Análise de Tráfego de Botnets com Machine Learning

Gabriel Manhães de Souza

PROJETO FINAL DE GRADUAÇÃO

CENTRO TÉCNICO CIENTÍFICO - CTC

DEPARTAMENTO DE INFORMÁTICA

Curso de Graduação em Engenharia da Computação

Rio de Janeiro, Dezembro de 2022

Gabriel Manhães de Souza

Análise de Tráfego de Botnets com Machine Learning

Relatório de Projeto Final, apresentado ao programa Engenharia de Computação da PUC-Rio como requisito parcial para a obtenção do título de Engenheiro de Computação.

Orientador: Hélio Côrtes Vieira Lopes

Coorientador: Anderson Oliveira da Silva

Rio de Janeiro

Dezembro de 2022.

Agradecimentos

Aos meus pais, Samuel e Adriana, e meu irmão Samuel, por toda luta, paciência, preocupação, amor, carinho e por nunca medir esforços, para garantir que eu chegasse onde estou hoje. Aos meus amigos Yago, Marina, Suzana e Lucca, pelos conselhos, distrações, encorajamento e por entenderem os momentos de ausência durante este ano. À minha namorada, Sara, pelo carinho, apoio e companheirismo incondicional. Sem seu encorajamento esse projeto não teria sido possível. Aos meus amigos Lucas, Miguel, Pedro e João Gabriel, vocês foram fundamentais para minha formação. A todos os professores que contribuíram com minha trajetória acadêmica, especialmente a Hélio e Anderson, responsáveis pela orientação do meu projeto. A Pontifícia, pelo acolhimento e pela oportunidade dada para mim e tantos outros alunos.

Resumo

Manhães, Gabriel. Lopes, Hélio. Análise de Tráfego de Botnets com Machine Learning. Rio de Janeiro, 2022. Número de páginas 30. Relatório de Projeto Final – Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

O objetivo deste estudo é produzir de forma satisfatória um modelo de detecção de tráfego de botnet, utilizando técnicas de pré processamento, engenharia de atributos e otimização especificamente para o dataset CTU-13, que conta com amostras reais de tráfego de malwares conhecidos, além de tráfego normal e background. A metodologia, de forma resumida, foi: remoção dos dados inválidos através de imputação simples; encoding; agrupamento em janelas de 5 segundos, endereço de origem e label; separação em treino e teste; treino do modelo; avaliação dos resultados da predição. Para a avaliação final, foi utilizado: Autoencoder, Stacked Autoencoders, Variational Autoencoder, Random Forest e KNN. Todos modelos tiveram boas métricas, sendo o melhor deles o Random Forest, com o f1-score igual a 0.96.

Palavras-chave

Botnet; Autoencoder; Random Forest; KNN; Machine Learning; Engenharia de Features.

Abstract

Manhães, Gabriel. Lopes, Hélio. Botnet Traffic Analysis using Machine Learning. Rio de Janeiro, 2022. Number of pages 30. Final Project Report – Informatics Department. Pontifícia Universidade Católica do Rio de Janeiro.

The objective of this study is to satisfactorily produce a botnet traffic detection model, using pre-processing, feature engineering and optimization techniques specifically for the CTU-13 dataset, which has real samples of malware related traffic in addition to normal and background traffic. The methodology, in short, was: removal of invalid data through simple imputation; encoding; grouping in 5 second windows, source address and label; evaluation of prediction results. For the final evaluation, the following were used: Autoencoder, Stacked Autoencoders, Variational Autoencoder, Random Forest and KNN. All models showed good metrics, and the best results were from Random Forest, with a 0.96 f1-score.

Keywords

Botnet; Autoencoder; Random Forest; KNN; Machine Learning; Feature Engineering.

SUMÁRIO

1. INTRODUÇÃO

Botnets tornaram-se uma grande ameaça à segurança e estabilidade da internet [1]. Tradicionalmente, a detecção de tráfego de botnet conta com métodos baseados em assinatura, que contam com regras e padrões predefinidos para identificar ameaças conhecidas. No entanto, esses métodos são limitados em sua capacidade de detectar ameaças novas e em evolução.

Para lidar com essas limitações, muitas organizações estão recorrendo a algoritmos de aprendizado de máquina para detecção de tráfego de botnet [2]. Esses algoritmos são treinados em grandes conjuntos de dados de tráfego de rede rotulado, permitindo que eles aprendam e se adaptem automaticamente a padrões complexos nos dados. Isso permite que eles identifiquem padrões sutis e diferenciados que podem não ser detectados por outros métodos, tornando-os mais eficazes na detecção de ameaças novas e em evolução.

Além disso, apesar dos avanços consistentes na prevenção a ataques, sempre há a necessidade de pesquisa: a ofuscação de payloads e assinaturas fazem com que se torne cada vez mais difícil a detecção de botnets.

Neste projeto, exploraremos o uso do aprendizado de máquina para detectar o tráfego de botnets, mais especificamente no dataset CTU-13 [3].

O código-fonte para o projeto pode ser consultado em *https://github.com/GabrielManhaes/ctu-13-machine-learning*.

2. BOTNETS

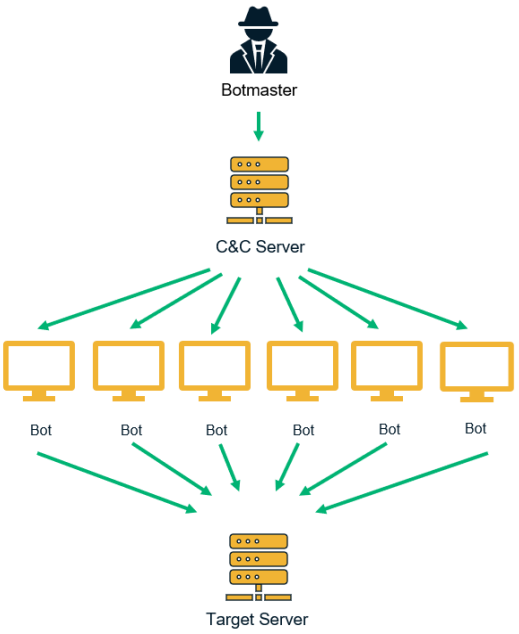


Figura 2.1: Diagrama demonstrando a hierarquia de uma botnet, com um servidor de comando e controle, botmaster, bots e vítima. Referência: *https://sectigostore.com/blog/botnet-attacks-what-is-a-botnet-how-does-it-work/*

Antes de partirmos para uma explicação do conjunto de dados utilizado, é importante definir do que se trata uma botnet. Uma botnet é uma rede de computadores infectados que estão sendo controlados por terceiros sem o conhecimento de seus proprietários. Esses computadores, referidos como bots, geralmente são comprometidos com malwares que permitem ao invasor controlá-los remotamente. O invasor pode usar as botnets para executar várias tarefas, como enviar e-mails de spam, lançar ataques distribuídos de negação de serviço (DDoS) ou roubar informações pessoais.

