa uma *Moth*, que contém em cada coluna o valor posicional (índices) de uma das d *features* a ser selecionada no *dataset*, sendo que esse valor posicional pode repetir (o que acaba por reduzir a quantidade de *features*).
- A função $fitnessFunction(M)$, que recebe a matriz *Moth* e realiza o treino de um algoritmo de ML para o subconjunto de *features* de cada linha da matriz M no *dataset* de treino, e retorna as acurácias para o conjunto de validação, o que resulta em uma matriz OM contendo n acurácias.
- Uma matriz de *flames*, F, que contém as melhores *Moths*, isto é, os subconjuntos de *features* com os melhores resultados de acurácia de validação. Assim, o subconjunto de *features* ótimas na iteração t será ordenando as matrizes $M_t, M_{t-1}, M_{t-2}, ..., M_1$ por ordem de acurácia.
- Um loop responsável por atualizar o valor da matriz M a partir da matriz F.
- Relata-se que, na primeira iteração do algoritmo só vai haver n amostras de *Moth* produzidas, mas nas seguintes esse número aumenta sendo necessário guardar o registro das acurácias obtidas da matriz M da iteração imediatamente anterior para produzir a matriz F na iteração atual.

A Figura 2.9 mostra os detalhes do algoritmo MFO, composta pelos elementos brevemente explicados. A implementação usada no presente trabalho foi obtida a partir da adaptação do código fonte obtido em [30].

2.1.4.3 Regulação de hiperparâmetros nos modelos

Regular os parâmetros de entrada é essencial para otimizar o desempenho do modelo na saída. Pode-se fazer isso alterando-se a quantidade de dados e com seleção de *features* como já explicado, mas também pode-se realizar experimentações focadas no modelo utilizado.

Existem modelos de ML que são usados para resolver problemas mais complexos como é o caso das DL, que são utilizadas principalmente em problemas relacionados a visão computacional (classificação de imagens e detecção de objetos) e modelos como *Logistic Regression* (LR), que são utilizados para resolver problemas mais simples, como os relacionados a previsão e análise de relações entre variáveis (o preço de uma venda baseado em seu histórico, por exemplo). Pode ser que usar um modelo complexo para uma tarefa simples gere *overfitting*, ou seja, o sobre ajuste nos dados de treino, portanto, é fundamental regular o modelo nesse sentido.

É possível também alterar os parâmetros de entrada dentro do próprio modelo, os chamados hiperparâmetros, que regulam a complexidade de um modelo. Como exemplo, em uma DT, pode-se escolher o limite de profundidade da árvore de decisão, sendo que menos camadas resultarão em um modelo mais simples.

Considera-se para o problema de classificação em questão que os melhores hiperparâmetros são aqueles que geram a melhor acurácia de validação. Considera-se que é uma métrica passível de ser utilizada, dado que no caso da classificação binária no NSL-KDD a quantidade de amostras normais e anômalas não é tão distinta (53% e 47% aproximadamente) como no caso da classificação multiclasse (isto é, a classificação entre normal com 53% das amostras, Dos com 36%, U2R com 0,04% , R2L com 0,8% e Probe com 9,3 % aproximadamente). Cita-se isso porque a acurácia é sensível a grandes desbalanceamentos de classe acabando por mascarar os erros de classes minoritárias.

Para encontrar os melhores hiperparâmetros são feitas várias experimentações com seus valores variando. Vale ressaltar que a acurácia não é o único critério de desempenho, podendo-se utilizar outras métricas como como precisão, recall e F1-score a depender da natureza do problema e do *dataset* utilizado.

Uma das possíveis formas de se modificar os hiperparâmetros é através da alteração de somente um fator e manter os outros fixos. Esse método não é considerado ideal, pois supõe que os fatores não tem interação entre si o que nem sempre é verdade segundo [27]. A abordagem mais correta para alterar hiperparâmetros segundo [27] é variar os fatores ao mesmo tempo ao invés de um por vez. Tal metodologia é chamada de forma popular de *grid search*.

Uma ferramenta amplamente utilizada para regular hiperparâmetros é o Optuna, que é um *framework* desenvolvido por [9], para otimizar o processo para se encontrar o conjunto ótimo de hiperparâmetros. A recomendação para se utilizar esse *software* é criar um estudo, que consiste de várias *trials* (tentativas).

Em cada *trial* um conjunto de hiperparâmetros, cujo intervalo de valores é escolhido pelo usuário, é fornecido na entrada e então é configurado no modelo de ML. A saída é uma métrica de avaliação de algoritmos de ML, como a acurácia.

No código de exemplo mostrado na Figura 2.10, confeccionado pelos próprios desenvolvedores da

```python
import sklearn

import optuna

# 1. Define an objective function to be maximized.
def objective(trial):

    # 2. Suggest values for the hyperparameters using a trial object.
    classifier_name = trial.suggest_categorical('classifier', ['SVC', 'RandomForest'])
    if classifier_name == 'SVC':
         svc_c = trial.suggest_float('svc_c', 1e-10, 1e10, log=True)
         classifier_obj = sklearn.svm.SVC(C=svc_c, gamma='auto')
    else:
        rf_max_depth = trial.suggest_int('rf_max_depth', 2, 32, log=True)
        classifier_obj = sklearn.ensemble.RandomForestClassifier(max_depth=rf_max_depth, n_estimators=10)
    ...
    return accuracy

# 3. Create a study object and optimize the objective function.
study = optuna.create_study(direction='maximize')
study.optimize(objective, n_trials=100)
```

Figura 2.10: Exemplo de estudo no optuna retirado de [9]

ferramenta, é feito um estudo para otimizar os hiperparâmetros dos modelos *Support Vector Machine* (SVM) e RF implementados na biblioteca *sklearn* do python.

No código são fornecidos pelo usuário os hiperparâmetros svc_c (é o parâmetro C, que controla a tolerância a erros de classificação no modelo treinado) e rf_max_depth (a profundidade máxima das árvores na RF) a serem otimizados em cada modelo e o intervalo de busca de cada hiperparâmetro ($e^{-10} >= svc\_c <= e^{10}$ e $2 >= rf\_max\_depth =< 32$), que o algoritmo do Optuna utilizará para encontrar a melhor solução. Ainda, a métrica escolhida para ser otimizada é a acurácia máxima, sendo que o conjunto de hiperparâmetros escolhidos serão aqueles da *trial* que obteve maior acurácia nos modelos. Para consultar mais detalhes sobre o algoritmo do Optuna na otimização dos hiperparâmetros recomenda-se consultar [9].

Ao final de n trials são escolhidos os hiperparâmetros que trouxeram os melhores resultados para a métrica em análise.

2.1.4.4 Técnicas para medir performance

Para medir o quão consistente é a performance para uma configuração de fatores como: modelo com seus hiperparâmetros e *features*, um experimento pode ser feito se utilizando dos mesmos fatores em amostras diferentes dos dados. Tal método é conhecido como *cross-validation*.

Neste trabalho utiliza-se o *K-fold cross validation*. Nesse método o *dataset* D é dividido em K partições do mesmo tamanho D_i, i = 1, ..., K. São feitos K experimentos, sendo que em cada experimento seleciona-se uma das partes D_i como conjunto de validação e as K-1 partes restantes são escolhidas como conjunto de treino.

Assim, todas as partes D_i acabam por atuar como conjunto de validação em um dos K experimentos

feitos. Isso permite medir a performance média de um modelo a partir de K experimentos.

Um ponto interessante no *K-fold cross-validation* é o fato de não ser necessário dividir um *dataset* entre um conjunto de treino e validação. Isso porque o conjunto de validação será composto pelas diferentes partes do conjunto de treino em cada experimentação do *K-fold*. Tal fato pode ser muito útil no caso de um *dataset* com poucas amostras em que ao se retirar um conjunto de dados para validação pode resultar perda preciosa de informação para o treinamento [10].

É interessante ressaltar que para um conjunto de dados com poucas amostras, um número de K pequeno pode significar perder muita informação para um conjunto de validação, o que pode gerar um *underfitting* do modelo, ou seja, um modelo que não consegue ser preciso por conta de poucas amostras que foram dadas no momento do seu treinamento. Assim, é preciso avaliar o tamanho do *dataset* para escolher um K que não comprometa a etapa de treino do modelo, como explica [27].

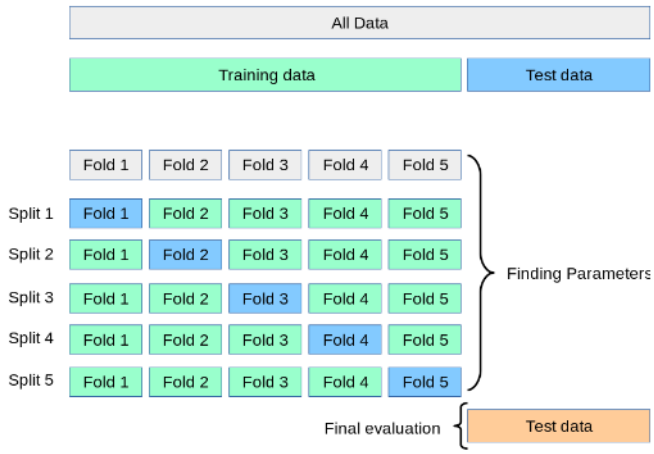

Figura 2.11: Exemplo de um *5-fold cross-validation* retirado de [10]

Na Figura 2.11 retirada de [10] é mostrado um exemplo de um *5-fold cross-validation*, em que o *dataset* é dividido em 5 partes e são realizados 5 experimentos.

Em resumo, é proveitoso utilizar o *K-fold cross-validation* para obter os melhores hiperparâmetros de forma mais segura, já que se obtêm uma média de performance deles em K modelos.

Para obter os melhores hiperparâmetros, é estratégico rodar o algoritmo *K-fold* n vezes até encontrar os fatores considerados consistentes pelo experimentador que então serão usados em um modelo para avaliação de um conjunto de teste.

2.1.4.5 Métricas para medir performance

Entre um dos aspectos mais importantes em um algoritmo de ML é saber o quão bem está performando. Para tanto, são utilizadas métricas de classificação, que avaliam a eficácia do modelo em identificar corretamente as classes.

Para chegar nos valores dessas métricas, que são medidas percentuais, é necessário antes definir os elementos da matriz de confusão, na Tabela 2.5, que é composta pelos seguintes elementos: verdadeiros

positivos (TP), falsos positivos (FP), verdadeiros negativos (TN) e falsos negativos (FN), sendo as classes da matriz de confusão adaptadas para o problema de IDS, que tem ataques com valor positivo e tráfego normal com valor negativo.

Tabela 2.5: Definindo a matriz de confusão no contexto do trabalho

True Label / Predicted	Attack (Pos.)	Normal (Neg.)
Attack (Pos.)	TP	FN
Normal (Neg.)	FP	TN

- TP: Quantidade de amostras da classe positivas (ataques) que foram classificadas como positivas (ataques).
- FN: Quantidade de amostras da classe positiva (ataques) que foram (erroneamente) classificadas como amostras da classe negativa (normais).
- TN: Quantidade de amostras negativas (normais) que foram classificadas como amostras negativas (normais).
- FP: Quantidade de amostras negativas (normais) que foram (erroneamente) classificadas como amostras positivas (ataques).

Tendo definido os elementos da matriz de confusão é possível extrair estatísticas a partir delas. Coloca-se as definições das seguintes métricas cujas fórmulas estão definidas logo em sequência:

- *True Positive Rate* (TPR): Também conhecida como *sensitivity* ou *recall* ou *detection rate* (DR) esta métrica calcula o percentual de amostras positivas (ataques) corretamente classificadas.
- *True Negative Rate* (TNR): Também conhecida como *specificity* calcula o percentual de amostras negativas (normais) corretamente classificadas.
- *False Negative Rate* (FNR) : Calcula o percentual de amostras positivas (ataques) erroneamente classificadas como negativas (normais). Pode ser obtida fazendo 1-TPR.
- *False Positive Rate* (FPR) : Calcula o percentual de amostras negativas (normais) erroneamente classificadas como positivas (ataques). Pode ser obtida fazendo 1-TNR.
- *Precision* (Precisão): Calcula o percentual de amostras corretamente classificadas como positivas (ataques) entre o total de amostras classificadas como positivas (ataques).
- *Accuracy* (Acurácia): Calcula o percentual de acerto para as classes positiva (ataque) e negativa (normal) entre todas as amostras.
- F1-score: É a média harmônica entre *precision* e *recall* (TPR).

$$TPR = Recall = \frac{TP}{TP + FN} \in [0, 1], \qquad FNR = \frac{FN}{TP + FN} \in [0, 1],$$

$$TNR = \frac{TN}{TN + FP} \in [0, 1], \qquad FPR = \frac{FP}{TN + FP} \in [0, 1]$$

$$Precision = \frac{TP}{TP + FP} \in [0, 1], \qquad f1\_score = \frac{2 * Precision * Recall}{Precision + Recall} \in [0, 1]$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \in [0, 1]$$

É interessante citar também a curva *Receiver Operating Characteristic* (ROC) como forma de medir a performance de um classificador binário. Ela leva em conta o fato da classificação ser feita com um grau de probabilidade, assim é possível definir um *threshold* (limiar) para classificar uma amostra pertencente a uma classe.

A curva ROC realiza a variação de *threshold*, começando com valores altos de *threshold* que são reduzidos gradativamente. Para cada valor de *threshold* é computado o valor das métricas TPR e FPR. A Figura 2.12 mostra o formato de uma curva ROC. Perceba que ela começa com valores baixos de TPR e FPR, que vão aumentando na medida em que se reduz o *threshold* usado para classificar uma amostra na classe positiva.

A medida usada para determinar a capacidade de classificação do modelo a partir desse gráfico é a *Area Under the Curve* (AUC), que varia de valores 0.5, no caso de um classificador aleatório (uma linha reta entre TPR e FPR), até 1 no caso de classificador perfeito (TPR=1 e FPR=0 para qualquer valor de *threshold*).

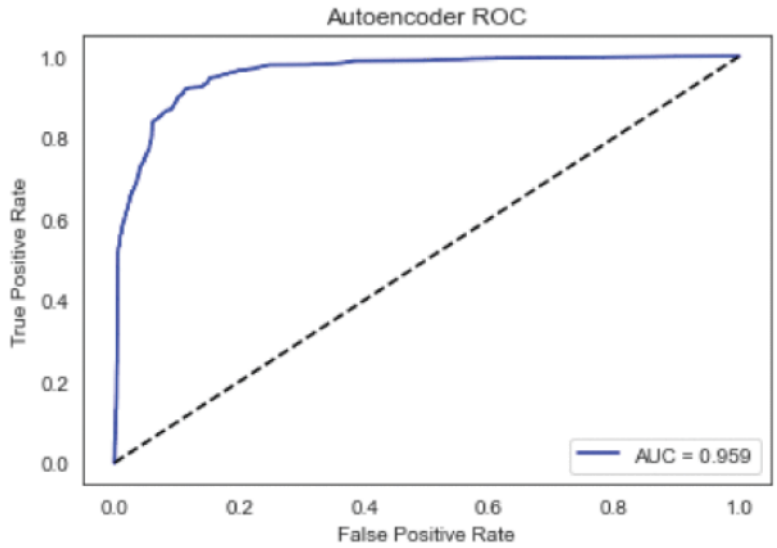

Figura 2.12: Exemplo de curva ROC retirado de [11]

2.1.5 Trabalhos Relacionados

O monitoramento constante do tráfego de rede é fundamental para identificar anomalias e é uma das boas práticas empregadas nas empresas atualmente. Nesse sentido, os Sistemas de Detecção de Intrusão (IDS) são ferramentas cruciais, juntamente com *firewalls*, para garantir a segurança das redes corporativas.

Nesta seção, são apresentados estudos que se dedicam à extração de *features* de *datasets*, analisando-os de forma a verificar suas contribuições e suas lacunas. Cita-se que o processo de extração, envolve a transformação dos dados brutos da rede em *features* a serem usadas em IDS que utilizam classificadores de ML, sendo fator crucial para manutenção de tais IDS como já discutido na seção introdutória.

Nos estudos encontrados, geralmente o processo de extração de *features* é acompanhado pelo uso de modelos treinados de ML para realizar a classificação do tráfego por um IDS. Ainda, os *datasets* utilizados para treino dos algoritmos nos trabalhos são aqueles já consolidados (*benchmarks*) na literatura, que são os citados por [1] (já mencionados em seções anteriores).

Muitos trabalhos medem o desempenho dos algoritmos de ML nos seus arquivos de testes ou nos de outros *benchmarks*, que possuem subconjuntos de features em comum, como é o caso dos trabalhos presentes no survey [19] e nos reviews [20] e [1].

