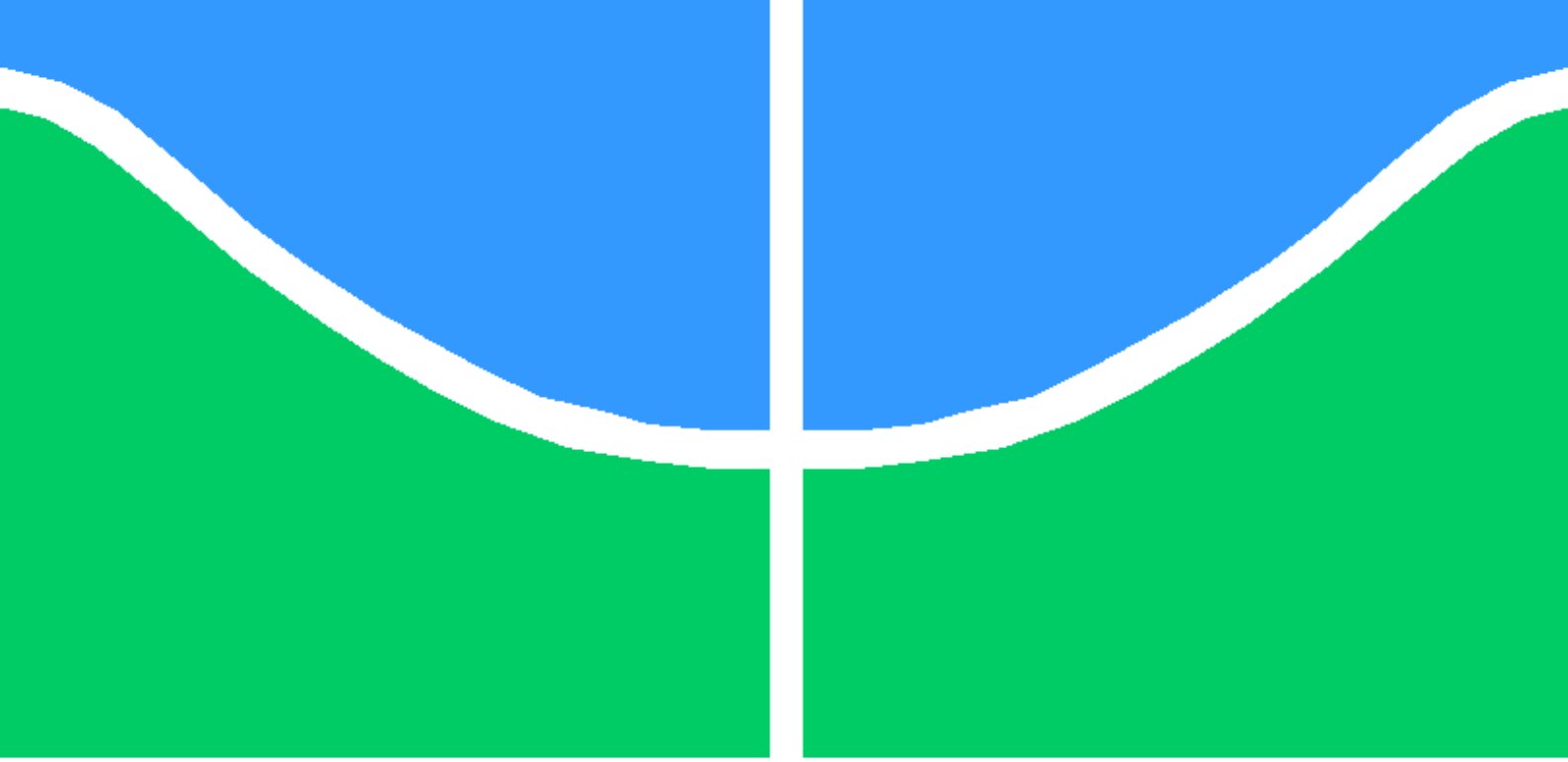


TRABALHO DE GRADUAÇÃO

ANÁLISE E DETECÇÃO AUTOMATIZADAS DE *WORMS* EM AMBIENTE CONTROLADO COM TÉCNICAS DE *MACHINE LEARNING*

Marco Túlio Campos de Oliveira

Rodrigo Lima Rocha

Brasília, Dezembro de 2017

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia

TRABALHO DE GRADUAÇÃO

ANÁLISE E DETECÇÃO AUTOMATIZADAS DE *WORMS* EM AMBIENTE CONTROLADO COM TÉCNICAS DE *MACHINE LEARNING*

Marco Túlio Campos de Oliveira

Rodrigo Lima Rocha

Relatório submetido ao Departamento de Engenharia Elétrica como requisito parcial para obtenção do grau de Engenheiro de Redes de Comunicação