As redes de bots podem ser muito grandes, algumas consistindo em centenas de milhares ou até milhões de bots. Isso os torna uma ferramenta poderosa para cibercriminosos, pois o combinado de todos os computadores infectados pode ser utilizado para lançar ataques em larga escala.

Além disso, botnets são especialmente difíceis de detectar e interromper, pois os computadores infectados geralmente estão espalhados pela internet e são controlados pelo invasor de um local remoto.

3. CTU-13

3.1. SUMÁRIO

O dataset CTU-13 foi criado pela Czech Technical University em Praga, República Tcheca com o objetivo de oferecer um conjunto de dados que contenha uma grande captura real de tráfego de botnet, tráfego normal/não-malicioso e tráfego de background (não rotulado). O dataset conta com 13 capturas distintas, que iremos nos referir como cenários. Em cada cenário, um malware distinto foi executado e seu tráfego bidirecional capturado.

Em cada cenário, podemos observar que os malwares têm características únicas e executam tarefas distintas, como demonstra a tabela abaixo fornecida pelos autores:

Table 2 – Characteristics of the botnet scenarios. (CF: ClickFraud, PS: Port Scan, FF: FastFlux, US: Compiled and controlled by us.)

Id	IRC	SPAM	CF	PS	DDoS	FF	P2P	US	HTTP	Note
1	√	√	√							
2	√	√	√							
3	√			√				√		
4	√				√			√		UDP and ICMP DDoS.
5		√		√					√	Scan web proxies.
6				√						Proprietary C&C. RDP.
7									√	Chinese hosts.
8				√						Proprietary C&C. Net-BIOS, STUN.
9	√	√	√	√						
10	√				√			√		UDP DDoS.
11	√				√			√		ICMP DDoS.
12							√			Synchronization.
13		√		√					√	Captcha. Web mail.

Figura 3.1.1: Tabela demonstrativa das atividades do tráfego capturado em cada um dos 13 cenários do dataset CTU-13. Referência: *https://www.stratosphereips.org/datasets-ctu13*

Além disso, os autores também fornecem mais dados relevantes sobre a natureza dos dados, como as tabelas abaixo:

Id	Duration(hrs)	# Packets	#NetFlows	Size	Bot	#Bots
1	6.15	71,971,482	2,824,637	52GB	Neris	1
2	4.21	71,851,300	1,808,123	60GB	Neris	1
3	66.85	167,730,395	4,710,639	121GB	Rbot	1
4	4.21	62,089,135	1,121,077	53GB	Rbot	1
5	11.63	4,481,167	129,833	37.6GB	Virut	1
6	2.18	38,764,357	558,920	30GB	Menti	1
7	0.38	7,467,139	114,078	5.8GB	Sogou	1
8	19.5	155,207,799	2,954,231	123GB	Murlo	1
9	5.18	115,415,321	2,753,885	94GB	Neris	10
10	4.75	90,389,782	1,309,792	73GB	Rbot	10
11	0.26	6,337,202	107,252	5.2GB	Rbot	3
12	1.21	13,212,268	325,472	8.3GB	NSIS.ay	3
13	16.36	50,888,256	1,925,150	34GB	Virut	1

Figura 3.1.2: Tabela demonstrativa do tráfego capturado em cada um dos 13 cenários do dataset CTU-13, mostrando a duração em horas, número de pacotes, número de amostras, tamanho em GB, malware e número de máquinas infectadas. Referência: *https://www.stratosphereips.org/datasets-ctu13*

Scen.	Total Flows	Botnet Flows	Normal Flows	C&C Flows	Background Flows
1	2,824,636	39,933(1.41%)	30,387(1.07%)	1,026(0.03%)	2,753,290(97.47%)
2	1,808,122	18,839(1.04%)	9,120(0.5%)	2,102(0.11%)	1,778,061(98.33%)
3	4,710,638	26,759(0.56%)	116,887(2.48%)	63(0.001%)	4,566,929(96.94%)
4	1,121,076	1,719(0.15%)	25,268(2.25%)	49(0.004%)	1,094,040(97.58%)
5	129,832	695(0.53%)	4,679(3.6%)	206(1.15%)	124,252(95.7%)
6	558,919	4,431(0.79%)	7,494(1.34%)	199(0.03%)	546,795(97.83%)
7	114,077	37(0.03%)	1,677(1.47%)	26(0.02%)	112,337(98.47%)
8	2,954,230	5,052(0.17%)	72,822(2.46%)	1,074(2.4%)	2,875,282(97.32%)
9	2,753,884	179,880(6.5%)	43,340(1.57%)	5,099(0.18%)	2,525,565(91.7%)
10	1,309,791	106,315(8.11%)	15,847(1.2%)	37(0.002%)	1,187,592(90.67%)
11	107,251	8,161(7.6%)	2,718(2.53%)	3(0.002%)	96,369(89.85%)
12	325,471	2,143(0.65%)	7,628(2.34%)	25(0.007%)	315,675(96.99%)
13	1,925,149	38,791(2.01%)	31,939(1.65%)	1,202(0.06%)	1,853,217(96.26%)

Figura 3.1.3: Tabela demonstrativa do tráfego capturado em cada um dos 13 cenários do dataset CTU-13, mostrando o número total de amostras, o número de amostras de botnet, número de amostras normais, número de amostras de comando e controle e número de amostras de background. Referência: *https://www.stratosphereips.org/datasets-ctu13*

3.2. FEATURES

O dataset utilizado conta com 15 features, contendo features numéricas, categóricas, rótulo e índice, sendo elas:

- StartTime: índice do dataset, é a data de início de uma amostra (*datetime*)
- Dur: duração da conexão em segundos (*float*)
- Proto: protocolo da conexão (*string*)
- SrcAddr/DstAddr: endereço de origem e destino, respectivamente (*endereço IP*)
- Sport/Dport: porta de origem e destino, respectivamente (*integer*)
- Dir: direção da conexão (*string*)
- State: estado da conexão (*string*)
- sTos/dTos: tipo de serviço de origem e destino, respectivamente (*float*)
- TotPkts: total de pacotes transmitidos durante a conexão (*float*)
- TotBytes: total de bytes transmitidos durante a conexão (*float*)
- SrcBytes: total de bytes transmitidos pelo endereço de origem durante a conexão (*float*)
- Label: rótulo da conexão (*string*)

Especificamente para as features Sport, Dport, State, sTos e dTos, foram encontrados valores NaN (*Not a Number*) e para as features Sport e Dport,

foram encontrados também valores em hexadecimal. Por isso, foi necessário o tratamento por imputação ou conversão, que será especificado no tópico 4.