Considera-se fundamental realizar a análise destes trabalhos para mapear os fatores mais eficazes utilizados, ou seja, as técnicas de *feature extraction*, experimentação e os modelos de ML utilizados. Espera-se que tal análise ajude na manutenção do desempenho de IDS que utilizam algoritmos de ML para classificar o tráfego, que é justamente um dos objetivos almejados no presente trabalho.

Ainda vale ressaltar que muitos dos trabalhos relacionados, bem como as conclusões obtidas ao analisá-los, são semelhantes às do artigo [15] , que foi desenvolvido simultaneamente a este tratado acadêmico, com menções explícitas ao artigo sempre que alguma de suas conclusões é utilizada.

2.1.5.1 IDS que utiliza DL

O trabalho de [12] propõe a arquitetura de IDS mostrada na na Figura 2.13, que consta com um módulo para extração de características em tempo real, que é uma máquina linux entre a LAN e o roteador *gateway*. O trabalho tem também um *pipeline* de ML, que recebe da máquina linux as *features* do *dataset* NSL-KDD via requisição HTTP, para obter uma predição de uma rede neural de 16 camadas implementada no Keras e pré-treinada com os arquivos de treino do NSL-KDD.

Os autores extraem 28 *features* do *dataset* NSL-KDD. As *features* relacionadas ao comportamento do usuário, nas aplicações não são computadas, alegando-se uma complexidade extra para sua extração. O sistema de captura foi feito usando a linguagem de programação C++ e o pacote LIBPCAP, uma biblioteca que tem uma API de alto nível para extração de tráfego na rede.

Foi feita uma função para analisar os pacotes capturados (isto é, os dados brutos vindos da rede) e, com isso, extrair as *features* do *dataset*, porém o trabalho não deixa claro como esta função foi feita. Tal observação foi feita por [15].

Ao tentar avaliar a performance dos algoritmos de ML (treinados a partir de um modelo de DL e de

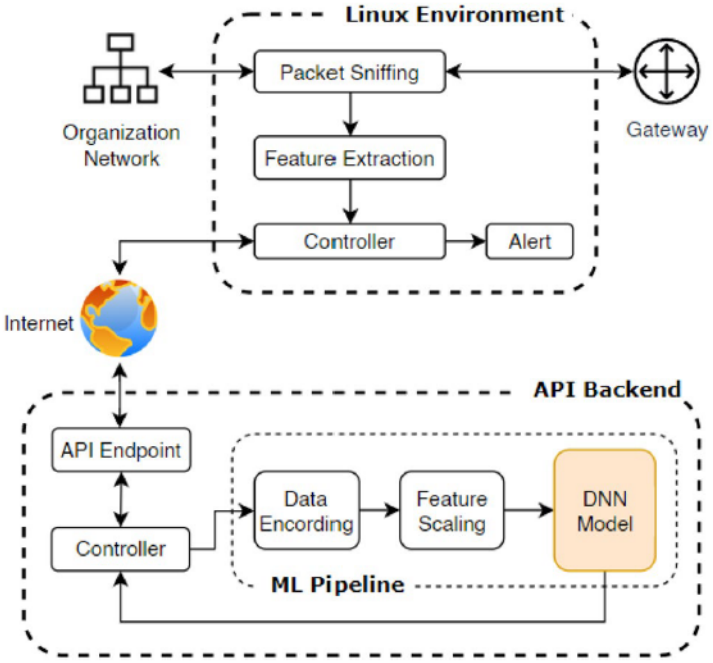

Figura 2.13: Arquitetura do IDS que utiliza DL retirada de [12].

outros 4 modelos de ML no arquivo de treino do NSL-KDD) no arquivo de testes do NSL-KDD foram obtidos os valores mostrados na Tabela 2.6.

Tabela 2.6: Avaliação do IDS que utiliza DL. Adaptada de [12].

Algoritmo	Acurácia	Precisão	Revocação	F1-score
KNN	0.7908	0.9584	0.6136	0.7481
SVM	0.7397	0.9643	0.5568	0.7059
OCSVM	0.7959	0.9600	0.5429	0.6935
K-Means	0.7328	0.9576	0.5369	0.6880
Proposed DNN	0.8187	0.9645	0.7071	0.8159

2.1.5.2 IDS baseado na nuvem

Os autores em [13] propõem uma aplicação hospedada na *Amazon Web Service* (AWS), que monitora o tráfego da rede e se integra a dois algoritmos de ML: um para classificação binomial (identificar entre normal ou ataque) e outro para classificação multinomial (identificando o tipo de ataque).

A aplicação interage com modelos de ML da biblioteca H20 por chamadas de API, sendo usados diferentes modelos de ML (RF, LR, *Naive Bayes* (NB) e SVM e duas arquiteturas de redes neurais para fins de comparação). No caso de um ataque ser identificado, o sistema desenvolvido gera um alerta, que é mostrado na interface web. A Figura 2.14 retrata a arquitetura do IDS.

Os algoritmos de ML foram treinados a partir dos modelos citados e dos dados de treino do NSL-KDD sendo que o processo de extração de *features* não foi detalhado. Somente é dito que os dados de entrada são convertidos de forma adequada para ser usada pelos algoritmos classificadores. Tal observação é a mesma

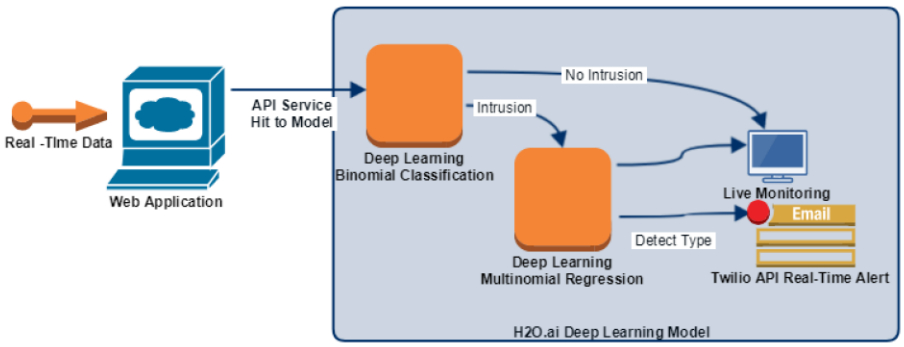

Figura 2.14: Arquitetura do IDS baseado na nuvem retirada de [13]

feita por [15].

Para avaliar os modelos de classificação binomial e multinomial, o trabalho utiliza o método de *cross-validation* e posteriormente o arquivo de teste do NSL-KDD. A performance dos modelos binomiais é mostrada na Tabela 2.7 enquanto dos multinomiais é mostrada na Tabela 2.8.

Tabela 2.7: Resultados para avaliação binomial. Adaptada de [13].

Avaliation	**Model**	**Accuracy [%]**	**Precision**	**Recall**	**F-measure**
Cross-Validation	Deep Learning H2O	99.52	99.95	100.00	99.55
	Deep Learning 4J	96.79	96.80	96.80	96.80
	Random Forest	99.91	99.90	99.90	99.90
	Logistic Regression	97.08	97.10	97.10	97.10
	Naive Bayes	90.34	90.40	90.40	90.30
NSL-KDD Test File	Deep Learning H2O	83.87	79.47	100.00	81.83
	Deep Learning 4J	76.12	80.20	76.10	76.00
	Random Forest	80.25	85.10	80.30	80.10
	Logistic Regression	74.58	79.80	74.60	74.30
	Naive Bayes	76.11	80.90	76.10	75.90
	LibSVM	72.36	81.60	72.40	71.50

Tabela 2.8: Resultados para avaliação multinomial. Adaptada de [13].

Avaliation	**Model**	**Accuracy [%]**	**Detection Rate [%]**			
			DoS	**Probe**	**R2L**	**U2R**
Cross-Validation	Deep Learning H2O	99.91	99.98	98.87	98.79	65.38
	Deep Learning 4J	98.77	99.04	98.23	95.38	44.23
	LibSVM	99.09	99.72	98.06	87.14	5.77
	Random Forest	99.97	99.99	99.97	99.50	90.38
	Logistic Regression	99.92	99.98	99.87	99.40	69.23
	Naive Bayes	93.66	96.43	87.08	42.71	90.38
NSL-KDD Test File	Deep Learning H2O	84.13	94.57	86.66	56.26	29.85
	Deep Learning 4J	71.36	84.66	67.20	41.07	44.78
	LibSVM	57.69	69.38	91.86	0.07	1.49
	Random Forest	76.92	88.35	88.76	37.75	62.69
	Logistic Regression	85.54	97.53	77.94	61.66	52.24
	Naive Bayes	71.11	75.23	90.91	43.95	67.16

2.1.5.3 IDS baseado em *Hadoop*

Em [14] é proposto um IDS dando principalmente atenção ao módulo de preprocessamento de *features*. O mesmo extraí 9 *features*, sendo 3 das *features* comuns ao *dataset* KDD-CUP 99.

O *pipeline* do IDS é descrito por meio do uso de pseudocódigos e fluxogramas, que detalham desde o processo utilizado para confeccionar conexões de rede a partir de um grupo de pacotes até a predição das conexões pelos algoritmos de ML.

A arquitetura do IDS contém 4 camadas, como mostrado na Figura 2.15. A primeira, é responsável por capturar os dados; a segunda, filtra os fluxos com assinatura e encaminha os que não tiveram assinatura encontrada para a camada seguinte (*Hadoop Layer*), que tem a função de construir os fluxos (conexões) de rede a partir dos pacotes e calcular as *features* dos fluxos usando um ecossistema *Hadoop* com *Apache Spark* e a última camada é responsável pela predição dos fluxos por algoritmos de ML pré-treinados com o *dataset* KDD-CUP 99.

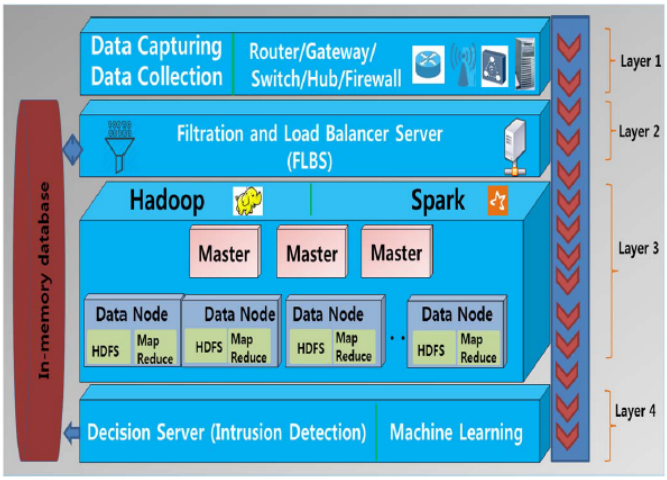

Figura 2.15: Arquitetura do IDS baseado em *Hadoop* retirada de [14]

Ressalta-se como ponto positivo deste trabalho o detalhamento do módulo de extração de *features*, dando detalhes sobre o algoritmo usado para a extração das *features*, usando-se de pseudocódigos e fluxogramas para tanto.

Não obstante, somente são implementadas as 9 *features* mostrada na Tabela 2.9. Dessas 9 *features* somente 3 (*Duration, Protocol, Service*) são comuns com as 41 do NSL-KDD, sendo que as 3 estão entre as mais simples de serem obtidas se comparadas com as *features* que extraem estatísticas de um conjunto de fluxos anteriores, como observa [15].

Tabela 2.9: *Features* extraídas do IDS baseado em Hadoop. Adaptada de [14].

Features	**Details**
Duration	Whole duration of the flow/session
protocol	Protocol
Service	Particular service the host is using
Num_root	Number of roots involved
No. of packets	No. of packets
Pkt_rate	Packet rate in packet/second for a particular flow
Pkt_size_mean	Mean value of the packet sizes
Pkt_sd_size	Pkt sizes standard deviation
Range_pkt_size	Range of the packet sizes

2.1.5.4 IDS que usa o modelo *bagging ensemble*

No trabalho de [8] é proposto um IDS, que utiliza o algoritmo MFO para selecionar as consideradas 23 melhores *features* entre as 41 do *dataset* NSL-KDD.

Essas *features* são utilizadas para treinar o modelo *bagging ensemble*, conseguindo resultados extremamente promissores para a classificação binária (Acurácia 87.44%) e multi-classe (Acurácia de 86.60%) no arquivo de teste do NSL-KDD ao se comparar com os outros trabalhos analisados na literatura (que são os analisados nesta seção).

Para realizar testes em tempo real, os autores criaram um módulo para extrair as 23 *features* (selecionadas pelo MFO) do *dataset* NSL-KDD. Assim, o trabalho relata que foi feito um extenso código para extrair dados da camada de rede com o intuito de computar tais *features*.

Como etapa inicial no cálculo das *features* foi utilizado o pyshark [31] para capturar tráfego em tempo real de um arquivo PCAP. Em seguida é feito um processamento para extrair as *features* do NSL-KDD a partir deste arquivo PCAP e salvá-las em um arquivo CSV. Por fim, os dados salvos no arquivo CSV serão utilizados para inferência pelo algoritmo *bagging emsemble*, pré-treinado com o *dataset* NSL-KDD. Tais etapas são mostradas na Figura 2.16.

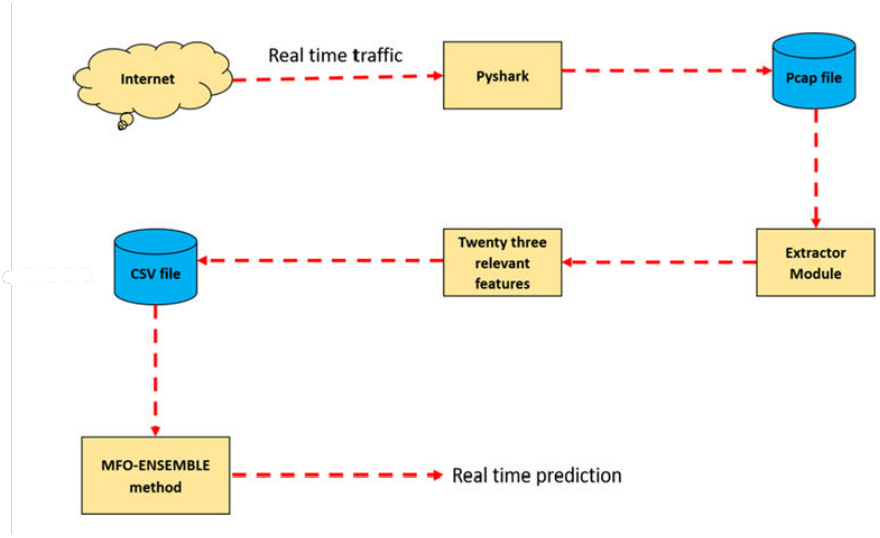

Figura 2.16: Arquitetura do IDS que usa o modelo *bagging ensemble* retirada de [8].

Após analisar o trabalho, foi possível perceber que o mesmo apresenta imagens que descrevem os passos para processamento das *features*, também apresenta as ferramentas utilizadas no processo. Não obstante, carece em informar os detalhes sobre a forma que a arquitetura é utilizada para confeccionar as *features* comparado com [14], como explica [15].

2.1.5.5 IDS utilizando tráfego IoT

No trabalho de [4] é feito um IDS que tem como principal motivação a descoberta de ataques em tráfego IoT. Considera-se a análise deste trabalho de suma importância dado o crescimento da utilização de dispositivos IoT no setor industrial e mesmo por usuários comuns.

Existem alguns trabalhos que realizam a análise de tráfego IoT, como [32], que propõe um *framework* eficiente para detecção de ataques Dos em redes IoT, e o trabalho de [33], que busca detectar ataques *botnets* em ambientes IoT. Entretanto [4] apresenta uma vantagem substancial na questão de geração dos dados por construir um *dataset* (que é o CIC-IoT *dataset* 2023 já mencionado na seção 2.1.1.3) com mais de 500 GB de tráfego IoT contando com 105 dispositivos na rede. Ainda apresenta uma grande quantidade de ataques que foram classificados em 7 categorias (DDoS, DoS, *Recon*, *Web-based*, *brute force*, *spoofing*, e *Mirai*).

Os autores ainda propõem uma ferramenta para extração de suas *features*, sendo elas em sua maioria diferentes do NSL-KDD. Para extração, os autores têm a ideia de separar os arquivos PCAP em tamanhos fixos de até 10MB e com isso computam as *features* em paralelo para cada arquivo PCAP dando celeridade ao processo.