Banca Examinadora

Prof. Flávio Elias Gomes de Deus, ENE/UnB _____
Orientador

Prof. Robson de Oliveira Albuquerque, ENE/UnB _____
Co-Orientador

Prof. Rafael Timóteo de Sousa Jr, ENE/UnB _____
Examinador interno

Dedicatórias

Dedico este trabalho a todos que me apoiaram nesses anos de estudos.

Marco Túlio Campos de Oliveira

Dedico este trabalho a Deus.

Rodrigo Lima Rocha

Agradecimentos

Agradeço a minha família e meus amigos por me ajudarem a fazer este trabalho. Agradeço também ao César Borges e aos professores Robson Albuquerque e Flávio Elias por nos orientar e ajudar a concluir este trabalho.

Marco Túlio Campos de Oliveira

Agradeço primeiramente a Deus por ter me dado forças para chegar até aqui, agradeço também aos meus familiares por todo apoio e suporte dado, também aos amigos que estiveram sempre me apoiando tanto em momentos de alegria quanto nos momentos difíceis. Por fim agradeço aos professores, Flávio Elias e Robson Albuquerque, e ao César Borges por todo apoio dado na realização deste trabalho.

Rodrigo Lima Rocha

RESUMO

A análise de *malwares* vem sendo feita por especialistas de maneira extensiva para combater a propagação em massa de tais *malwares*, além do roubo de dados importantes da vítima. Estas análises são feitas a partir dos binários do código malicioso, do comportamento durante sua execução, e suas ações tomadas na rede. A propagação de códigos maliciosos, normalmente, é significativamente mais rápida do que a análise manual do especialista, que é dificultada por inúmeras técnicas usadas por programadores de códigos maliciosos.

Neste trabalho, é apresentado um sistema *web* feito para auxiliar a análise manual de um especialista com o intuito de diminuir o tempo perdido em arquivos não maliciosos que apresentam características de *malware*. O sistema *web* apresentado utiliza sete algoritmos de aprendizado de máquina para fazer predições objetivando criar um processo de análise mais rápido, que, consequentemente, agilizará na criação de soluções para infecções de *malwares*, do tipo *worms*. Foi modificada a ferramente *Cuckoo Sandbox* para que as predições funcionem juntos com os seus relatórios padrões.

ABSTRACT

Malware analysis have been done by specialists extensively in order to combat the widespread of such malwares ans also the theft of important data of victims. These analysis are made from the binaries of malicious code, the behavior during its execution, and actions taken through the network. The spread of malicious code is usually significantly quicker than the manual analysis made by specialist, this analysis is hindered by numerous techniques used by malicious code programmers.

In this project, a web system designed to aid the manual analysis of a specialist in order to reduce the time spent in non-maicious files which have malware features is presented. The presented web system uses seven machine learning algorithms to make predictions in order to create a more responsive analysis process that will consequently speed up in the creation of solutions against worm kind of malwares. The tool Cuckoo Sandbox was modified in order to have the predictions working with the default reports of it.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

Acrônimos

ANN	Redes Neurais Artificiais
API	Application Program Interface
AV	Antivírus
CART	Árvores de Classificação e Regressão
CSS	Cascading Style Sheets
CSV	Comma-Separated Values
CWD	Cuckoo Working Directory
DLL	Dynamic-link library
EPO	Entry-Point Obscuring
HTML	HyperText Markup Language
IDS	Intrusion Detection System
IM	Instant Message
IP	Internet Protocol
IPDS	Intrusion Prevenction and Detection System
IRC	Internet Relay Chat
JSON	JavaScript Object Notation
KNN	K Vizinhos Mais Próximos
LDA	Análise do Discriminante Linear
LR	Regressão Logística
MCDM	Multi-Criteria Decision-Making
ML	Aprendizado de Máquina
MLP	Perceptron de Múltiplas Camadas
MLE	Maximum-Likelihood Estimation
NB	Naive Bayes
NIST	National Institute of Standards and Technology
RAM	Random Access Memory
P2P	Peer-to-Peer
PCAP	Packet Capture
PDF	Portable Document Format
PHP	Hypertext Preprocessor
PID	Process Identification