3.3. TAMANHO DO DATASET

Apesar do dataset não conter uma quantidade grande de features, seu tamanho de 20,643,076 linhas contribui imensamente para a complexidade deste projeto.

Dado que todo o trabalho desenvolvido foi executado localmente em uma máquina com especificação AMD Ryzen 5 5600X 6-Core, AMD Radeon RX 6800 XT e 16GB RAM e, quando se trata de um dataset de mais de 20 milhões de linhas, uma tarefa simples como carregar o dataset em memória já leva alguns minutos. Então, desta forma, uma solução precisou ser desenvolvida de modo a possibilitar o uso do dataset no projeto e esta será discutida no tópico 3.4 e 6.

3.4. RÓTULO BACKGROUND

Convenientemente, o rótulo background se trata, na verdade, da ausência de um rótulo. Este foi extraído pelos autores do CTU-13 e, em sua origem, não continha um rótulo. Portanto, o rótulo background não deve ser utilizado para fins de treinamento ou predição, já que não podemos afirmar que uma amostra qualquer rotulada como background é ou não maliciosa.

Por conta disso, o caminho escolhido foi a eliminação das linhas rotuladas como background. E, por isso, acabamos por resolver parte do desafio acima, já que aproximadamente 90% do dataset era rotulado como background.

4. LIMPEZA DE DADOS

A limpeza de dados [4] se refere ao processo de preparação e pré-processamento dos dados para utilização em um modelo de aprendizado de máquina. Nesse caso, a limpeza foi feita com a imputação de valores NaN, conversão de valores em formatos inválidos e a simplificação dos rótulos.

4.1. VALORES NAN

Nas colunas Sport, Dport, State, sTos e dTos, a atribuição de substitutos para os valores NaN foi feito da seguinte forma: computamos a moda de cada coluna, e substituímos os valores que eram anteriormente NaN. A escolha foi feita a partir da experimentação, desde métodos mais simples como o escolhido, como métodos mais complexos/custosos, como KNN. Apesar dos métodos complexos funcionarem muito bem, a melhoras nos resultados não compensam seus custos computacionais elevados. Além disso, a natureza enviesada da distribuição de valores nas colunas contribuiu para a escolha da moda ao invés de média ou mediana.

4.2. TRATAMENTO DE HEXADECIMAIS

Nas colunas Sport e Dport, foi necessária a substituição de valores inválidos, já que algumas amostras apresentaram valores em string hexadecimal (0xabcd). Esta etapa é importante para eliminar a incompatibilidade dos dados de uma mesma coluna, neste caso, a existência de valores do tipo *string* e *integer* impedia o funcionamento de todos modelos utilizados. A correção foi feita através da função abaixo. Dessa forma, se a *string* resultante da conversão na base hexadecimal de um valor para *integer* for diferente da *string* original, isso significa que o valor é uma *string* hexadecimal, então a convertemos para *integer*. Se não, simplesmente convertemos para *integer*.

```
def replacePort(port):
  if str(int(str(port), 16)) != str(port):
      return int(str(port), 16)
  return int(str(port))
```

4.3. SIMPLIFICAÇÃO DE RÓTULOS

Na coluna Label, havia uma grande variedade de rótulos descritivos (“From-Normal-V48-Stribrek”, “From-Botnet-V48-UDP-DNS”) que aumentavam a legibilidade dos dados mas, em contrapartida, não podiam ser utilizados como rótulo em treinamento ou predição. Por isso, a seguinte função foi utilizada. Dessa forma, todo rótulo que contém a string “Botnet” corresponde a 1, caso contrário, 0.

```
def label_simple(label):
    if 'Botnet' in label:
      return 1
    else:
        return 0
```

5. ENCODING

Encoding [5] é um processo utilizado no aprendizado de máquina para converter dados em um formato que possa ser entendido e processado por um computador. Isso é necessário porque muitos algoritmos de aprendizado de máquina só são capazes de trabalhar com dados representados de forma numérica. Existem vários tipos diferentes de encoding que podem ser utilizados no aprendizado de máquina, incluindo one-hot encoding, label encoding e binary encoding.

O one-hot encoding [6] é um tipo comum de encoding utilizado para converter dados categóricos, como texto ou rótulos, em um formato numérico. No one-hot encoding, cada categoria é representada por um vetor binário, com 1 na posição correspondente à categoria e 0 em todas as outras posições. Por exemplo, se houver três categorias, cada categoria seria representada por um vetor com três elementos, como [1, 0, 0] para a primeira categoria, [0, 1, 0] para a segunda categoria e [0 , 0, 1] para a terceira categoria.

O label encoding [5] é outro tipo de encoding frequentemente implementado em aprendizado de máquina. Ao contrário do one-hot encoding, que cria um vetor binário para cada categoria, o label encoding atribui um número inteiro exclusivo a cada categoria. Por exemplo, se houver três categorias, elas podem ser representadas pelos inteiros 1, 2 e 3. O label encoding pode ser útil para algoritmos que podem trabalhar com dados ordinais, como árvores de decisão e regressão linear. No entanto, é importante observar que o label encoding pode introduzir um viés nos dados.

Para este projeto, o one-hot encoding foi amplamente utilizado nas colunas categóricas Proto, Dir, sTos, dTos e na versão categórica da coluna DstAddr, que será discutida posteriormente. Além disso, uma versão modificada do label encoding foi utilizada na coluna Label, como foi explicado acima.