Após extrair as *features*, os autores realizam vários treinamentos para classificar o tráfego com diferentes algoritmos (LR, *Perceptron*, adaBoost, *Deep Neural Networks* (DNN) e RF).

São feitos classificadores de ML que buscam separar entre tráfego normal e malicioso e outros dois que buscam separar entre 8 classes (normal e 7 categorias de ataque) e 34 classes (normal e 33 ataques específicos).

Para treinar os modelos, as *features* calculadas de tráfego normal e malicioso são misturadas e então é feita uma divisão no *dataset* entre um conjunto de treino (80%) e teste (20%). As métricas utilizadas para avaliar o desempenho dos algoritmos de ML foram a acurácia, o *recall*, *precision* e o *f1-score*.

O *pipeline* para avaliação dos algoritmos de ML é ilustrado na Figura 2.17, enquanto que os resultados a partir dos classificadores é mostrado na Figura 2.18. Em relação a Figura 2.18 o único comentário a ser feito é a constância do algoritmo RF que apresenta as melhores métricas de desempenho de forma consistente nos diversos classificadores.

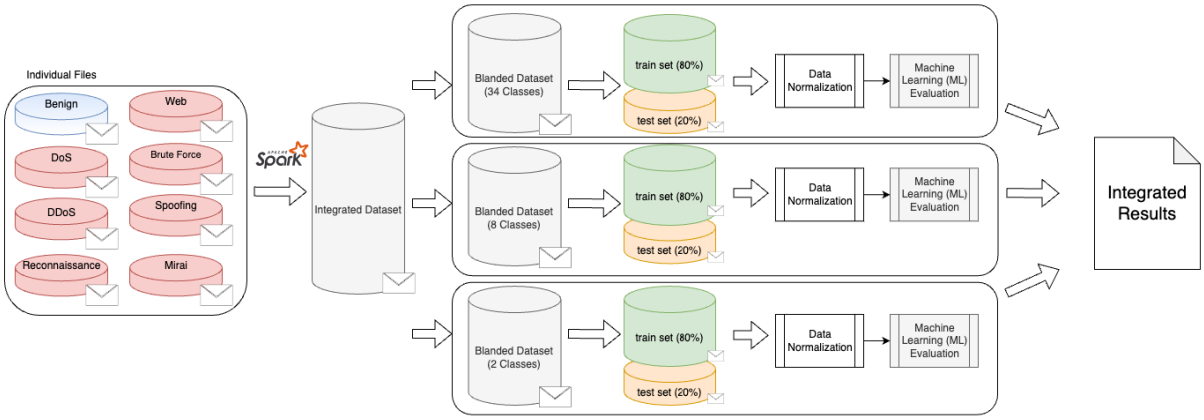

Figura 2.17: Pipeline usado para treinar e testar os algoritmos de ML no CIC-IoT *dataset* 2023 retirado de [4].

Em resumo, o trabalho é interessante principalmente pelo fato de gerar um *dataset* rotulado de mais de 500 GB. Os próprios autores ainda falam sobre a possibilidade de usar os arquivos PCAP como base para avaliar a eficácia de outras *features* geradas por outros pesquisadores, que foi exatamente o que foi feito no presente trabalho com as *features* do NSL-KDD.

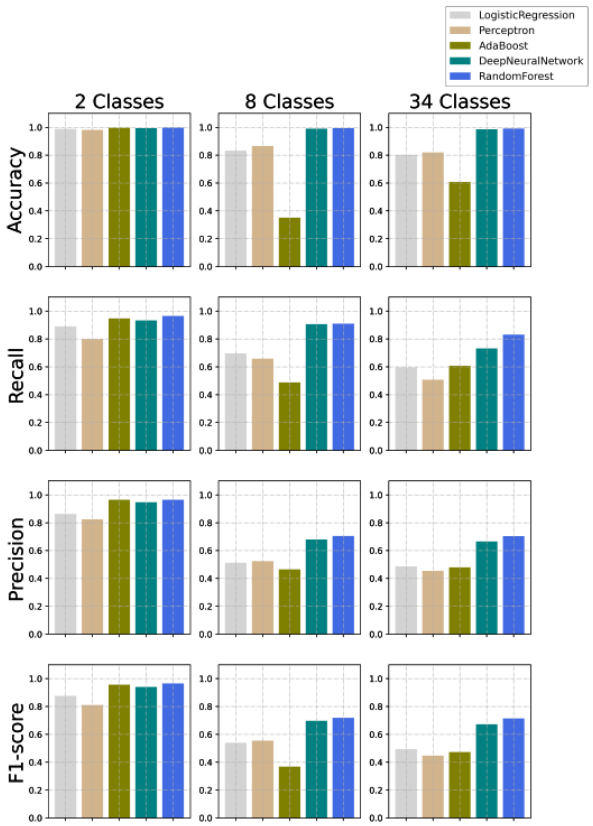

Figura 2.18: Resultados para a inferência nos fluxos do CIC-IoT *dataset* 2023 retirados de [4].

2.1.5.6 *Surveys* e *reviews* de algoritmos de ML em *datasets* de conexão de rede

Considerando a importância da literatura para escolher os melhores fatores a serem usados no treinamento de ML, foi feita a análise do *survey* [19] e do *review* [20], que trazem conclusões interessantes, sobretudo, a respeito do NSL-KDD.

Na maioria dos trabalhos contidos em [19] e [20] o foco não está tanto no processo de extração de *features*, mais sim no processo de seleção de fatores e em algoritmos de ML para avaliar desempenho em *datasets* amplamente utilizados na literatura.

São apresentados 35 trabalhos em [19] sendo que todos utilizam RF como modelo de classificação para detectar ataques em rede. Entre os motivos elencados para usá-lo são o fato de apresentar baixa complexidade no tempo de treinamento, predições rápidas, resiliência com *datasets* desbalanceados e por estar entre os melhores classificadores no NSL-KDD, como exposto por [34], que comparou as métricas para 10 classificadores diferentes concluindo que o RF apresenta as melhores métricas de *accuracy*, *sensitivity* e *specificity*.

Ainda entre estes trabalhos, 16 utilizam o *dataset* KDD-CUP 99 e 10 utilizam o *dataset* NSL-KDD para treinar os algoritmos de ML. Entre os trabalhos que utilizam o NSL-KDD 4 realizam o uso de IG, sendo o mais utilizado para extrair *features* neste *dataset*. Considerou-se este fator determinante para usar este método no presente trabalho.

Vale ressaltar ainda que as métricas utilizadas para medir desempenho nos algoritmos em [19] foram a

DR e FPR. Sendo que as taxas de DR foram sempre maiores que 95 % em todos os trabalhos e as de FPR menores que 2% na maioria dos casos.

Já [20] faz a análise de 20 trabalhos sendo que 14 utilizam o NSL-KDD para treinar seus algoritmos. No caso deste *review* são utilizados diversos algoritmos de ML. Sendo os principais utilizados: *K-Nearest Neighbors* (KNN), SVM, NB, RF e DT. A principal métrica de desempenho foi a acurácia, que para a maioria dos trabalhos no *review* esteve com valores superiores a 95%.

Um ponto importante a se ressaltar em relação aos trabalhos [19] e [20] é o *dataset* usado para medir desempenho, pois por vezes não é feita a diferenciação entre o conjunto de validação e teste.

Geralmente o conjunto de validação é retirado de uma parte do arquivo de treino (com treino sendo 75% do arquivo de treino e validação sendo 25% do arquivo de treino), ou obtido a partir da validação cruzada (*cross-validation*).

Observou-se nos trabalhos pesquisados [19] [20], que ao se medir a performance, são obtidos valores superiores a 90% nas métricas (acurácia ou DR) no caso de validação cruzada. Já no caso do arquivo de teste do NSL-KDD os melhores valores para classificadores binários foram encontrados nos trabalhos de [8] e [11] com 87% e 90% de acurácia, respectivamente.

Considera-se que, no caso do NSL-KDD, é o arquivo de teste que deve ser usado para medir performance nos algoritmos de ML, já que apresenta ataques não vistos no conjunto de treino, o que representa um cenário mais realista para detectar anomalias.

2.1.5.7 Lacunas a serem exploradas no processo de extração de *features* do NSL-KDD com base nos trabalhos vistos

Como observado por [15], os trabalhos de [12], [14] e [8], apresentam detalhes sobre as tecnologias utilizadas no módulo para confecção de *features*. Já em [13], notou-se a falta de um maior detalhamento.

Outro ponto notado por [15] ao avaliar [12], [13] e [8], foi a carência em detalhar o módulo responsável pela confecção de *features*. Somente no trabalho de [14] são fornecidos detalhes sobre tal processo, mas ele implementa apenas 9 *features*, das quais apenas 3 estão presentes nas 41 *features* do NSL-KDD.

Assim, os autores em [15] concluem que os trabalhos carecem de informações que facilitem o desenvolvimento de um módulo de processamento de *features* por parte de outros pesquisadores.

Mesmo que existam ferramentas amplamente utilizadas para extração de *features* como [35], [36] e [37] as mesmas não extraem as *features* do NSL-KDD de maneira automática como observa [15].

Levando em conta a complexidade das *features* do NSL-KDD e a sua utilização em diversos trabalhos na literatura, [15] avaliou-se a necessidade de um detalhamento de sua arquitetura, de forma a mapear suas tecnologias e a sequência de passos para extrair as *features* no *pipeline* proposto. A Tabela 2.10, reitera a lacuna encontrada e explica o que foi feito no trabalho para explorá-la.

Tabela 2.10: Exploração das lacunas relacionadas a extração de *features*

Lacunas	Exploração das lacunas no trabalho
Módulo para extração de *features* do NSL-KDD	Criação de um *software para extração* de 26 *features* do NSL-KDD, definindo claramente a arquitetura e metodologia empregada para se extrair as *features* a partir da arquitetura proposta.

2.1.5.8 Lacunas para manutenção de IDS com aprendizado de máquina usando o NSL-KDD

O fato de não existir um módulo para processamento de *features* do NSL-KDD impõe algumas lacunas relacionadas ao aprendizado de máquina no *dataset*.

Por exemplo, sem um módulo para extração de *features* não é possível aplicar o algoritmo de ML para classificar o tráfego em tempo real ou o tráfego de um arquivo PCAP, já que antes é necessário agrupar pacotes em conexões e então extrair estatísticas delas para gerar as *features*.

Dada a lacuna em se gerar *features* do NSL-KDD, pretende-se treinar um modelo em que isso possa ser feito constantemente. Para tanto, pretende-se treinar um algoritmo de ML com as 26 *features*, que o *software* proposto consegue extrair. Assim, é possível a avaliação de amostras em algoritmos de ML a partir de tráfego em tempo real ou *benchmarks* (com *features* compatíveis ou não, já que o *software* proposto foi feito para realizar a conversão dos dados brutos em *features* do NSL-KDD).

Cita-se também que sem um módulo de extração de *features* a adição de tráfego para o treinamento de algoritmos de ML limita-se somente a *benchmarks* com subconjunto de *features* do NSL-KDD. Entre os trabalhos que exploram o mesmo subconjunto de *features* em busca de melhorar os algoritmos, ressalta-se o trabalho de [22] e [21]. O primeiro realiza a fusão do *dataset* NSL-KDD com o *dataset* UNSW-NB15. Entretanto, a fusão não ocorre de forma a adicionar amostras, mas sim realizando um *LEFT JOIN* a partir das *features* em comum dos dois *datasets*, que são poucas. Já o segundo cria critérios específicos para selecionar as amostras mais importantes do KDD99 para construir o NSL-KDD. Assim, também não adiciona amostras, mas indica métodos para se selecionar amostras para fazerem parte de um conjunto de treino.

Tabela 2.11: Exploração das lacunas relacionadas ao aprendizado de máquina no NSL-KDD

Lacunas	Exploração das lacunas no trabalho
Definição de critérios para adição de amostras no conjunto de treino	Criação um algoritmo para adicionar somente as amostras mais significativas ao conjunto de treinos.
Experimentos com adição de amostras no conjunto de treino	Experimentar a adição de amostras do Kyoto 2006 + junto ao NSL-KDD e verificar o desempenho no arquivo de teste do NSL-KDD.
Algoritmo de ML com manutenção constante	Treinar um modelo que possa sofrer adição de novas amostras. Isso será possível ao utilizar um modelo com as *features* extraídas pelo *software* proposto.
Classificação de amostras extraídas pelo *software* proposto	No trabalho foram geradas amostras do *benchmark* CIC-IoT-2023, a partir do *software* proposto, para avaliar o algoritmo de ML em tráfego IoT.

A fim de tornar o algoritmo de ML mais generalista, e assim realizar a manutenção de um IDS, foram feitos experimentos para se avaliar o uso do Kyoto 2006+ *dataset* para aumentar o conjunto de treino do *dataset* NSL-KDD, já que os dois possuem 14 *features* em comum. Até onde se tem conhecimento não foi visto nenhum trabalho que tentasse fazer este tipo de agregação (NSL-KDD + Kyoto 2006+ Dataset).

Ainda, ressalta-se, como ponto importante a ser explorado, os métodos para se selecionar amostras durante o treinamento de um modelo de ML. Ao se criar um *software* que permite adicionar amostras é importante adicioná-las com critério para garantir que o modelo não fique enviesado. Isso foi feito no presente trabalho ao se criar um algoritmo, baseado no algoritmo de amostragem utilizado para transformar o KDD99 no NSL-KDD proposto por [21], para adicionar as amostras mais relevantes do Kyoto 2006+ junto ao NSL-KDD.

Para tornar mais claro para o leitor como as lacunas relacionadas ao apredizado de máquina no NSL-KDD foram exploradas no presente trabalho foi feita a Tabela 2.11.

3 METODOLOGIA

3.1 SOFTWARE PARA GERAÇÃO DE *FEATURES*

Nesta seção descreve-se os passos necessários para a confecção do *software* proposto por [15] (que é parte do presente trabalho) para extração de *features* do NSL-KDD. É exposta arquitetura do *software* e o detalhamento de cada um dos módulos que ele contém.

3.1.1 Arquitetura

A Figura 3.1 mostra a arquitetura proposta. Ela é composta por: um módulo de captura composto pelo Tshark; um módulo de *containers* composto pelo MongoDB, que salva as informações essenciais a respeito dos pacotes de rede, e o Redis, que salva as informações do estado do processamento dos pacotes e conexões (fluxos) da rede; e por último, um módulo de filtragem para montar as conexões e extrair as *features* do NSL-KDD, salvando-as em um arquivo CSV.

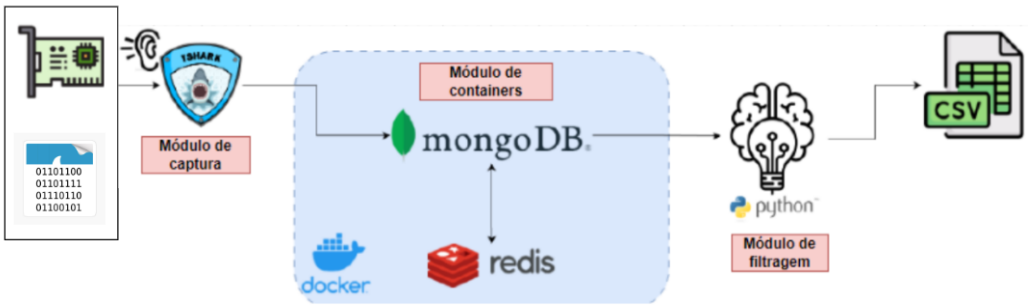

Figura 3.1: Arquitetura proposta adaptada de [15]

3.1.2 Captura dos dados

Como forma de ajudar no entendimento do processo de captura dos dados, foi feito o pseudocódigo mostrado no Algoritmo 1, que tenta explicar de forma sucinta o funcionamento desse módulo, com o pseudocódigo adaptado de [15].

Inicialmente no Algoritmo 1 escolhe-se entre extração *online*, isto é, a partir de uma interface de rede, ou *offline*, que extraí os dados de rede a partir de um arquivo PCAP.

A função utilizada no *pyshark* para monitorar uma interface de rede é a $sniff\_continuously(net\_interface)$, monitorando a interface selecionada de maneira ininterrupta, enquanto a função para extrair dados de um arquivo PCAP é a $FileCapture(pcap\_file)$, conforme mostrado no primeiro bloco $if\ else$ do Algoritmo 1.

Após a decisão entre extração *online* ou *offline*, existe um bloco for que extraí de cada pacote as informações: $localtime$ (*timestamp*), $protocol$ (UDP, TCP ou ICMP), $byte$ (quantidade de *bytes*), src_addr

(endereço ip de origem), $ip\_checksum$ (*checksum* do pacote), src_port (porta de origem do pacote), dst_port (porta de destino) e dst_addr (endereço ip de destino).