5.1. ENDEREÇOS IP

Quando lidamos com endereços IPs em datasets, algum tipo de pré-processamento é necessário, já que a natureza desse dado não é numérica e nem categórica e não é apropriada como entrada de um modelo de aprendizado de máquina.

Nesse projeto, a abordagem escolhida foi a seguinte: gerar uma nova coluna categorizando os IPs em *A* (1.X.X.X até 126.X.X.X), *B* (128.X.X.X até 191.X.X.X), *C* (192.X.X.X até 223.X.X.X) ou *N/A*, para endereços IPv4, e *ipv6* para endereços IPv6; gerar uma segunda coluna que categoriza o IP em interno (10.0.0.0 até 10.255.255.255, 172.16.0.0 até 172.31.255.255, 192.168.0.0 até 192.168.255.255) ou externo (qualquer caso que não se encaixe nos acima). Dessa forma, resolvemos o desafio do uso dessa feature nos modelos de aprendizado, transformando-a, após o One-Hot Encoding, em duas features categóricas. Além disso, o treinamento certamente pode se beneficiar desses dados, sem que um viés seja introduzido no dataset, com o modelo aprendendo que um IP X está relacionado com o tráfego de botnet, o que seria extremamente prejudicial para os resultados.

6. FEATURE ENGINEERING

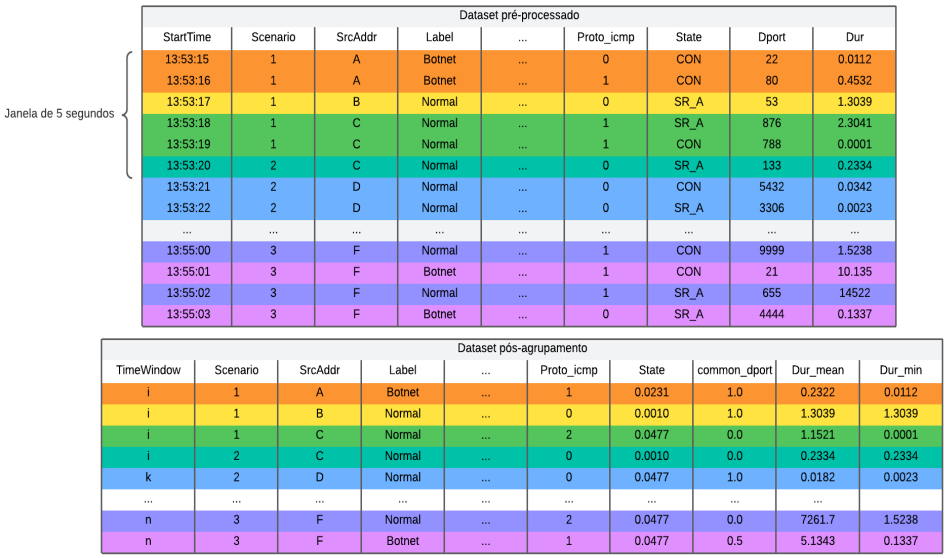


Figura 6.1: Tabelas demonstrativas do agrupamento realizado; as cores representam a correspondência entre n linhas da tabela superior e uma única linha na tabela inferior; para cada janela de cinco segundos, agrupamos pela tupla (Scenario, SrcAddr, Label) e computamos novas features.

6.1. SUMÁRIO

Feature engineering [16] é o processo de usar o conhecimento de domínio dos dados para criar recursos que fazem os algoritmos de aprendizado de máquina funcionarem melhor. Esses recursos são tipicamente novas features derivadas que são construídas a partir dos dados brutos e destinam-se a melhor representar o problema subjacente.

Ao selecionar e construir cuidadosamente seus recursos, é possível que o modelo passe a capturar padrões subjacentes nos dados com mais eficiência, levando a melhores previsões. Isso é especialmente importante para problemas do mundo real, nos quais os dados brutos podem não ser diretamente úteis para fazer previsões. No geral, feature engineering é uma etapa crucial no processo de aprendizado de máquina que pode ajudá-lo a criar modelos mais eficazes e obter informações mais profundas sobre seus dados.

Para esse projeto, a principal técnica utilizada foi a de agrupamento em tuplas (janela de tempo, cenário, endereço de origem, rótulo): com isso, temos

que para cada janela de cinco segundos, cada conjunto de *n* linhas em que cenário, endereço de origem e rótulo são iguais, geramos uma nova linha que representa o grupo anterior e, dessa forma, foi possível extrair características temporais relevantes do dataset e diminuir seu número de linhas, tornando o treinamento menos custoso e melhorando suas métricas.

6.2. NOVAS FEATURES

No dataset original, somente 15 colunas são capturadas e, ao final do processo de feature engineering, foram geradas 52 colunas. De forma geral, as novas colunas tinham como objetivo a identificação de padrões em features que anteriormente não contribuíam de forma significativa com o desempenho do modelo. Abaixo estão listadas todas features resultantes:

- *Dur*, *TotPkts*, *SrcBytes* e *TotBytes*:
    - média;
    - mínimo;
    - máximo;
    - mediana;
    - desvio padrão;
    - soma;
- Colunas *Dir_X, category_dstaddr_X, type_dstaddr_X, sTos_X, dTos_X* e *Proto_X*, derivadas do One-Hot Encoding das colunas Dir, DstAddr, sTos, dTos e Proto, respectivamente:
    - soma, dessa forma, temos o número de ocorrências de cada categoria de uma coluna;
- *DstAddr*, *Sport*, *Dport* e *State:*
    - Entropia, calculado como número de valores únicos dividido pelo número total de valores possíveis para cada coluna;
- *Sport* e *Dport*:
    - Porcentagem de portas comuns (SSH, FTP, Postgres, HTTP, etc.);
- *Min Start Time*:
    - Timestamp da primeira amostra do grupo, utilizada somente para ordenação;
- *Count*
    - Quantidade de linhas dentro do grupo;

7. SEPARAÇÃO EM TREINO, VALIDAÇÃO E TESTE

Uma das principais razões pelas quais precisamos separar os dados em fatias de treinamento e teste no aprendizado de máquina é evitar o overfitting [7]. O overfitting ocorre quando um modelo exibe ótimos resultados em um determinado conjunto de dados de treinamento e tem um desempenho ruim em dados novos e não vistos. Ao avaliar o desempenho de um modelo em um conjunto de dados de teste separado, podemos ter uma noção melhor de como o modelo irá generalizar para novos dados. Isso é importante porque o objetivo final de qualquer modelo de aprendizado de máquina é poder fazer previsões precisas sobre dados novos.

Tradicionalmente, o split do dataset é feito em proporções fixas, por volta de 70-80% dos dados para treino e 20-30% dos dados para teste. No entanto, para esse projeto, a abordagem escolhida foi a separação sugerida pelos autores do dataset [3]:

- Treino e validação: Cenários 3, 4, 5, 7, 10, 11, 12 e 13.
- Teste: Cenários 1, 2, 6, 8 e 9.

Dessa forma, além de garantirmos que o modelo não sofra com overfitting, ainda podemos testar quão bem o modelo consegue generalizar o problema quando nos deparamos com um bot inédito, dado que a separação dos cenários é feita de modo que não haja um bot presente em ambas fatias de treino e teste.

8. SCALING

Scaling [8] refere-se ao processo de transformar os valores das features em um dataset para que fiquem dentro de um intervalo controlado. Isso é importante porque muitos algoritmos de aprendizado de máquina usam cálculos baseados em distância, como o cálculo de distâncias euclidianas no processo de aprendizado. Se as features em um dataset não forem dimensionadas, o algoritmo poderá dar mais importância às features com valores de maior magnitude e menos importância às features com menor magnitude, o que pode gerar resultados indesejáveis. O scaling também ajuda a tornar o processo de treinamento mais eficiente, pois os algoritmos tendem a convergir mais rapidamente nos dados com scaling.

Nesse projeto, o *StandardScaler* da biblioteca scikit-learn gerou resultados suficientemente satisfatórios. Além disso, como algumas das abordagens testadas são focadas na detecção de anomalias, usar estratégias de scaling como min-max não contribuiu com o bom desempenho dos modelos.

9. MODELOS

9.1. AUTOENCODERS

Um autoencoder [9] é um tipo de rede neural artificial geralmente utilizada para aprendizado não supervisionado. Esta consiste em duas camadas: um encoder e um decoder. O encoder pega os dados de entrada e os mapeia para uma representação de dimensão inferior, conhecida como espaço latente, que, essencialmente, tende a capturar os recursos essenciais dos dados. O decoder então pega essa representação do espaço latente e a mapeia de volta ao espaço de entrada original, produzindo uma reconstrução da entrada original. O objetivo de um autoencoder é aprender uma representação dos dados de entrada que seja a mais precisa possível e, ao mesmo tempo, a mais compacta possível.

Uma das principais aplicações dos autoencoders é a redução de dimensionalidade. Ao treinar um autoencoder em um conjunto de dados, o encoder pode aprender a compactar os dados em um espaço latente de dimensão inferior, o que pode ser útil para visualizar os dados ou para acelerar as tarefas subsequentes de aprendizado de máquina. Autoencoders também podem ser usados para detecção de anomalias, onde o modelo é treinado em dados normais e então usado para sinalizar amostras que são significativamente diferentes dos dados normais.

Para os fins deste trabalho, os autoencoders discutidos aqui foram utilizados para detecção de anomalias. Isto é, os autoencoders foram treinados somente com amostras não-maliciosas e, na predição, foram introduzidas também amostras maliciosas. Desta forma, quando o erro de reconstrução é calculado, podemos definir um limiar em que classificamos satisfatoriamente as amostras em não-maliciosas/maliciosas.

Neste projeto, diferentes tipos de autoencoders foram implementados utilizando a biblioteca *Keras* e serão discutidos abaixo.

9.1.1. AUTOENCODER

A seguinte estrutura modificada [10] resultou em melhores predições. Ao invés de implementar um Encoder que mapeia diretamente para uma dimensão inferior, o Encoder abaixo aumenta a dimensão dos dados, para posteriormente mapeá-los para o espaço latente. Da mesma forma que o Decoder

implementado também aumenta a dimensão dos dados, para posteriormente mapeá-los de volta à mesma dimensão da entrada. A figura abaixo demonstra melhor o descrito acima.

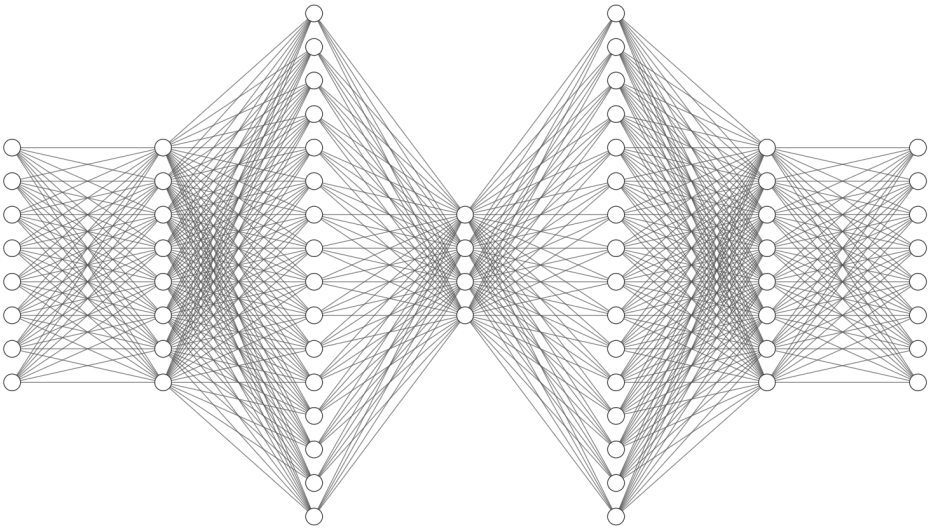


Figura 9.1: Representação simplificada do autoencoder implementado.
Referência: *https://alexlenail.me/NN-SVG/index.html*

Os seguintes hiperparâmetros foram utilizados:

- Encoder Neurons: 512, 512, 1024;
- Decoder Neurons: 1024, 512, 512;
- Latent Dimension: 128;
- Hidden Activation: LeakyReLU
- Output Activation: LeakyReLU
- Epochs: 1000;
- Batch Size: 128;
- Optimizer: Adam;
- Learning Rate: 1e-5;
- Decay: 1e-5;
- Loss: Mean Squared Error;

9.1.2. STACKED AUTOENCODERS

Como o nome já sugere, a implementação de Stacked Autoencoders [11] envolve o encadeamento de mais de um Autoencoder, dessa forma, possibilitamos o aprendizado de representações hierárquicas dos dados de entrada.