Tendo extraído as informações fundamentais de cada pacote, é avaliado se os pacotes utilizam os protocolos TCP, UDP ou ICMP, estando essa verificação no último bloco $if\ else$ do Algoritmo 1. Para cada um desses protocolos existe um *template* que salva as informações de forma adequada em um *container* do MongoDB. Isso porque o TCP, diferente do ICMP e UDP, possui *flags* em seu cabeçalho, sendo usadas pelo NSL-KDD para construção de *features*.

Algoritmo 1 Captura de dados

```
if extraction = online then
    packets_array ← sniff_continuously(net_interface)
else if extraction = offline then
    packets_array ← FileCapture(pcap_file)
end if
for packet in ( packets_array) do
    localtime ← packet.sniff_timestamp;
    protocol ← packet.transport_layer;
    byte ← packet.ip.len;
    src_addr ← packet.ip.src;
    ip_checksum ← packet.ip.checksum;
    src_port ← packet[protocol].srcport;
    dst_addr ← packet.ip.dst;
    dst_port ← packet[protocol].dstport;
    if protocol = TCP then
        tcp_flags ← packet[protocol].flags
        Saves on Mongodb with TCP data format;
    else if protocol = UDP then
        Saves on Mongodb with UDP data format;
    else if protocol = ICMP then
        Saves on Mongodb with ICMP data format;
    else
        Ignore the packet;
    end if
end for
```

3.1.3 Filtragem dos dados

Após salvar as informações dos pacotes em um *container* MongoDB, foi realizado o processamento dessas informações, na linguagem de programação Python, a fim de obter as *features* do *dataset* NSL-KDD.

Na Figura 3.1, que expõe a arquitetura, o módulo de filtragem é mostrado como um cérebro, por se tratar do módulo onde ocorre o maior processamento de informação, sendo o principal componente para extração de *features*.

No presente trabalho foram extraídas 26 *features* das 28 ($wrong\_fragment$ e $service$ não foram implementadas) pertencentes as categorias I, III e IV mostradas na Figura 2.2 retiradas do trabalho de [3].

Optou-se por não extrair as *features* da camada II por elas trazerem a complexidade de analisar *payloads*, o que pode ser inviável, por vezes, quando eles estão criptografados. Assim, as *features* extraídas estão relacionadas ao fluxo em análise (categoria I) e às informações de seus antecessores (categorias III e IV).

O processo para obter *features* é segmentado com base nessas categorias. Para detalhar os passos para extrair as *features* de cada categoria na arquitetura proposta foi feito o pseudocódigo mostrado no Algoritmo 2, adaptado de [15]. Reitera-se que, no presente trabalho, o uso da palavra conexão é análogo ao da palavra fluxo para se referir ao conjunto de pacotes com mesma tupla na rede.

Ao analisar o pseudocódigo mostrado no Algoritmo 2, inicialmente é criada uma lista para guardar as informações a respeito dos fluxos a serem processados, chamada de $Flow\_list$. Em seguida, realiza-se o cálculo das *features* até que o programa seja interrompido manualmente, no caso de extrair pacotes de uma interface de rede, ou até que os pacotes acabem, no caso de se extrair a partir de um arquivo PCAP. (Isso está traduzido no primeiro *loop while* do Algoritmo 2).

Algoritmo 2 Filtragem de dados

$Flow\_list \leftarrow list()$
while $user\_stop\_signal$ == False & $pcap\_end\_signal$ == False **do**
    $start\_packets \leftarrow mongo.find(\{START\_CONN\_PACKETS\})$
    $flow\_tuple \leftarrow get\_next\_flow\_tuple(start\_packets)$
    $--$
    $current\_flow\_packets \leftarrow mongo.find(\{FLOW\_PACKETS\ from\ flow\_tuple)\}\}$
    $I \leftarrow compute\_category\_I\_features(current\_flow\_packets)$
    $--$
    $CL, SCL \leftarrow list()$
    $CL, SCL \leftarrow search\_last\_2\_seconds(Flow\_list, flow\_tuple)$
    $III \leftarrow Compute\_category\_III\_features(CL, SCL)$
    $--$
    $DHCL, DHSCL \leftarrow list()$
    $DHCL, DHSCL \leftarrow search\_last\_100\_flows(Flow\_list, flow\_tuple)$
    $IV \leftarrow Compute\_category\_IV\_features(DHCL, DHSCL)$
    $--$
    $features = concat([I, III, IV])$
    $append\_line\_on\_CSV(features)$
    $current\_flow\_info \leftarrow concat([flow\_tuple, features])$
    $Flow\_list.append(current\_flow\_info)$
end while

Como etapa inicial para obter as *features*, é necessário pesquisar no banco de dados os pacotes que indicam início de conexão ($start\_packets \leftarrow mongo.find(START\_CONN\_PACKETS)$). Para isso, é feito uma *query* no MongoDB que busca os pacotes com a SYN *flag* ou pacotes UDP/ICMP não processados ainda, isto é, que tenham *timestamp* maior do que o pacote de início de conexão escolhido na iteração anterior do *loop while*.

Dentre os pacotes encontrados, que indicam início de conexão, escolhe-se um que não tenha pertencido a um fluxo analisado anteriormente, sendo que isso pode acontecer no caso dos protocolos ICMP e UDP que não possuem *flags* para indicar o estado da conexão (início e término). Escolhido o pacote a partir

desses critérios, as informações contidas em seu cabeçalho de endereço ips e portas serão consideradas a tupla da conexão atual ($flow\_tuple \leftarrow get\_next\_flow\_tuple(start\_packets)$).

Obtida a tupla de conexão ($flow\_tuple$), busca-se no MongoDB os pacotes que pertencem a essa tupla (ip origem, ip destino, porta origem, porta destino) dentro de um determinado intervalo de tempo ($current\_flow\_packets$) para formar um fluxo (uma conexão). Construído o fluxo, extraí-se as *features* de categoria I, que estão relacionadas, exclusivamente, aos pacotes do fluxo atual.

As próximas *features* a serem extraídas são as de categoria III. Essas *features* extraem estatísticas das conexões com mesma porta e ip de destino com a conexão atual nos últimos 2 segundos. Para tanto, é necessário realizar uma filtragem nos fluxos antecessores, contidos em $Flow\_list$. Após a filtragem, salva-se a lista de fluxos com mesmo ip de destino que a conexão atual na lista $count\_list$ (CL) e os de mesma porta de destino na lista srv_count_list (SCL).

A partir das conexões contidas em CL e SCL é possível computar as *features* de categoria III do NSL-KDD ($III \leftarrow Compute\_category\_III\_features(CL, SCL)$).

Em seguida, calcula-se as *features* de categoria IV, que fazem extrações de estatísticas semelhantes a de categoria III, mas relacionadas as últimas 100 conexões antecessoras à conexão atual. Assim, acontece a filtragem de fluxos em $Flow\_list$, salvando as últimas 100 conexões com mesmo ip de destino na lista $dst\_host\_count\_list$ (DHCL) e as 100 últimas com mesma porta de destino na lista $dst\_host\_srv\_count\_list$ (DHSCL).

Por último, como indicado no fim do Algoritmo 2, as *features* de cada categoria são agrupadas e salvas em um arquivo CSV e a as informações da conexão atual são adicionadas na lista de conexões ($Flow\_list$) analisadas corretamente.

Um ponto importante a comentar é que caso não haja conexões (fluxos) com similaridade de endereço e porta de destino com a conexão atual, as *features* de categoria III e IV recebem valor 0.

Outro ponto a ressaltar é o *container* Docker do Redis, que desempenha o importante papel de guardar o identificador da última conexão que teve as *features* extraídas com sucesso e do último pacote salvo corretamente no banco de dados. Tal recurso foi feito pensando no caso da aplicação sofrer uma parada inesperada. Assim, quando a aplicação for reiniciada, ela retornará para o estado que estava, ao invés de processar novamente as mesmas conexões.

3.1.4 Inicializando o *software*

Um programa em Python foi desenvolvido para gerenciar todos os demais programas. Além disso, as interfaces gráficas foram criadas utilizando o Dialog dentro desse mesmo programa.

Ao iniciar esse programa principal em Python, a interface gráfica é carregada e são feitas perguntas usando as caixas de diálogo do dialog a fim de obter informações a respeito da interface de rede a ser monitorada ou do arquivo PCAP a ser processado, bem como das *features* o usuário deseja que sejam calculadas durante a execução do programa.

Após responder a tais perguntas, o programa irá inicializar os *containers* Redis e MongoDB. Em se-

guida, acionará o programa em Python que salva os pacotes no banco de dados e depois o programa em Python que extraí as *features* a partir das conexões montadas com os pacotes que já estão populando o MongoDB. A Figura 3.2 ilustra o fluxograma das etapas descritas.

3.2 MODELO DE *MACHINE LEARNING*

Para avaliar a capacidade de detectar ataques em rede a partir das 26 *features*, que se consegue extrair a partir do *software* desenvolvido nesta pesquisa, decidiu-se treinar e testar algoritmos de ML em *benchmarks* que já possuem algumas dessas *features* computadas, como é o caso do Kyoto 2006 + *dataset* (e do próprio NSL-KDD).

Enfatiza-se que se utilizaram apenas as 26 *features* extraídas pelo software nesse processo, pois, no caso do presente trabalho, se fossem utilizadas todas as 41 *features* (26 extraídas e 15 não extraídas pelo *software*), não seria possível realizar a manutenção de algoritmos a partir da adição de dados nos modelos de ML a serem treinados. Isso só pode ser feito de forma adequada tanto em tráfego em tempo real como por meio de *benchmarks* a partir das *features* extraídas.

Ainda como forma de tornar o conjunto de teste mais representativo e o trabalho do algoritmo mais desafiador, buscou-se fazer a avaliação das *features* do NSL-KDD no *dataset* CICIoT2023. Esse *dataset* não tem as *features* do NSL-KDD computados. Sendo assim, é necessário utilizar o *software* para extrair as *features* do NSL-KDD a partir dos arquivos PCAP do *dataset* CICIoT2023.

Diante do exposto, esta seção objetiva fornecer a metodologia empregada para encontrar o algoritmo de ML ótimo para classificação, sobretudo no arquivo de teste do NSL-KDD, e em outros *benchmarks* analisados. Para tanto, foram avaliados diferentes fatores como: modelos de ML, técnicas de extração de *features* e mesclagem de dados entre diferentes *benchmarks*.

3.2.1 Fatores analisados

Entre os fatores a serem alterados e analisados no trabalho tem-se: o conjunto de dados a ser utilizado para treino, validação e teste; as *features* a serem utilizadas; e os modelos de ML com seus hiperparâmetros particulares.

O primeiro fator mencionado é o conjunto de dados a ser usado para treinamento com as *features* desenvolvidas. Utiliza-se o *dataset* NSL-KDD para treinar os modelos de ML. Optou-se por tal *dataset*, pois é um refinamento do KDD-CUP 99, usando técnicas para reduzir o viés que ele apresentava. Outro ponto é possuir um conjunto de treino e teste, propício para medir a performance em detecção de anomalias, pois tem um subconjunto de ataques diferentes nos arquivos de treino e teste.

Ainda em relação ao conjunto de dados, utiliza-se também nos experimentos a mesclagem de dados entre o Kyoto 2006 + e o NSL-KDD inspirado no trabalho de [22], que realiza combinação entre os dados de *benchmarks* com o intuito de tornar o modelo de ML mais robusto.

Com base na literatura, avaliou-se realizar a extração de *features* para verificar quais as mais relevantes

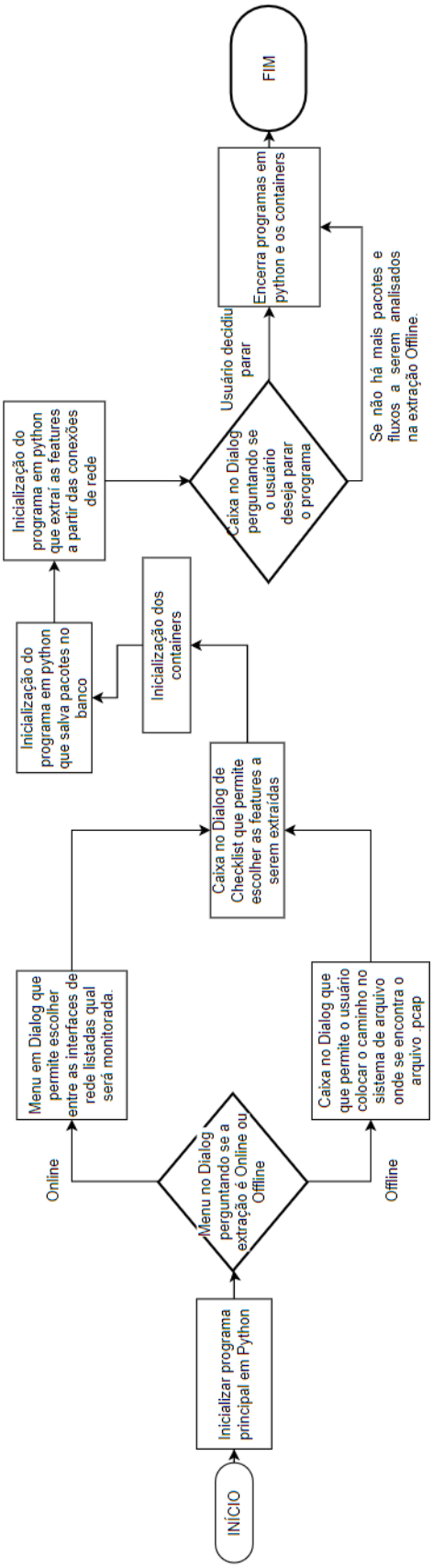

Figura 3.2: Fluxograma do *software* proposto.

dentre as 26 extraídas pelo *software*. Os métodos para extração de *features* foram baseados nos melhores resultados que foram encontrados na literatura no arquivo de teste do NSL-KDD (o algoritmo MFO) ; e no mais utilizado na literatura junto ao NSL-KDD (o algoritmo IG).

Os modelos utilizados serão baseados em árvores de decisões, usando-se os modelos DT, RF e adaBoost pelos motivos já explicitados nas secões anteriores.

O conjunto de teste para medir a performance é o arquivo de teste do NSL-KDD, por ser ponto de comparação em muitos trabalhos na literatura. Muitos trabalhos também medem o desempenho por meio de validação cruzada a partir do arquivo de treino do NSL-KDD. Entretanto, considera-se o arquivo de teste mais propício para medir performance em detecção de anomalias, por possibilitar conjuntos diferentes de ataques durante o treino e teste de algoritmos.

O fluxograma mostrado na Figura 3.3 mostra as etapas que serão utilizadas para se chegar no algoritmo de de ML ótimo. Serão treinados 12 modelos de ML no total: 3 modelos utilizando todas as *features*; 6 utilizando as *features* selecionadas usando IG e MFO; e 3 usando o conjunto de dados mesclado do NSL-KDD e do Kyoto *dataset* 2006+ (por óbvio, com as *features* do NSL-KDD que tem intersecção entre ambos).

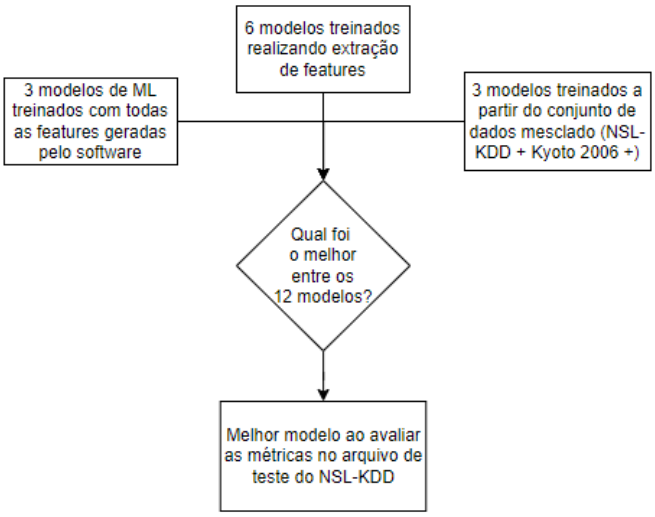

Figura 3.3: Etapas para se chegar no algoritmo ótimo

Como acréscimo para avaliar a performance, busca-se realizar testes, a partir do algoritmo ótimo, em tráfego de rede IoT utilizando-se de amostras do *dataset* CICIoT2023 (não houve tempo para analisar o *dataset* inteiro, que consta com mais de 500 GB), que está salvo em arquivos PCAP.