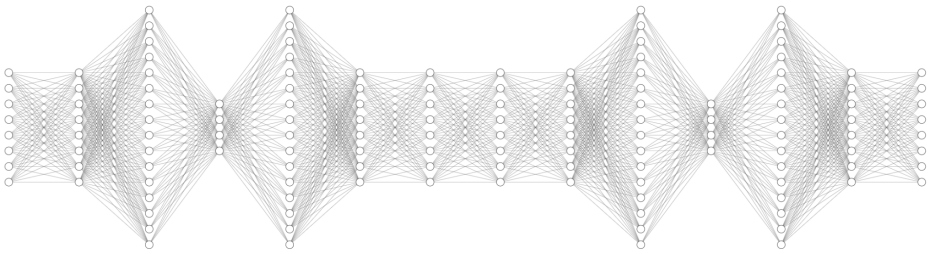


Figura 9.2: Representação simplificada do stacked autoencoder implementado. Na imagem, estão representados dois autoencoder idênticos a Figura 9.1, encadeados. Referência: *https://alexlenail.me/NN-SVG/index.html*

Os seguintes hiperparâmetros foram utilizados:

- Encoder Neurons: 512, 512, 1024;
- Decoder Neurons: 1024, 512, 512;
- Latent Dimension: 128;
- Hidden Activation: LeakyReLU
- Output Activation: LeakyReLU
- Epochs: 1000;
- Batch Size: 128;
- Optimizer: Adam;
- Learning Rate: 1e-5;
- Number of Autoencoders: 3;
- Decay: 1e-5;
- Loss: Mean Squared Error;

9.1.3. VARIATIONAL AUTOENCODER

Ao contrário de um autoencoder tradicional, que é treinado para representar os dados em um espaço latente de tamanho fixo, o variational autoencoder [12] é treinado para aprender uma distribuição probabilística sobre o espaço latente, que pode ser amostrado para gerar novas amostras de dados semelhantes ao conjunto de dados original. Isso se deve pela introdução de uma restrição no espaço latente, conhecida como restrição "variacional", que força a rede a aprender uma representação compacta e contínua dos dados. Isso permite que a rede gere amostras de dados diversas e realistas e também pode ser usada para tarefas como aprendizado não supervisionado e redução de dimensionalidade.

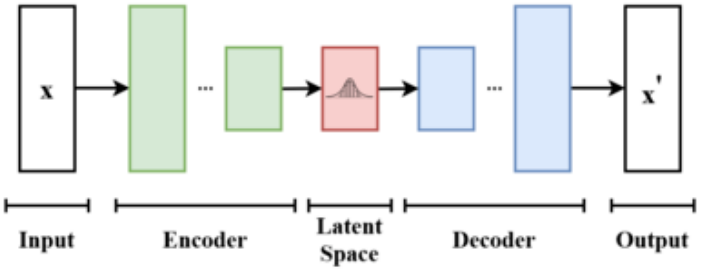


Figura 9.3: Representação simplificada de um variational autoencoder.
Referência: *https://en.wikipedia.org/wiki/Variational_autoencoder*

Os seguintes hiperparâmetros foram utilizados:

- Encoder Neurons: 512, 512, 1024;
- Decoder Neurons: 1024, 512, 512;
- Latent Dimension: 128;
- Hidden Activation: LeakyReLU
- Output Activation: LeakyReLU
- Epochs: 350;
- Batch Size: 128;
- Optimizer: Adam;
- Learning Rate: 1e-4;
- Decay: 1e-3;
- Loss: Mean Squared Error;

9.2. RANDOM FOREST

Random Forest [13] é um tipo de ensemble learning [14] para classificação e regressão. Ensemble Learning é um método onde múltiplos modelos são treinados e suas decisões são combinadas por algum meio para obter-se as predições finais. No caso do Random Forest, o ensemble aplicado consiste em uma combinação de árvores de decisão [15].

A árvore de decisão é um modelo de aprendizado de máquina que faz predições baseadas em um conjunto de regras extraídas do dataset. Cada nó da árvore representa uma feature e cada ramo representa uma decisão baseada no valor daquela feature. As folhas da árvore representam a predição do modelo.

Random Forest treina um grande número de árvores de decisão usando diferentes subconjuntos de dados e subconjuntos aleatórios de features. Assim,

cada árvore de decisão na floresta faz uma previsão e a previsão final é determinada pela combinação das previsões de todas as árvores.

Ao usar várias árvores de decisão, as Random Forests podem capturar uma gama mais ampla de padrões nos dados e fazer previsões mais precisas do que seria possível com uma única árvore de decisão. Além disso, usar subconjuntos aleatórios de dados e recursos pode reduzir o overfitting.

Com o dataset utilizado, o melhor número de estimadores encontrado foi 500.

9.3. KNN

K-nearest Neighbors (KNN) [17] é um algoritmo de aprendizado supervisionado para classificação e regressão. A ideia por trás do KNN é usar os rótulos conhecidos em dados do dataset para fazer previsões para novos pontos de dados. O algoritmo funciona encontrando os K pontos de dados no conjunto de treinamento que estão mais próximos do novo ponto de dados e usando os rótulos conhecidos para esses pontos de dados para fazer uma previsão.

Para encontrar os pontos de dados mais próximos, o KNN usa uma métrica de distância, nesse caso a distância euclidiana. A distância entre dois pontos de dados é calculada medindo a diferença entre os valores de cada característica dos pontos de dados.

Depois que os K pontos de dados mais próximos são encontrados, o algoritmo usa a maioria dos votos de seus rótulos para fazer uma previsão para o novo ponto de dados. Por exemplo, se K = 3 e dois dos pontos de dados mais próximos forem rotulados como "Botnet" e um for rotulado como "Normal", então a previsão para o novo ponto de dados seria "Botnet".

O KNN é um algoritmo simples e eficaz para muitos tipos de dados, mas pode ser computacionalmente caro e pode não ser adequado para conjuntos de dados muito grandes. Além disso, a escolha de K e a métrica de distância podem ter um impacto significativo no desempenho do algoritmo, portanto, para o dataset escolhido, o melhor valor de K encontrado foi 5.

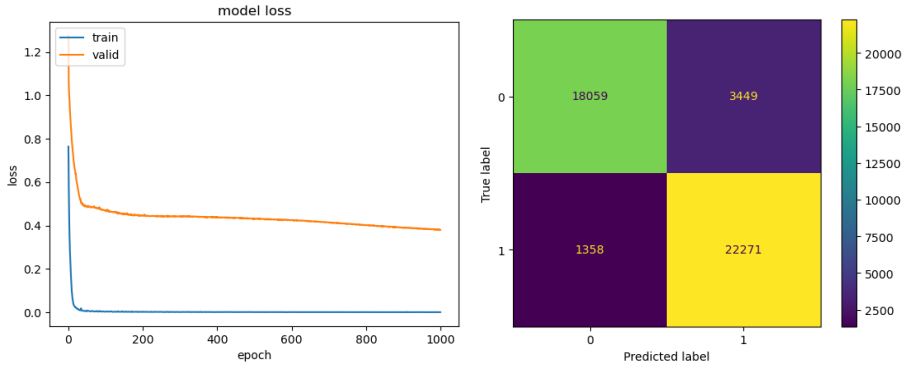
10. RESULTADOS

	Precision	Recall	F1-score	AUROC
AE	0.8659	**0.9425**	0.9026	0.8910
VAE	0.7923	0.8853	0.8362	0.8151
SAE	0.8286	0.9402	0.8809	0.8633
RF	**0.9923**	0.9393	**0.9651**	**0.9657**
KNN	0.9701	0.8470	0.9044	0.9092

Tabela 10.1: Métricas dos modelos, em negrito estão os maiores valores em cada coluna.

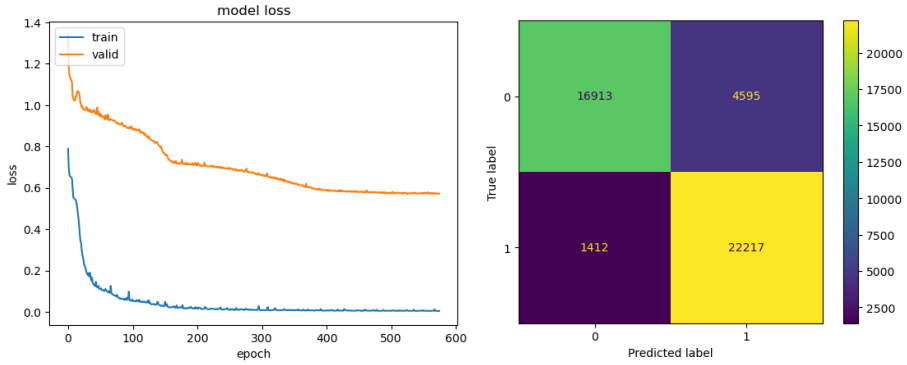
Como pode se observar acima, de modo geral, os modelos performaram relativamente bem. Além disso, apesar de representar um algoritmo mais simples, já era esperado que o Random Forest fosse performar melhor que os demais.

Isso se deve provavelmente pela natureza dos dados e, no geral, outras pesquisas no mesmo dataset mostram o mesmo comportamento. [3] [18] Além disso, o tuning de hiperparâmetros se mostrou como um grande desafio durante o desenvolvimento do projeto, o que pode ter contribuído com os resultados dos modelos VAE (Variational Autoencoder) e SAE (Stacked Autoencoders), inferiores ao AE (Autoencoder).



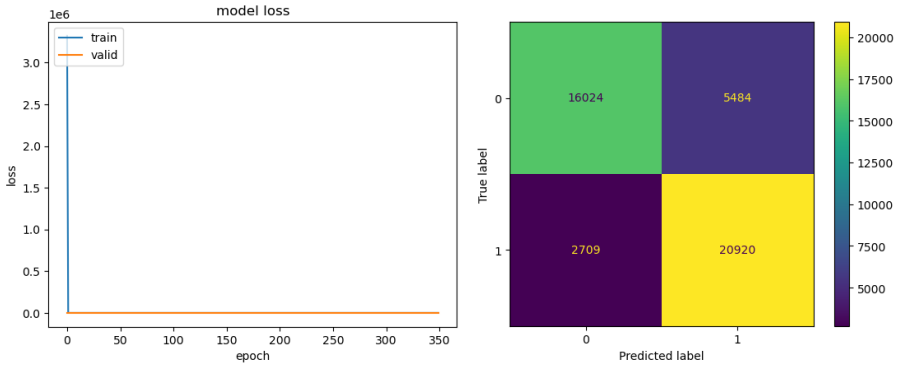
Figuras 10.1 (esquerda) e 10.2 (direita): Curva da função de loss e matriz de confusão, respectivamente (Autoencoder).

Com o gráfico de perda acima, podemos observar que houve aprendizado significativo, havendo apenas uma pequena diferença entre a perda de validação e treino. Além disso, com a matriz de confusão apresentada, concluímos que o modelo generaliza satisfatoriamente o dataset.



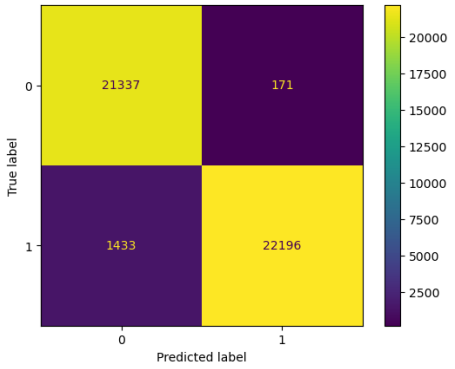
Figuras 10.3 (esquerda) e 10.4 (direita): Curva da função de loss e matriz de confusão, respectivamente (Stacked Autoencoders).

Com o gráfico de perda acima, podemos observar que houve aprendizado significativo, apesar do ruído apresentado durante o treino. No entanto, com a matriz de confusão apresentada, concluímos que o modelo também generaliza satisfatoriamente o dataset.