3.2.1.1 Avaliação inicial usando todas as *features*

Como forma de verificar o desempenho inicial de alguns algoritmos foram feitos testes com o NSL-KDD contendo todas as 26 *features* que são extraídas pelo *software* proposto.

Os dados neste treinamento foram normalizados e foi aplicada a codificação *one-hot encoding* nos cam-

pos de texto (*flag* e protocolo). Todos os algoritmos de ML no presente trabalho utilizaram de normalização e da codificação citada, a menos que seja explicitamente mencionado o contrário.

Após codificados, os dados foram treinados em 3 algoritmos de ML, que usaram os modelos DT, adaBoost e RF no *dataset* de treino do NSL-KDD, e avaliadas as métricas *accuracy*, *precision*, *recall* e *f1-score*, por meio do método *5-fold cross-validation* e do *dataset* de teste (que é o arquivo de teste do NSL-KDD).

Como forma de ajustar os parâmetros (e evitar *overfitting*) no caso da DT, foi utilizado um algoritmo de pós poda. Nesse algoritmo, são testados modelos de DT a partir dos valores de ccp_alpha obtidos da função $cost\_complexity\_pruning\_path$ de [38].

Começa-se dos maiores valores (*underfitting*) para os menores valores de ccp_alpha (*overfitting*). Quando acontece uma queda na acurácia de validação de um modelo em relação ao da iteração anterior, o algoritmo escolhe o valor de ccp_alpha antes da referida queda como ideal. No caso de não encontrar uma queda no valor de acurácia, o algoritmo segue até encontrar ccp_alpha = 0. O pseudocódigo é mostrado no Algoritmo 3. Tal abordagem para escolha de ccp_alpha foi baseada na proposta em [38].

Algoritmo 3 Pós poda em DT

```
set_ccp_alphas ← Invert(set_ccp_alphas)
best_model = None
best_ccp_alpha = None

for ccp_alpha in set_ccp_alphas do
    best_val_acc = 0
    model = sklearn.dt_model(ccp_alpha)
    current_acc = model.predict(val_set)

    if best_val_acc < current_acc then
        best_val_acc = current_acc
        best_ccp_alpha = ccp_alpha
    else if best_val_acc > current_acc then
        best_model = sklearn.dt_model(best_ccp_alpha)
        break
    end if
end for
```

Já no caso dos outros dois modelos, foi criado um estudo no optuna que realiza 50 iterações, buscando os hiperparâmetros que conseguem a melhor média de acurácia ao se utilizar o método *5 fold cross-validation*.

Para o modelo de RF, foram regulados os parâmetros considerados mais importantes segundo [29], quais sejam: $n\_estimators$ (variando de 1 a 100) e $max\_features$ (com valores sqrt, log2, None). Já para o adaBoost, os hiperparâmetros considerados foram: $n\_estimators$ (variando de 1 a 100) e $learning\_rate$ (variando de 0.001 até 0.1).

A Figura 3.4 ilustra esse processo de regulação de hiperparâmetros para DT e os outros dois modelos. Ressalta-se que a regulação de hiperparâmetros será sempre a mesma para os modelos subsequentes treinados neste trabalho, o que não impede que outros fatores variem, como o conjunto de dados e de

features.

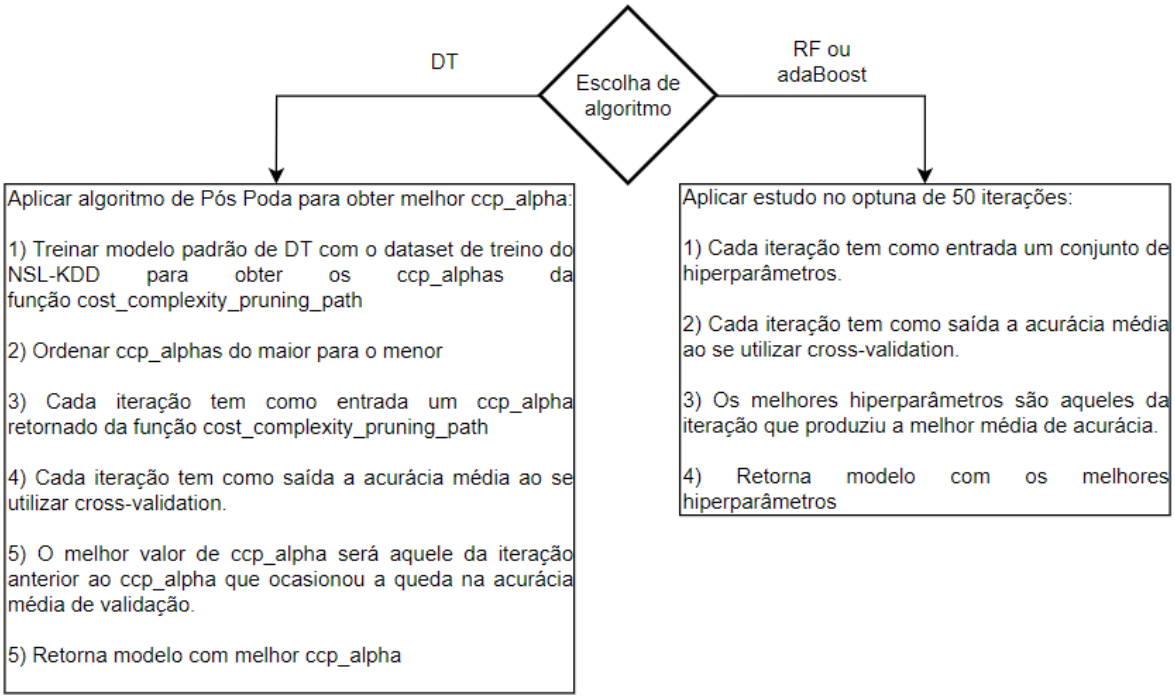

Figura 3.4: Regular hiperparâmetros

3.2.1.2 Avaliação com extração de *features*

Foram feitas extrações de *features* no intuito de avaliar uma possível melhoria nos *benchmarks* de teste, acreditando que esse resultado significaria uma melhor generalização dos algoritmos de ML para detectar anomalias. Os métodos de extração de *features* utilizados para esse feito foram o IG e o MFO.

A função $fitnessFunction(M)$ utilizada no MFO foi o algoritmo de DT. A saída da função é a matriz OM, que contém as acurácias ao se utilizar o algoritmo DT em cada linha de M, ou seja, as acurácias para cada um dos subconjuntos de *features*.

No caso do MFO, o arquivo de treino do NSL-KDD foi dividido entre um conjunto de treino (75 % do arquivo de treino) e validação (25 % do referido arquivo). Isso porque realizar *cross-validation* torna o algoritmo muito demorado, principalmente se for escolhido um número grande de iterações. Outro ponto é que os dados foram normalizados e a codificação utilizada foi a ordinal, seguindo o que foi proposto por [8] ao utilizar do mesmo algoritmo.

Ao rodar o algoritmo várias vezes (mais de 10 x), percebeu-se que, logo na primeira iteração, as acurácias de validação resultantes da função $fitnessFunction()$ já eram maiores que 99%. Nesse ponto, percebeu-se que os valores de acurácia ainda eram altos, mesmo com as condições iniciais deferentes, ou seja, a matriz Moth $(M_{n,d})$ diferente.

Assim, para extrair as melhores *features*, decidiu-se rodar o algoritmo MFO 10 vezes utilizando somente 1 iteração, e escolher somente as *features* que foram selecionadas, ao mesmo tempo, em 7 ou mais

das 10 iterações. Esperou-se com isso obter as *features* que são em média mais selecionadas pelo algoritmo MFO ao final de uma iteração.

No caso do algoritmo IG foi utilizado o arquivo de treino do NSL-KDD para extrair as melhores *features*. Assim como no MFO, foi utilizada a codificação ordinal para os campos textuais e foi feita a normalização dos dados. A saída do algoritmo, que releva a importância de cada *feature*, está normalizada para valores entre 0 e 1, sendo que as *features* com importância superior a 0.1 foram mantidas.

3.2.1.3 Avaliação com um conjunto mesclado

Como forma de generalizar melhor um algoritmo de ML, foram realizados experimentos mesclando o conjunto de dados do NSL-KDD e o Kyoto 2006 + inspirado no trabalho de [22], que faz a mesclagem do *dataset* UNSW-NB15 com o NSL-KDD, e [21], que seleciona um conjunto de amostras do KDD99 para gerar o NSL-KDD (Optou-se por utilizar o Kyoto 2006 + ao invés do UNSW-NB15, pois o primeiro tem mais *features* em comum com NSL-KDD).

Para gerar o conjunto de treino mesclado, foi feito um algoritmo para selecionar amostras do Kyoto 2006 +, já que é um *dataset* que inclui milhares de amostras. O Algoritmo 4 mostra um pseudocódigo do funcionamento da amostragem do Kyoto 2006 + *dataset*.

O Algoritmo 4 recebe como entrada a quantidade de amostras que se deseja incluir no *dataset* NSL-KDD, $qtd\_add\_samples$. O Kyoto 2006 + *dataset* contém as *features* para cada dia do ano durante 3 anos em arquivos CSV. Foram somadas as quantidades de amostras de todos os arquivos CSV e armazenados na variável $total\_kyoto\_size$.

No esforço de extrair informações de cada dia do ano de forma proporcional, amostra-se, de forma aleatória, em cada arquivo CSV o equivalente a fração $\frac{qtd\_add\_samples}{total\_kyoto\_size}$, que indica a porcentagem do Kyoto 2006+ *dataset* que será utilizada. Assim, cada arquivo contribui com a mesma porcentagem para o tamanho final das amostras adicionadas como indica a Equação 3.1.

Ainda é feito um processamento adicional em cada arquivo, sendo que a quantidade de amostras em cada arquivo n , $samples\_file_n = \frac{qtd\_add\_samples}{total\_kyoto\_size} * file\_size_n$, tem a mesma quantidade de amostras normais e de ataque, ou seja, cada um com a quantidade $\frac{samples\_file_n}{2}$ (cita-se, como observação, que para cada arquivo é necessário verificar se a quantidade máxima de amostras de cada classe, normal ou ataque, é maior ou igual do que $\frac{samples\_file_n}{2}$, caso não seja, seleciona-se a quantidade de amostras da classe com menos amostras no arquivo CSV como quantidade a ser adicionada para cada uma das classe).

$$qtd\_add\_samples = \frac{qtd\_add\_samples}{total\_kyoto\_size} * \sum^{n} file\_size_n \tag{3.1}$$

Tendo extraído as amostras de todos os arquivos CSV, é feito um processamento semelhante ao de [21], removendo as amostras repetidas para reduzir o viés do modelo.

Após, inspirado em [21], que cria um algoritmo próprio de seleção de amostras para transformar o KDD-CUP99 no NSL-KDD, foi feito um algoritmo para selecionar as amostras do Kyoto 2006 + mais relevantes para serem adicionadas no conjunto final de dados.

Algoritmo 4 Adicionar amostras

Input $\leftarrow qtd\_add\_samples$
$kyoto\_samples = None$
for csv_file in $kyoto\_files$ **do**
$qtd\_file\_add\_samples = \frac{qtd\_add\_samples}{total\_kyoto\_size} * csv\_file.size$

$quantity\_attack\_samples = \frac{qtd\_file\_add\_samples}{2}$
$quantity\_normal\_samples = \frac{qtd\_file\_add\_samples}{2}$

$attack\_samples = get\_random\_attack\_samples(csv\_file, quantity\_attack\_samples)$
$normal\_samples = get\_random\_normal\_samples(csv\_file, quantity\_normal\_samples)$
$file\_samples = concat([attack\_samples, normal\_samples], axis = 0)$

$kyoto\_samples = concat([kyoto\_samples, file\_samples], axis = 0)$
end for

$kyoto\_samples.drop\_duplicates()$

$kyoto\_samples['sample\_score'] = get\_6\_models\_samples\_scores(kyoto\_samples)$
$groupby\_score = file\_samples.groupby('sample\_score')['sample\_score'].value\_counts()$
$samples\_per\_cluster = cluster\_scores(groupby\_score)$

$kyoto\_samples = select\_more\_less\_scored\_samples(kyoto\_samples, samples\_per\_cluster)$
$dataset\_fusion = concat([train\_NSL\_KDD, kyoto\_samples], axis = 0)$

Isso foi feito neste trabalho a partir da escolha de 6 modelos de ML (DT, RF, NB, SVM e xgBoost, adaBoost). Esses usam o *dataset* NSL-KDD para treinamento de seus algoritmos e realizam a predição nas amostras a serem adicionadas do Kyoto 2006 + *dataset*.

Ao final da inferência por todos os 6 algoritmos de ML, cada amostra recebe um *score*, que indica quantos desses 6 algoritmos de ML foram capazes de prever seu rótulo corretamente. Em seguida ocorre uma redução de amostras de forma que as amostras que tiveram mais acertos nesses 6 modelos treinados sofrerão redução proporcionalmente maior do que as que tiveram menos acerto, semelhante ao que é feito para transformar o KDD99 no NSL-KDD.

Em [21] a escolha da proporção ocorre levando em conta que o número de amostras é diretamente proporcional ao *score*, o que era verdade para o conjunto de dados analisado naquele trabalho. Entretanto, nem sempre isso pode ser verdade, sendo necessário criar grupos (*clusters*) de *score* que satisfaçam essa situação, o que é apresentado no Algoritmo 4 na função $cluster\_scores(groupby\_score)$, que recebe a quantidade de amostras por *score* ($groupby\_score$) e as agrupa de modo que as amostras com mais *score* terão grupos com mais amostras.

Depois de realizar todos os processamentos citados, o número de amostras provavelmente reduzirá bastante em relação ao seu número inicial almejado ($qtd\_add\_samples$), mas acredita-se, com base no que foi feito em [21], que realizando isso, obtém-se um conjunto de dados mais representativo.

3.2.2 Arquivo de teste CIC-IoT-dataset 2023

Ao final de todos os testes envolvendo o *dataset* NSL-KDD e o Kyoto 2006 +, foi realizada a inferência de tráfego no *dataset* CICIoT2023. Para tanto, escolheu-se o algoritmo de ML que nas etapas anteriores teve melhor desempenho no conjunto de teste do NSL-KDD nas 4 métricas avaliadas (acurácia, precisão, *recall* e *f1-score*).

Para realizar tal feito, antes é necessário extrair as *features* do NSL-KDD se utilizando do *software* desenvolvido, já que os dados do CICIoT2023 estão codificados em arquivos PCAP.

Um ponto a ser ressaltado é que o CICIoT2023 consta com mais de 500 GB de dados codificados em arquivos PCAP. Como não houve tempo e recursos computacionais suficientes para analisá-lo por inteiro, foi realizada uma amostragem dele.

A amostragem se deu inicialmente selecionando o conjunto de arquivos PCAP a serem usados para extração de *features*. Escolheu-se o tráfego benigno e os ataques: *Backdoor*, *Dictionary*, Dos *HTTP flooding* e *Recon port Scan* para serem extraídos do *dataset* CICIoT2023.

Os arquivos dos ataques foram escolhidos baseando-se nas categorias de ataques contidas dentro do NSL-KDD. O objetivo com esses testes é verificar a capacidade do algoritmo de ML proposto em classificar o tráfego de um *benchmark* consideravelmente mais recente. Com isso, cria-se um cenário desafiador para o algoritmo, principalmente por se tratar de ataques e tráfego normal em rede IoT.

É importante destacar que, dentro de uma categoria de tráfego, existem vários arquivos PCAP, como mostrado na Figura 3.5, escolhendo-se sempre o primeiro arquivo do diretório que contém os PCAP da categoria do tráfego (com exceção do tráfego benigno em que foram escolhidos os dois primeiros arquivos).

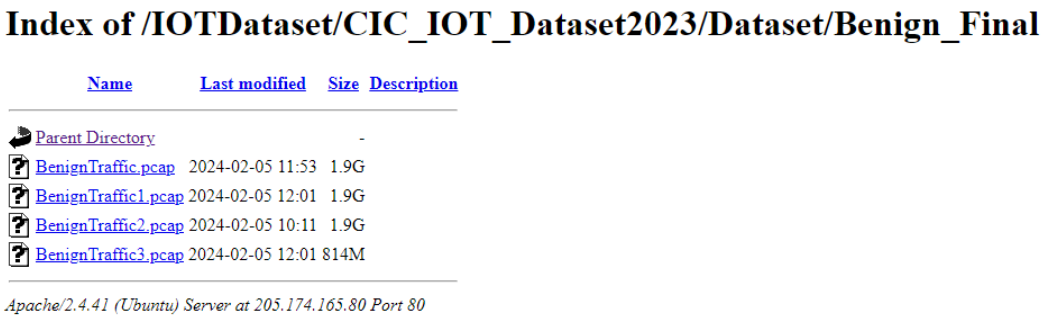

Figura 3.5: Pasta contendo arquivos PCAP da categoria de tráfego benigno no CICIOT2023 retirada de [16].

Após escolher os arquivos PCAP, esses são divididos em partes de no máximo 2MB, para agilizar o processamento pelo *software* proposto, já que cada arquivo tem entre 1.5 GB a 2GB, salvo algumas exceções que tem alguns MB de tamanho. Com isso, os arquivos PCAP menores são processados pelo *software* desenvolvido para obter as *features* do NSL-KDD.

Antes de passar para o algoritmo de ML ainda foi feita uma amostragem dentro dos fluxos gerados para inferência com o intuito de otimizar o tempo de predição do conjunto de teste retirado do CICIoT2023. Nesse ponto, é retirada uma porcentagem equivalente de cada arquivo de tamanho de 2MB, imitando o processo feito para amostrar o *dataset* Kyoto 2006 +.

Obtidas as predições das conexões a partir do *software* é então montada uma matriz de confusão, da

qual é possível extrair diversas métricas de performance. A Figura 3.6 ilustra o pipeline proposto para obter o conjunto de teste do CICIoT2023.

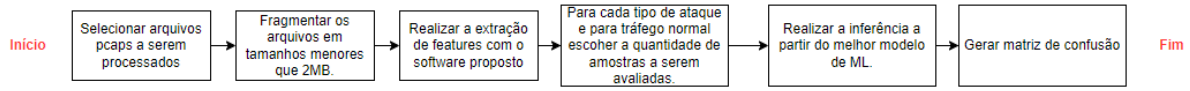

Figura 3.6: *Pipeline* para construir conjunto de teste do CICIoT2023

4 RESULTADOS

4.1 AVALIAÇÃO DO *SOFTWARE* DE EXTRAÇÃO DE *FEATURES*

4.1.1 Software em funcionamento

Como forma de avaliar o *software* foram feitos testes em uma máquina Ubuntu 22.04 COM 4GB de RAM no VirtualBox. Ao iniciar o *software* o usuário interage com a sequência de telas mostradas na Figura 4.1.

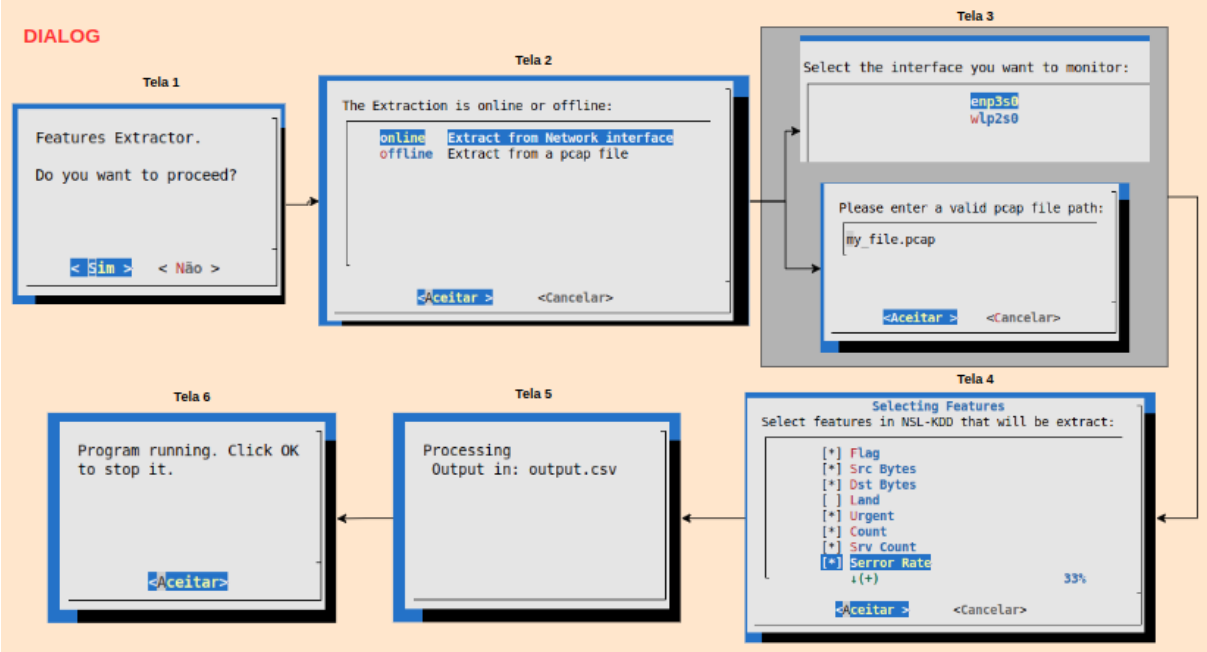

Figura 4.1: Sequência de telas do *software* no Dialog

A primeira tela pergunta se o usuário quer seguir com a extração, a segunda se a extração será *online* ou *offline*. Já a terceira tela dependerá se o usuário escolhe entre uma extração *online* ou *offline*.

No caso da extração ser *online*, a terceira tela mostrará uma lista com as interfaces de rede da máquina para que o usuário escolha a interface a ser monitorada. Se a extração for *offline* então aparece uma tela para o usuário digitar o caminho do arquivo PCAP a ser usado para extração.

A quarta tela mostra uma lista de *features* que podem ser selecionadas ou não para serem extraídas. A quinta tela mostra o caminho onde serão salvas as *features* geradas a partir das extrações.

Por fim, a última tela pergunta se o usuário deseja parar a extração por conta própria. No caso de uma extração *online*, a parada do programa depende do desejo do usuário, enquanto no caso *offline* o fim do programa acontece automaticamente ao final do processamento das conexões no arquivo PCAP escolhido.

Na Figura 4.2 é dado um exemplo de saída de um arquivo CSV ao se utilizar o *software* para extrair as *features* de uma das interfaces da máquina virtual supracitada. A saída contém, em cada linha, algumas

das *features* do NSL-KDD e algumas informações sobre o fluxo de rede analisado (ip, porta e *timestamp*).

Timestamp	Duration	Protocol	Source IP	Source Port	Dst IP	Dst Port	Flag	Src Bytes	Dst Bytes	Count	Srv Count	Srv Serror Rate	Rerror Rate
1694197286.47834	0.0387952327728271	UDP	192.168.1.200	34852	8.8.8.8	53	SF	65	114	10	10	0	0
1694197286.49807	0.0223472118377686	UDP	192.168.1.200	38113	8.8.8.8	53	SF	61	77	15	15	0	0
1694197289.12328	0.0251128673553467	TCP	192.168.1.200	58808	152.199.54.201	443	S1	112	60	15	31	1	0
1694197289.15023	0.0227453708648682	TCP	192.168.1.200	58810	152.199.54.201	443	S1	112	60	20	36	1	0
1694197289.19453	0.127603530883789	TCP	192.168.1.200	58812	152.199.54.201	443	S1	785	4336	26	41	1	0
1694197289.61731	0.327825546264648	TCP	192.168.1.200	58814	152.199.54.201	443	S1	785	4336	31	46	1	0

Figura 4.2: Amostra do arquivo CSV de saída do *software* com algumas *features* do NSL-KDD retirada de [15]

Em relação às informações que não são *features* do NSL-KDD, mostradas na Figura 4.2, elas foram mantidas, pois são úteis no processo de rotulação de tráfego entre normal e malicioso, caso se use o *software* para fazer um futuro *benchmark*.

As *features* foram implementadas de maneira a seguir o mais fielmente possível as especificações do *dataset* NSL-KDD, segundo o trabalho de [21]. Foram feitos sucessivos testes na máquina virtual Ubuntu citada para garantir um comportamento assertivo no *software* de extração.

4.1.2 Comparação do *software* proposto com outras soluções

Neste ponto do trabalho é feita a mesma análise que [15], comparando o *software* proposto com outros já existentes. Na Tabela 4.1, adaptada de [15], é comparado o *software* proposto com outros 3 *softwares*.

Ao comparar os *softwares*, notou-se que o proposto é o único que extraí as *features* do NSL-KDD de forma automática. O uso das novas amostras geradas pelo *software* vai permitir a manutenção de algoritmos de ML a partir do incremento na base de treino ou a partir da avaliação do algoritmo em bases de dados mais recentes (o que foi feito no presente trabalho ao se extrair amostras do CICIoT2023 com o *software* proposto e avalia-las no algoritmo de ML treinado).

Outro ponto notado no *software* proposto, é o fato dele ser o único a ter técnicas de recuperação de falha, permitindo que um fluxo que teve o processamento interrompido por algum tipo de condição adversa, como por exemplo uma queda de energia, seja processado novamente, isto é, recupera o estado do programa em relação ao processamento dos pacotes e conexões.

Entre os *softwares* na Tabela 4.1, aquele que mais se parece com o proposto é o CICFlowMeter, que consegue, assim como o *software* proposto, realizar a extração de estatísticas de fluxo (as *features*) em tempo real de interface de rede ou a partir de um arquivo PCAP, gerando na saída, por padrão, um arquivo CSV.

A diferença marcante entre o *software* proposto e o CICFlowMeter reside no fato das *features* do CICFlowMeter não serem iguais a do NSL-KDD, exigindo um esforço a nível de ETL (*Extract, Transform and Load*).

Considera-se que a saída em formato CSV é uma vantagem do CICFlowMeter e do *software* proposto dado que é um formato amplamente conhecido, exigindo menos manipulação por parte do usuário em comparação com os formatos do Zeek-IDS e do Argus.

Tabela 4.1: Comparação entre os *softwares*

	Extração em tempo real	Extração de arquivo pcap	Geram estatísticas (features) de fluxo	Extração automática das *features* do NSL-KDD	Arquivo de saída padrão	Recuperação de Estado
CICFlow Meter [35]	X	X	X	-	CSV	-
Argus-3.0 [36]	X	X	X	-	argus-output-file	-
Zeek-IDS [37]	X	X	X	-	zeek-log-files	-
Software proposto	X	X	X	X	CSV	X

4.1.3 Sobre o registro de *software*

Tendo atingido os resultados desejados no *software* para extração de *features* do NSL-KDD, decidiu-se registrar a propriedade intelectual do *software*, chamado de NDE, no INPI (Instituto Nacional da Propriedade Industrial) na categoria de programa de computador (BR512023001038-3). Para isso, foram exercidos todos os procedimentos necessários de modo a proteger judicialmente os códigos desenvolvidos.

4.2 AVALIAÇÃO DOS ALGORITMOS DE ML

Neste ponto do trabalho, avalia-se os resultados dos modelos de ML treinados, seguindo os passos propostos na metodologia.

4.2.1 Resultado após avaliação inicial usando todas as *features*

Após realizar testes com as todas as *features* extraídas do *software* proposto, foram obtidos os resultados mostrados na Tabela 4.2.

Tabela 4.2: Avaliação Inicial dos algoritmos de ML com todas as *features* do *software* proposto

	Evaluation set	***Accuracy***	***Precision***	***Recall***	***f1-score***	**Hiperparâmetros**
DT	*cross-validation*	0.9965	0.9958	0.9967	0.9963	ccp_alpha=0.0
	nsl-kdd test file	0.8006	0.9583	0.6792	0.7950	
RF	*cross-validation*	0.9977	0.9984	0.9966	0.9975	max_features: sqrt, n_estimators: 95
	nsl-kdd test file	0.7890	0.9679	0.6510	0.7784	
AdaBoost	*cross-validation*	0.9545	0.9646	0.9365	0.9503	learning_rate: 0.09870964895589418, n_estimators: 100
	nsl-kdd test file	0.7812	0.9683	0.6365	0.7681	

Ao avaliar o arquivo de teste do NSL-KDD, percebe-se que o algoritmo com melhor acurácia, *recall* e *f1-score* foi o DT, enquanto que a melhor precisão foi do algoritmo adaBoost (0.9683).

Percebe-se, ainda, que o conjunto de dados avaliado com *cross-validation* apresentou um desempenho muito superior em relação a acurácia e *recall* do arquivo de teste do NSL-KDD. Nota-se isso, ao observar que no método *cross-validation* os menores valores das métricas são do algoritmo treinado com adaBoost,

com valores de 0.9545 e 0.9365, enquanto os maiores valores encontrados no conjunto de teste do NSL-KDD foram de 0.8006, 0.6792 no algoritmo DT, para as respectivas métricas.

4.2.2 Resultado aplicando algoritmos de *feature extraction*

Ao aplicar os métodos de extração de *features* seguindo a metodologia já explicitada, foram obtidas as *features* mostradas na Tabela 4.3 para o algoritmo MFO e as *features* mostradas na Tabela 4.4 para o IG.

Tabela 4.3: Features extraídas MFO

protocol_type	srv_rerror_rate	dst_host_same_srv_rate	dst_host_srv_serror_rate
src_bytes	rerror_rate	dst_host_count	dst_host_srv_count
flag	dst_host_same_src_port_rate	dst_host_serror_rate	srv_serror_rate
land	srv_diff_host_rate	srv_count	same_srv_rate
dst_host_diff_srv_rate	serror_rate	diff_srv_rate	dst_host_srv_diff_host_rate

Tabela 4.4: Features extraídas IG

src_bytes	dst_bytes	flag	same_srv_rate
diff_srv_rate	dst_host_srv_count	dst_host_same_srv_rate	dst_host_diff_srv_rate
dst_host_serror_rate	dst_host_srv_serror_rate	count	serror_rate
srv_serror_rate	dst_host_srv_diff_host_rate	dst_host_count	dst_host_same_src_port_rate

Após extrair as *features*, os modelos foram treinados com o NSL-KDD e foi feita a avaliação deles por meio de *5-fold cross validation* e do arquivo de teste do NSL-KDD.

A Tabela 4.5 mostra os resultados obtidos ao se treinar a partir das *features* extraídas pelo MFO e a Tabela 4.6 os resultados com as *features* pelo método IG.

Tabela 4.5: Resultados do algoritmo usando as *features* do MFO

	Evaluation set	***Accuracy***	***Precision***	***Recall***	***f1-score***	**Hiperparâmetros**
DT	*cross-validation*	0.9940	0.9947	0.9924	0.9935	ccp_alpha: 0.0005126
	nsl-kdd test file	0.7904	0.9677	0.6536	0.7802	
RF	*cross-validation*	0.9967	0.9973	0.9956	0.9964	max_features: log2, n_estimators: 95
	nsl-kdd test file	0.7840	0.9679	0.6418	0.7718	
AdaBoost	*cross-validation*	0.9439	0.9574	0.9204	0.9385	learning_rate: 0.09943401304116867, n_estimators: 92
	nsl-kdd test file	0.7687	0.9653	0.6157	0.7519	

Tabela 4.6: Resultados do algoritmo usando as *features* do IG

	Evaluation set	***Accuracy***	***Precision***	***Recall***	***f1-score***	**Hiperparâmetros**
DT	*cross-validation*	0.9831	0.9840	0.9795	0.9817	ccp_alpha: 0.00048539937420352593
	nsl-kdd test file	0.8020	0.9273	0.7076	0.8027	
RF	*cross-validation*	0.9938	0.9944	0.9928	0.9934	max_features: sqrt, n_estimators: 83
	nsl-kdd test file	0.7723	0.9651	0.6224	0.7568	
AdaBoost	*cross-validation*	0.9391	0.9627	0.9041	0.9325	learning_rate: 0.07319519673356795 , n_estimators: 97
	nsl-kdd test file	0.7323	0.9627	0.5510	0.7009	

Percebe-se dos resultados das Tabelas 4.5 e 4.6 que os valores das 4 métricas para o método *cross-validation* continuam satisfatórios, mas, de maneira geral, houve uma leve queda nas métricas analisadas, ao se comparar com a abordagem que utiliza todas as *features*, variando de milésimos percentuais a até um pouco mais de um ponto percentual em alguns casos.

Já, ao se avaliar as métricas relacionadas ao arquivo de teste do NSL-KDD, notou-se uma diminuição ou constância nas métricas de maneira geral. A única abordagem que representou um leve ganho nas métricas em relação abordagem sem extração de *features*, e que se julgou pertinente mencionar, foi a que utiliza o IG com DT, tendo aumento de acurácia para 0.8020 e de *recall* para 0.7076, mas em contrapartida uma redução na precisão para 0.9273.