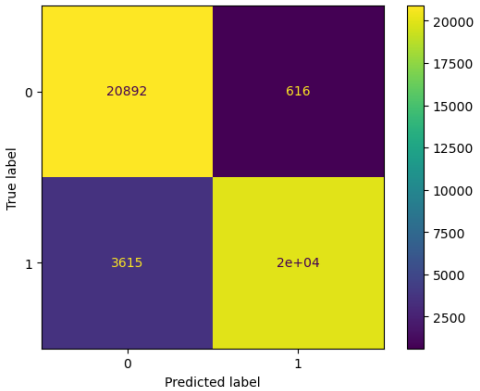


Figuras 10.5 (esquerda) e 10.6 (direita): Curva da função de loss e matriz de confusão, respectivamente (Variational Autoencoder).

Com o gráfico de perda acima, dada a magnitude da perda no início do treino, não há uma análise clara a se fazer aqui, mas há suspeitas de overfitting. Por conta do desafio de tuning de hiperparâmetros, esse foi o modelo mais afetado.



Figuras 10.7: Matriz de confusão (Random Forest).



Figuras 10.8: Matriz de confusão (KNN).

Com as matrizes acima, podemos notar a disparidade clara dos resultados obtidos com os autoencoders, com estes tendo uma quantidade muito inferior de falso-positivos, que é uma qualidade extremamente desejável de um sistema em produção.

Os modelos de aprendizado de máquina mostraram resultados promissores em termos de exatidão e precisão. O modelo de Random Forest foi capaz de aprender e fazer previsões efetivamente no conjunto de dados fornecido, atingindo uma pontuação de precisão de 99%. Isso indica que o modelo tem uma forte capacidade de generalizar para dados não vistos e pode ser útil em um cenário do mundo real. No entanto, mais experimentos e ajustes podem ser necessários para melhorar o desempenho do modelo e alcançar resultados ainda melhores, principalmente nos autoencoders implementados. No geral, os resultados sugerem que o uso de aprendizado de máquina nesse contexto tem o potencial de melhorar a precisão das previsões e aprimorar nossa compreensão dos dados subjacentes, quando comparado a métodos tradicionais de detecção de botnets.

Para trabalhos futuros, uma abordagem promissora seria o uso de Ensemble Learning, desta forma, poderíamos tirar proveito das qualidades das diversas abordagens.

11. REFERÊNCIAS BIBLIOGRÁFICAS

[1] Netscout. **“DDoS THREAT INTELLIGENCE REPORT”**, 2022.

[2] A. KARIM, R. B. SALLEH, M. SHIRAZ, S. A. A. SHAH, I. AWAN, N. B. ANUAR. **“Botnet detection techniques: Review, future trends, and issues”**. Journal of Zhejiang University SCIENCE C, vol. 15, no. 11, 2014.

[3] S. GARCIA, M. GRILL, J. STIBOREK, A. ZUNINO . **“An Empirical Comparison of Botnet Detection Methods”**, 2014.

[4] **“Guide To Data Cleaning: Definition, Benefits, Components, And How To Clean Your Data”**. Tableau, 2022. Disponível em: https://www.tableau.com/learn/articles/what-is-data-cleaning. Acesso em: 01 de Novembro de 2022.

[5] A. CAVIN. **“6 Ways to Encode Features for Machine Learning Algorithms”**. Towards Data Science, 2022. Disponível em: https://towardsdatascience.com/6-ways-to-encode-features-for-machine-learning-algorithms-21593f6238b0. Acesso em 09 de Novembro de 2022.

[6] J. BROWNLEE. **“Ordinal and One-Hot Encodings for Categorical Data”**. Machine Learning Mastery, 2020. Disponível em: https://machinelearningmastery.com/one-hot-encoding-for-categorical-data/. Acesso em 11 de Novembro de 2022.

[7] A. GILLIS. **“Data Splitting”**. Tech Target, 2022. Disponível em: https://www.techtarget.com/searchenterpriseai/definition/data-splitting. Acesso em 15 de Novembro de 2022.

[8] B. ROY. **“All about Feature Scaling”**. Towards Data Science, 2020. Disponível em: https://towardsdatascience.com/all-about-feature-scaling-bcc0ad75cb35. Acesso em 15 de Novembro de 2022.

[9] D. E. RUMELHART, G. E. HINTON, R. J. WILLIAMS. **“Parallel distributed processing: Explorations in the microstructure of cognition”**. MIT Press, Cambridge, EUA, 1986.

[10] Q. P. NGUYEN, K. W. LIM, D. M. DIVAKARAN, K. H. LOW, M. C. CHAN. **“GEE: A Gradient-based Explainable Variational Autoencoder for Network Anomaly Detection”**. National University of Singapore, Singapura, 2019.

[11] P. VINCENT, H. LAROCHELLE, I. LAJOIE, Y. BENGIO, P. A. MANZAGOL. **"Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion"**. Journal of Machine Learning Research 11, Canadá, 2010.

[12] D. P. KINGMA, M. WELLING. **"Auto-Encoding Variational Bayes"**. Universiteit van Amsterdam, Holanda, 2013.

[13] T. YIU. **"Understanding Random Forest"**. Towards Data Science, 2019. Disponível em: https://towardsdatascience.com/understanding-random-forest-58381e0602d2. Acesso em 27 de Novembro de 2022.

[14] J. BROWNLEE. **"A Gentle Introduction to Ensemble Learning Algorithms"**. Machine Learning Mastery, 2021. Disponível em: https://machinelearningmastery.com/tour-of-ensemble-learning-algorithms/. Acesso em 27 de Novembro de 2022.

[15] **"Decision Trees"**. SciKit Learn, 2022. Disponível em: https://scikit-learn.org/stable/modules/tree.html. Acesso em 28 de Novembro de 2022.

[16] H. PATEL. **"What is Feature Engineering — Importance, Tools and Techniques for Machine Learning"**. Towards Data Science, 2021. Disponível em: https://towardsdatascience.com/what-is-feature-engineering-importance-tools-and-techniques-for-machine-learning-2080b0269f10. Acesso em 01 de Dezembro de 2022.

[17] A. CHRISTOPHER. **"K-Nearest Neighbor"**. Medium, 2021. Disponível em: https://medium.com/swlh/k-nearest-neighbor-ca2593d7a3c4. Acesso em 02 de Dezembro de 2022.

[18] J. KIM, A. SIM, J. KIM, K. WU, J. HAHM. **"Improving Botnet Detection with Recurrent Neural Network and Transfer Learning"**. Lawrence Berkeley National Laboratory, Berkeley, EUA, 2021.