4.2.3 Resultado após treinamento com *dataset* mesclado

A fim de verificar a possível melhoria nas métricas de desempenho, foi selecionado um valor de 1 milhão de amostras para serem extraídas ($qtd\_add\_samples = 10^6$). Ao remover as amostras redundantes e realizar a amostragem de forma a garantir a mesma proporção na adição de amostras em cada arquivo e aproximadamente o mesmo número de amostras classificadas como normais e ataques, obteve-se 572653 amostras.

Nessas amostras, a distribuição para cada acerto (*score*) entre os 6 modelos de ML treinados é como mostrado no gráfico da Figura 4.3. Com o intuito de imitar o que foi feito em [21], formaram-se grupos (*clusters*) de *score* de modo que quanto mais acertos, mais amostras o *cluster* (grupo) deveria ter. O resultado do processo de clusterização para satisfazer tal proporcionalidade é mostrado no gráfico da Figura 4.4 nas barras em azul.

Nota-se que os *clusters* com valores maiores de acertos (*scores*) tiveram redução de amostras proporcionalmente maior do que os *clusters* com menos acertos, o que é mostrado ao comparar as barras azuis na Figura 4.4, que indicam a proporção de amostras antes da redução, e as laranjas, que indicam a proporção depois da redução.

Os valores percentuais para cada grupo de *score* em relação ao total de amostras, bem como a redução feita ao remover proporcionalmente mais as amostras de grupos com *scores* mais altos, conforme [21], são mostrados na Tabela 4.7. Como último procedimento antes de combinar os dados ao NSL-KDD, foi feito balanceamento de classes dentro das 406388 amostras (que é o total mostrado na Tabela 4.7). Como a classe com menos amostras tinha 122314 instâncias, o total de amostras após balanceamento foi $2 * 122314 = 244628$ amostras.

Tabela 4.7: Distribuição de amostras por grupo

Grupos	**Antes**	**percentual**	**Depois = (100-percentual)*Antes**
I (0 acertos)	76880	13,43%	66558
II (1 ou 2 acertos)	97475	17,02%	80883
III (3 ou 4 ou 5 acertos)	183682	32,08%	124764
IV (6 acertos)	214616	37,48%	134183
Somatório	572653	100%	406388

Por fim, este conjunto de amostras do *dataset* NSL-KDD é mesclado ao conjunto de treino do NSL-KDD, formando uma só base para treinamento. Tendo o conjunto mesclado para treinamento, foi medido o desempenho dele utilizando-se o método *cross-validation* e o arquivo de teste do NSL-KDD. Os resultados obtidos a partir do *dataset* mesclado são mostrados na Tabela 4.8.

O algoritmo com melhor desempenho no *cross-validation* foi ao se utilizar o RF, alcançando valores

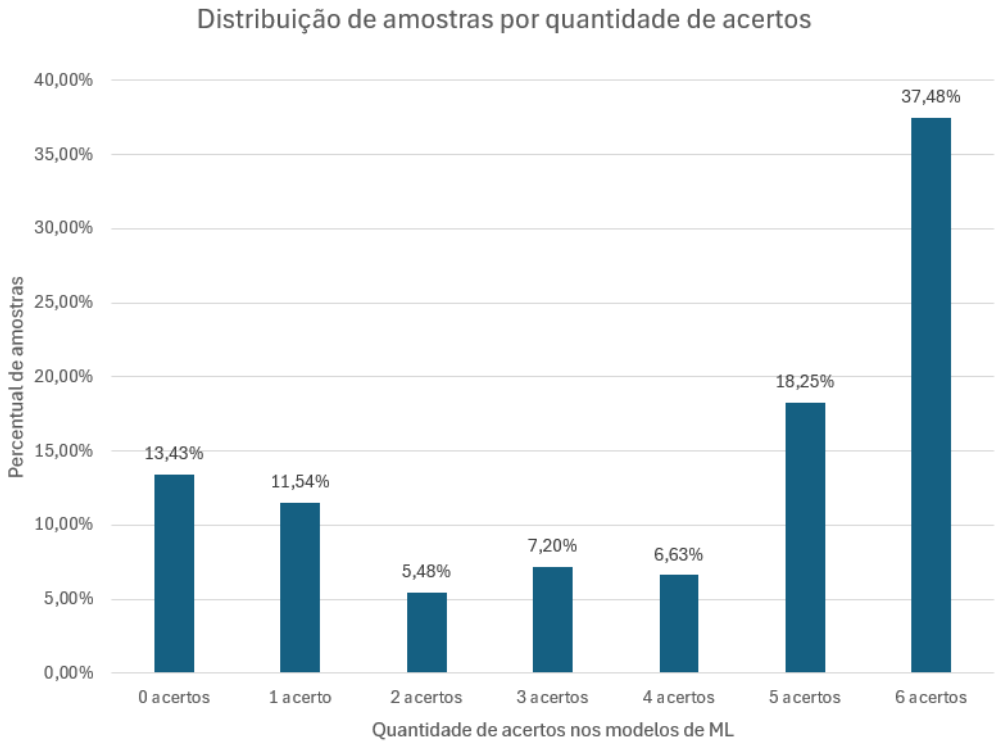

Figura 4.3: Gráfico que mostra as amostras por acerto

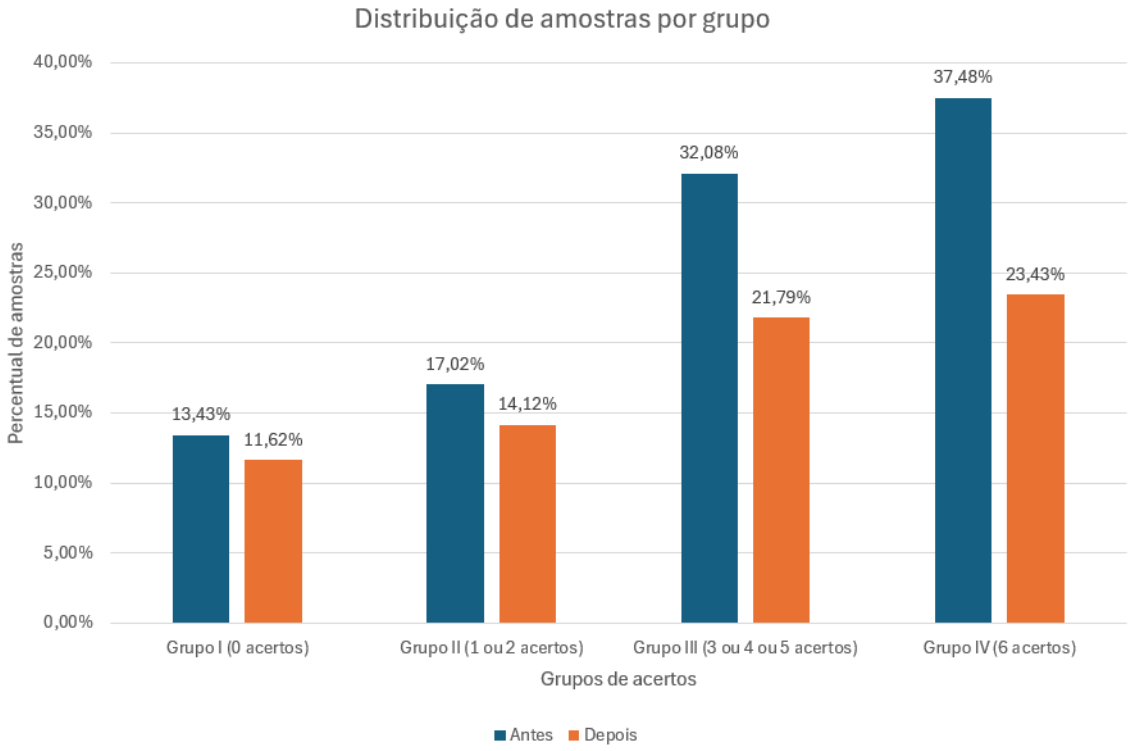

Figura 4.4: Gráfico que mostra as amostras por grupo (cluster)

acima de 98 % para todas as métricas. Já no caso do arquivo de testes do NSL-KDD, o melhor algoritmo foi ao se utilizar a DT, obtendo melhoras substanciais nas métricas de *recall* e acurácia, sem perdas significativas na métrica de precisão.

Vale ressaltar que esse ganho ainda ocorre em meio a redução de 26 *features* para 13 *features* (retirando-se o rótulo e a *feature service*) do *dataset* NSL-KDD, pois o Kyoto 2006 + *dataset* apresenta somente tais *features* em comum com as do NSL-KDD extraídas pelo *software* proposto.

Outro ponto a mencionar é o fato de que, para o *dataset* de teste, as métricas de acurácia, *recall* e *f1-score* tiveram aumento em relação às abordagens anteriores (com todas as *features* e com extração de *features*) para os 3 algoritmos testados (DT, RF e adaBoost).

Tabela 4.8: Métricas após realizar a fusão dos *datasets*

	Evaluation set	***Accuracy***	***Precision***	***Recall***	***f1-score***	**Hiperparâmetros**
DT	*cross-validation*	0.9404	0.9289	0.9513	0.9396	ccp_alpha = 0.0010013265013273489
	nsl-kdd test file	**0.8381**	**0.9487**	**0.7565**	**0.8417**	
RF	cross-validation	0.9898	0.9879	0.9914	0.9896	max_features: sqrt, n_estimators: 100
	nsl-kdd test file	0.7964	0.9670	0.6650	0.7880	
AdaBoost	*cross-validation*	0.8744	0.9017	0.8423	0.8611	learning_rate: 0.0981693349666789, n_estimators: 95
	nsl-kdd test file	0.8107	0.9416	0.7116	0.8110	

4.2.4 Algoritmo de ML ótimo resultante

Após treinar todos os 12 algoritmos de ML, escolheu-se aquele com melhor desempenho no arquivo de teste do NSL-KDD, como ótimo. Isso porque esse arquivo infere bem a capacidade de um modelo de identificar tráfego anômalo pela forma como foi elaborado.

Nesse ponto, escolheu-se o algoritmo que teve a maior métrica de *f1-score* no arquivo de teste, pois a métrica *f1-score* vai ser maior na medida que as métricas *precision* e *recall* tenham disparidade menor entre si, ou seja, é uma métrica que mede o equilíbrio entre a capacidade de identificar ataques (*recall*) sem aumentar muito os FP (tráfego normal classificado como ataque), o que resultaria em bloqueio de tráfego legítimo.

Levando em conta esses critérios, o algoritmo escolhido é o DT treinado com o conjunto de dados mesclado entre os *datasets* Kyoto 2006 + e NSL-KDD, considerando também esse algoritmo como ideal para ser utilizado junto ao conjunto de teste gerado do *dataset* CICIoT2023.

Com o intuito de realizar uma comparação justa do algoritmo proposto com outros na literatura, escolheu-se trabalhos que apresentam módulos para extração de *features*. Isso é importante, pois nesses trabalhos geralmente é apresentado um conjunto menor de *features* do NSL-KDD, dado a dificuldade de se processar todas as *features* do *dataset* NSL-KDD.

Mesmo tendo um conjunto de *features* geralmente inferior, os algoritmos de ML desses trabalhos apresentam a vantagem de poderem ser usados para predição de tráfego em tempo real ou para adição de tráfego de outros *benchmarks* no conjunto de treino, o que é fundamental para a manutenção de um IDS.

Os resultados mostrados na Tabela 4.9 mostram que o algoritmo proposto é o que apresenta menos

Tabela 4.9: Comparação de desempenho dos algoritmos de ML na literatura no arquivo de teste do NSL-KDD

	Quantidade de *features*	***accuracy***	***precision***	***recall***	***f1-score***
DNN [12]	28	0.8187	**0.9645**	0.7071	0.8159
RF [13]	-	0.8025	0.8510	0.8030	0.8010
Bagging ensemble [8]	23	**0.8744**	0.8800	**0.8700**	**0.8700**
algoritmo de ML proposto	**13**	0.8381	0.9487	0.7565	0.8417

features entre os comparados, o que é uma vantagem em termos de processamento, sem ficar atrás nas métricas avaliadas, estando próximo dos melhores valores para o arquivo de teste do NSL-KDD vistos na literatura.

4.2.5 Avaliação do algoritmo de ML junto ao CIC-IoT *dataset*

Para avaliar o algoritmo de ML, além de se utilizar do *dataset* de teste do NSL-KDD, também foram realizados testes no CICIoT2023. Essa é uma forma de avaliar a eficiência das *features* e do conjunto de dados mesclados (entre NSL-KDD e Kyoto 2006 +) em *datasets* mais recentes, sobretudo, no tráfego de redes IoT.

Realizada a extração de *features* nos arquivos PCAP escolhidos para avaliação, foram selecionadas 50000 amostras de conexões para cada tipo específico de ataque analisado para classificação pelo algoritmo. No caso do tráfego benigno, foram selecionadas 140000 conexões.

Houve ataques em que o primeiro arquivo PCAP não continha 50000 conexões, sendo que nesse caso todos os fluxos (conexões do arquivo) foram extraídos. Levando isso em conta, foram extraídos um conjunto de 275762 amostras do CICIoT2023.

Tendo essas amostras, realiza-se sua classificação, utilizando-se do algoritmo de DT pré-treinado com o conjunto de dados mesclados, obtendo a matriz de confusão mostrada na Tabela 4.10.

Tabela 4.10: Matriz de confusão ao testar o algoritmo de ML no CIC-IoT2023

		Ataque				Normal
		Backdoor	*Dictionary*	*Dos HTTP flood*	*Probe Recon port scan*	
Ataque	*Backdoor*	4262	-	-	-	5386
	Dictionary	-	10307	-	-	15807
	Dos HTTP flood	-	-	47093	-	2907
	Probe Recon port scan)	-	-	-	36098	13902
Normal		47978				92022

Ao calcular as métricas de acurácia, precisão e *recall* a partir da matriz de confusão mostrada na Tabela 4.10, considerando duas classes (ataque e normal), ou seja, agrupando os tipos de ataques em uma classe só, foram obtidos os valores de 0.6882, 0.6708 e 0.7201, para as respectivas métricas.

Esperava-se resultados mais promissores, sobretudo em relação aos FP (falsos positivos), ou seja, as amostras normais classificadas como ataques, que foram 47978. Em um algoritmo é fundamental evitar que esse valor seja alto para não bloquear tráfego legítimo e prejudicar os serviços de rede.

Entre os possíveis motivos para o resultado não ter sido tão bom quanto esperado, é o fato de que o

tráfego normal dos *datasets* Kyoto 2006 + e NSL-KDD não é especifico de redes IoT como é o caso do *dataset* CICIoT2023. Assim, mesmo que a mesclagem produza resultados promissores no conjunto de dados do NSL-KDD e Kyoto 2006 +, o resultado pode não ser totalmente refletido em um *dataset* que contém majoritariamente tráfego IoT.

Outro ponto a ser mencionado, é o fato de que, ao realizar a mesclagem dos *datasets* Kyoto 2006 + e NSL-KDD, houve a perda de *features* que poderiam ser importantes no desempenho do algoritmo. Como forma de contornar isso, poderia se adicionar as *features* faltantes usando algum algoritmo de regressão, tendo como entrada as *features* que se têm. Todavia, isso não é garantia de um desempenho melhor, considerando-se necessários mais testes para avaliar o caso.

Em suma, é notória a necessidade de manutenção do *dataset* de treino para melhorar o desempenho do algoritmo ótimo no CICIoT2023. Uma possível melhoria a ser feita é adicionar amostras, sobretudo de tráfego benigno do próprio CICIoT2023, no *dataset* de treino mesclado.

Para tanto, pode-se usar do algoritmo de adição de amostras proposto no trabalho. Nesse ponto, é necessário tomar cuidado para não utilizar as amostras adicionadas do CICIoT2023 no *dataset* mesclado em uma futura avaliação de desempenho do *dataset* CICIoT2023.

4.2.6 Discussão

Neste ponto, expõe-se os desafios enfrentados no desenvolvimento do trabalho e observações julgadas pertinentes.

4.2.6.1 Sobre os conjuntos de dados que contém as *features* do NSL-KDD

Em relação ao conjunto de dados, cita-se o desafio de implementar todas as *features* do NSL-KDD, pois as *features* da Categoria II do NSL-KDD envolvem a extração de informações do *payload* ou de *logs* de aplicações, o que gera uma complexidade considerável.

Outra observação é o fato de não se ter encontrado na literatura um método claro para extração de *features* do NSL-KDD, razão pela qual se propôs um *software*, detalhando sua arquitetura e as etapas para extração de *features* a a partir da referida arquitetura.

Houve dificuldade em achar *benchmarks* que implementassem as *features* do NSL-KDD, encontrando-se o Kyoto 2006 + e o UNSW-NB15. Entretanto, esses *datasets* continham apenas um subconjunto das *features* do NSL-KDD em suas bases.

4.2.6.2 Sobre a avaliação dos algoritmos de ML

Em relação a performance de algoritmos, julga-se que muitos trabalhos na literatura não deixaram claro o conjunto de dados utilizado para teste, expondo, por vezes, os resultados relativos a *cross-validation* em detrimento dos resultados no conjunto de teste do *dataset* NSL-KDD (que mensura de forma mais realista o desempenho de um IDS, pelos motivos já explicitados).

Relembra-se que a criação de um algoritmo restrito as 26 *features* extraídas pelo *software* é o que possibilita realizar a manutenção do algoritmo de ML por meio de adição de amostras em tempo real ou de outros *benchmarks*.

4.2.6.3 Sobre usar o arquivo de teste do NSL-KDD para realizar a extração de features

Ao rodar o algoritmo MFO, não foram obtidos os resultados em [8], atingindo-se resultados próximos somente ao se utilizar o conjunto de teste do NSL-KDD como conjunto de validação durante a execução do algoritmo MFO, ou seja, valendo-se de suas acurácias como retorno da função $fitnessFunction()$.

Reitera-se que o algoritmo utilizado na função $fitnessFunction()$ é o DT com os dados normalizados e usando a codificação ordinal para os campos textuais. Além disso, o modelo tem os valores de hiperparâmetros como os padrões da biblioteca *scikit learn*.

Ao fim de 100 iterações, foi possível obter com o algoritmo de DT o conjunto de *features* mostradas na Tabela 4.11, sendo que o resultado do algoritmo para as 4 métricas analisadas extraídas no arquivo de teste do NSL-KDD é exposto na Tabela 4.12.

Tabela 4.11: *Features* extraídas ao usar o arquivo de teste do NSL-KDD como conjunto de validação

dst_host_same_srv_rate	dst_bytes	srv_serror_rate
duration	land	dst_host_same_src_port_rate
flag	dst_host_srv_rerror_rate	dst_host_rerror_rate
protocol_type	count	src_bytes
dst_host_srv_serror_rate	dst_host_serror_rate	

Tabela 4.12: Resultados da DT ao usar o arquivo de teste do NSL-KDD como conjunto de validação no MFO

accuracy	precision	recall	f1-score
0.8364	0.9663	0.7384	0.8371

Considera-se que, ao se utilizar o conjunto de teste do NSL-KDD para extração de *features*, ocorre um vazamento de informações de treino no conjunto de teste, que deveria ser um conjunto do qual não se tem nenhuma informação prévia.

Assim, para o caso em estudo, o correto seria usar o arquivo de teste do NSL-KDD como conjunto de validação. A consequência disso é que o arquivo de teste não poderia ter seus resultados comparados com outros trabalhos na literatura que o utilizam como conjunto de teste, mas sim como conjunto de validação.

Ressalta-se que no presente trabalho não haveria problema em utilizar o arquivo de teste do NSL-KDD como validação, pois foi criado um método próprio para extrair conexões de arquivos PCAP do CICIoT2023 e dos CSV do Kyoto 2006 +, podendo assim, construir uma base de testes autêntica.

4.2.6.4 Sobre o uso do CICIoT2023 e do Kyoto 2006 +

Por último, cita-se a limitação em analisar com mais critério o CICIoT2023, já que é um *dataset* com mais de 500GB e que contém as informações em PCAP exigindo processamento pelo *software* desenvolvido para se chegar as *features* do NSL-KDD.

Dessa forma, limitou-se o uso do *dataset*, selecionando os arquivos PCAP pelos critérios já explicitados e ainda realizando uma amostragem nos fluxos resultantes para criar um conjunto de teste com número reduzido de amostras, sendo essa última amostragem feita para controlar o tempo necessário da avaliação do conjunto de teste do CICIoT2023.

5 CONCLUSÃO E TRABALHOS FUTUROS

5.0.1 Visão Geral

O presente trabalho propôs mecanismos para manutenção constante de um IDS a partir de algoritmos de ML. Para isso, foi selecionado o *dataset* NSL-KDD, buscando realizar a manutenção de algoritmos de ML a partir da avaliação de diferentes modelos de ML, de técnicas de extração de *features* e mesclagem de dados, e da análise de desempenho dos algoritmos no arquivo de teste do NSL-KDD e em outros *benchmarks*.

Para alcançar o objetivo proposto, foi necessário primeiramente:

- Propor um software capaz de extrair as *features* de rede do NSL-KDD em tempo quase real ou a partir de um arquivo PCAP, o que foi fundamental para realizar a formatação de dados brutos de outros *benchmarks* em *features*. Tal software também permite a adição de tráfego de rede (em tempo real ou de outros *benchmarks*) nos modelos de ML para melhorar suas predições.
- Propor um algoritmo de amostragem para manutenção de algoritmos de ML a partir do incremento do conjunto de treino com amostras de outros *benchmarks*.
- Propor um algoritmo de ML, com desempenho ótimo no arquivo de teste do NSL-KDD, utilizando-se somente das 26 *features* extraídas pelo *software*. Para isso, treina-se algoritmos com o *dataset* de treino do NSL-KDD, explorando diferentes modelos de ML, técnicas de extração de *features* e técnicas de mesclagem de dados.

Ao cumprir os objetivos mencionados, avaliou-se o desempenho do algoritmo de ML ótimo mencionado no CICIoT2023, buscando verificar a necessidade de manutenção do IDS, o que pôde ser feito usando o *software* proposto para formatação dos dados em *features* e dos algoritmos para mesclagem (adição) de dados e extração de *features* estudados.

5.0.2 Trabalhos Futuros

- Implementar no software proposto *features* provenientes de outros *benchmarks*.
- Avaliar o uso de diferentes algoritmos de regressão como forma de adicionar as *features* faltantes do NSL-KDD no Kyoto 2006 + *dataset*, permitindo um treinamento com mais *features* em comum entre os *datasets*.
- Colocar *softwares* com detecção de assinatura para operar ao lado do algoritmo de ML, criando um algoritmo híbrido (por assinatura + anomalia), ou até mesmo, incluir o resultado da classificação do IDS por assinatura como *feature* no algoritmo de ML, como é feito no Kyoto 2006 +.

- Gerar um *dataset* mesclado com amostras do CICIoT2023 baseando-se no mesmo algoritmo (de amostragem) proposto para adição de amostras do Kyoto 2006 + e avaliar o desempenho de modelos treinados de ML e DL em ambientes IoT em tempo real ou em outras amostras (aquelas não adicionadas ao *dataset* de treino) do CICIoT2023.
- Avaliar a utilização de redes *Generative Adversarial Networks* (GAN) para gerar amostras artificiais no *dataset* e ajudar no refinamento dos classificadores, conforme o trabalho proposto por [39].
- Adaptar o *framework* proposto por [32] para permitir a predição de tráfego e o incremento da base de treinamento de forma constante, utilizando-se do *software* proposto para extrair as *features* e baseando-se no algoritmo de adição de amostras proposto.

REFERÊNCIAS BIBLIOGRÁFICAS

1 AHMAD, Z.; KHAN, A. S.; SHIANG, C. W.; ABDULLAH, J.; AHMAD, F. Network intrusion detection system: A systematic study of machine learning and deep learning approaches. *Transactions on Emerging Telecommunications Technologies*, Wiley Online Library, v. 32, n. 1, p. e4150, 2021.

2 GUPTA, S. C. *MLOps: Machine Learning Life Cycle*. 2023. Disponivel em: https://www.ml4devs.com/articles/mlops-machine-learning-life-cycle/, Acesso em: 26 julho de 2024.

3 AHMAD, M. S.; SHAH, S. M. Mitigating malicious insider attacks in the internet of things using supervised machine learning techniques. *Scalable Computing: Practice and Experience*, v. 22, n. 1, p. 13–28, 2021.

4 NETO, E. C. P.; DADKHAH, S.; FERREIRA, R.; ZOHOURIAN, A.; LU, R.; GHORBANI, A. A. Ciciot2023: A real-time dataset and benchmark for large-scale attacks in iot environment. *Sensors*, MDPI, v. 23, n. 13, p. 5941, 2023.

5 OVERVIEW, D. *dialog(1) - Linux man page*. 2024. Disponivel em: https://docs.docker.com/get-started/overview/, Acesso em: 07 Maio de 2024.

6 KHUSHAKTOV, M. F. *Introduction Random Forest Classification By Example*. 2024. Disponivel em: https://medium.com/@mrmaster907/introduction-random-forest-classification-by-example-6983d95c7b91, Acesso em: 07 Maio de 2024.

7 LEE, J.; KANG, J.-H.; JUN, S.; LIM, H.; JANG, D.; PARK, S. Ensemble modeling for sustainable technology transfer. *Sustainability*, MDPI, v. 10, n. 7, p. 2278, 2018.

8 CHOWDHURY, R.; SEN, S.; ROY, A.; SAHA, B. An optimal feature based network intrusion detection system using bagging ensemble method for real-time traffic analysis. *Multimedia Tools and Applications*, Springer, v. 81, n. 28, p. 41225–41247, 2022.

9 AKIBA, T.; SANO, S.; YANASE, T.; OHTA, T.; KOYAMA, M. Optuna: A next-generation hyperparameter optimization framework. In: *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. [S.l.: s.n.], 2019. p. 2623–2631.

10 DEVELOPERS, S. learn. *Cross-validation: evaluating estimator performance*. 2024. Disponivel em: https://scikit-learn.org/stable/modules/cross_validation.html, Acesso em: 08 Maio de 2024.

11 XU, W.; JANG-JACCARD, J.; SINGH, A.; WEI, Y.; SABRINA, F. Improving performance of autoencoder-based network anomaly detection on nsl-kdd dataset. *IEEE Access*, IEEE, v. 9, p. 140136–140146, 2021.

12 THIRIMANNE, S. P.; JAYAWARDANA, L.; YASAKETHU, L.; LIYANAARACHCHI, P.; HEWAGE, C. Deep neural network based real-time intrusion detection system. *SN Computer Science*, Springer, v. 3, n. 2, p. 145, 2022.

13 PARAMPOTTUPADAM, S.; MOLDOVANN, A.-N. Cloud-based real-time network intrusion detection using deep learning. In: IEEE. *2018 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*. [S.l.], 2018. p. 1–8.

14 RATHORE, M. M.; PAUL, A.; AHMAD, A.; RHO, S.; IMRAN, M.; GUIZANI, M. Hadoop based real-time intrusion detection for high-speed networks. In: IEEE. *2016 IEEE global communications conference (GLOBECOM)*. [S.l.], 2016. p. 1–6.

15 PIRES, L. A. d. S.; OLIVEIRA, F. B. d. O.; NZE, G. D. A.; GONCALVES, V. P.; MENDONCA, F. L. L. Real-time feature extraction software for machine learning models focused on cyber attacks. *Revista Ibérica de Sistemas e Tecnologias de Informação (RISTI)*, RISTI, Porto, Portugal, E66, n. 1, p. 93–105, 2024.

16 CYBERSECURITY, C. I. for. *CIC IoT dataset 2023*. 2023. Disponivel em: https://www.unb.ca/cic/datasets/iotdataset-2023.html, Acesso em: 21 Maio de 2024.

17 GHURAB, M.; GAPHARI, G.; ALSHAMI, F.; ALSHAMY, R.; OTHMAN, S. A detailed analysis of benchmark datasets for network intrusion detection system. *Asian Journal of Research in Computer Science*, v. 7, n. 4, p. 14–33, 2021.

18 SONG, J.; TAKAKURA, H.; OKABE, Y. *Description of Kyoto University Benchmark Data*. 2024. Disponivel em: https://www.takakura.com/Kyoto_data/BenchmarkData-Description-v5.pdf, Acesso em: 16 de setembro de 2024.

19 RESENDE, P. A. A.; DRUMMOND, A. C. A survey of random forest based methods for intrusion detection systems. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 51, n. 3, p. 1–36, 2018.

20 MUSA, U. S.; CHHABRA, M.; ALI, A.; KAUR, M. Intrusion detection system using machine learning techniques: A review. In: IEEE. *2020 international conference on smart electronics and communication (ICOSEC)*. [S.l.], 2020. p. 149–155.

21 TAVALLAEE, M.; BAGHERI, E.; LU, W.; GHORBANI, A. A. A detailed analysis of the kdd cup 99 data set. In: IEEE. *2009 IEEE symposium on computational intelligence for security and defense applications*. [S.l.], 2009. p. 1–6.

22 ANJUM, N.; LATIF, Z.; LEE, C.; SHOUKAT, I. A.; IQBAL, U. Mind: A multi-source data fusion scheme for intrusion detection in networks. *Sensors*, MDPI, v. 21, n. 14, p. 4941, 2021.

23 MIT, L. L. *1999 DARPA Intrusion Detection Evaluation Dataset*. 2024. Disponivel em: https://www.ll.mit.edu/r-d/datasets/1999-darpa-intrusion-detection-evaluation-dataset, Acesso em: 07 Maio de 2024.

24 MONGODB, I. *MongoDB Basics*. 2024. Disponivel em: https://www.mongodb.com/resources/products/fundamentals/basics, Acesso em: 07 Maio de 2024.

25 IBM, I. *What is Redis?* 2024. Disponivel em: https://www.ibm.com/topics/redis#: :text= What%20is%20Redis%3F,cache%20or%20quickresponse%20database., Acesso em: 07 Maio de 2024.

26 DICKEY, T. E. *dialog(1) - Linux man page*. 2024. Disponivel em: https://linux.die.net/man/1/dialog, Acesso em: 07 Maio de 2024.

27 ALPAYDIN, E. *Introduction to machine learning*. [S.l.]: MIT press, 2020.

28 DEVELOPERS, S. learn. *Decision Trees*. 2024. Disponivel em: https://scikit-learn.org/stable/modules/tree.html#minimal-cost-complexity-pruning, Acesso em: 07 Maio de 2024.

29 DEVELOPERS, S. learn. *Ensembles: Gradient boosting, random forests, bagging, voting, stacking*. 2024. Disponivel em: https://scikit-learn.org/stable/modules/ensemble.html#random-forests-and-other-randomized-tree-ensembles, Acesso em: 07 Maio de 2024.

30 SAHA, S. *Feature-selection-using-wrapper-based-Moth-Flame-algorithm*. 2021. Disponivel em: https://github.com/Soumyajit-Saha/Feature-selection-using-wrapper-based-Moth-Flame-algorithm/tree/main, Acesso em: 08 Maio de 2024.

31 KIMINEWT. *pyshark*. 2023. Disponivel em: https://github.com/KimiNewt/pyshark/, Acesso em: 16 de setembro de 2024.

32 OLIVEIRA, F. B. de. *Framework para detecção de ataques DoS em dispositivos IoT, utilizando abordagens de aprendizado de máquinas*. Tese (Master's Thesis) — University of Brasilia, Campus Universitário Darcy Ribeiro - Cep: 70.910-900, 2023.

33 MEIDAN, Y.; BOHADANA, M.; MATHOV, Y.; MIRSKY, Y.; SHABTAI, A.; BREITENBACHER, D.; ELOVICI, Y. N-baiot—network-based detection of iot botnet attacks using deep autoencoders. *IEEE Pervasive Computing*, IEEE, v. 17, n. 3, p. 12–22, 2018.

34 CHAUHAN, H.; KUMAR, V.; PUNDIR, S.; PILLI, E. S. A comparative study of classification techniques for intrusion detection. In: IEEE. *2013 International Symposium on Computational and Business Intelligence*. [S.l.], 2013. p. 40–43.

35 DRAPER-GIL, G.; LASHKARI, A. H.; MAMUN, M. S. I.; GHORBANI, A. A. Characterization of encrypted and vpn traffic using time-related. In: *Proceedings of the 2nd international conference on information systems security and privacy (ICISSP)*. [S.l.: s.n.], 2016. p. 407–414.

36 QOSIENT. *openargus*. 2023. Disponivel em: https://github.com/openargus/argus, Acesso em: 08 Maio de 2024.

37 PAXSON, V. Bro: a system for detecting network intruders in real-time. *Computer networks*, Elsevier, v. 31, n. 23-24, p. 2435–2463, 1999.

38 DEVELOPERS, S. learn. *Post pruning decision trees with cost complexity pruning*. 2024. Disponivel em: https://scikit-learn.org/stable/auto_examples/tree/plot_cost_complexity_pruning.html, Acesso em: 21 Maio de 2024.

39 XU, W.; JANG-JACCARD, J.; LIU, T.; SABRINA, F.; KWAK, J. Improved bidirectional gan-based approach for network intrusion detection using one-class classifier. *Computers*, MDPI, v. 11, n. 6, p. 85, 2022.