Acrônimos

SMB	Server Message Block
SO	Sistema Operacional
SP	Service Pack
SQL	Structured Query Language
SSD	Solid-State Drive
SVM	Support Vector Machine
TCE	Taxa de Classificações Erradas
TCP	Transport Control Protocol
TFN	Taxa de Falsos Negativos
TFP	Taxa de Falsos Positivos
UDP	User Datagram Protocol
USB	Universal Serial Bus
UPX	Ultimate Packer for Executables
URL	Uniform Resource Locator
VMT	Venus Malware Trap
VB	Visual Basic
VM	Máquina Virtual
VT	VirusTotal
XSS	Cross-Site Scripting
WSGI	Web Server Gateway Interface

Capítulo 1

Introdução

Com o decorrer do desenvolvimento de *softwares* e da computação em geral, tanto programas maliciosos quanto de intenções legítimas se tornaram mais complexos, pesados, e com múltiplas funções. Isto quer dizer que assim como um simples programa de troca de mensagens instantâneas evoluiu para receber e enviar imagens, arquivos de áudio e de vídeo, um *malware* também acompanhou este desenvolvimento. Para isto, ferramentas de detecção, prevenção e remoção de códigos maliciosos precisam estar atualizados e prontos para quaisquer novas ameaças.

Aycock (2006) relata em seu livro que, em tempos antigos, as necessidades das pessoas eram simples: comida, água, abrigo, e, ocasionalmente, a chance de propagar a espécie. Nossas necessidades básicas não mudaram, mas o jeito de realizá-las sim. Comida é comprada em lojas que são alimentadas por redes de abastecimento com sistemas de inventário computadorizados; água é dispensada por sistemas de água controlados por computadores; móveis para residências novas vêm de fornecedores com redes de fornecimento controladas por computadores, e casas antigas são compradas e vendidas por corretores de imóveis com computadores. Todos esses sistemas são controlados por computadores, e estes gerenciam as transações financeiras que pagam por isto tudo.

Tendo em vista o desenvolvimento da civilização como um todo e o advento dessa evolução em relação aos perigos de softwares maliciosos, faz-se necessário uma maior qualidade na análise desses artefatos, tanto no que diz respeito aos seus binários, quanto ao seu comportamento em si. Técnicas de análise estática e dinâmica são usadas para obter informações que ajudem a reconhecer *exploits*[1], e possibilitem criar defesas contra o tipo analisado. Infelizmente, técnicas como estas necessitam de conhecimento aprofundado e se tornam impossíveis para análise de uma grande quantidade de artefatos, pois exigem bastante tempo para uma análise individual.

Por outro lado, técnicas que visam automatizar essa operação vêm sendo desenvolvidas, tais como a *sandbox*, mostrada na seção 3.2 deste projeto, que aborda a ferramenta *Cuckoo*. Este traba-

[1] *Exploit*: pode ser um pedaço de código ou de dados malformados que fazem com que um *software* comporte de maneira diferente da pretendida pelo desenvolvedor do programa (ELISAN, 2015).

lho visa utilizar técnicas de análise dinâmica, estática, e de rede, para gerar relatórios detalhados a fim de, por meio destes, encontrar padrões que possam ser utilizados em detecções automatizadas. Existem trabalhos , como por exemplo o C. A. B. d. Andrade et al. (2013), com o enfoque em diferenciar *malwares* de arquivos benignos ao usar *sandboxes* e aprendizado de máquinas a partir, apenas, da análise comportamental do programa.

Este projeto visa aproveitar de técnicas já conhecidas para alterar o funcionamento comum da *sandbox*, de maneira que se torne possível a predição de um dado arquivo uma vez submetido nesta.

1.1 Definição do Problema

Informações sigilosas, pessoais ou não, são armazenadas geralmente em dispositivos eletrônicos, que podem, de alguma forma, ser comprometidos por arquivos, programas ou ações maliciosas provocadas por pessoas ou grupos mal intencionados, assim como a tecnologia, que vem crescendo rapidamente na última década. No entanto, medidas de segurança não acompanham a velocidade das técnicas usadas para ataques.

1.1.1 Crescimento do número de *Malwares*

Com o passar do tempo, o número de arquivos maliciosos vem se tornando cada vez maior. Eles são normalmente usados para roubar ou destruir informações privadas, ou até mesmo com a intenção de 'sequestrar' as informações de um dispositivo. Em um mundo em que a população quase em sua totalidade possui algum tipo de computador, a crescente desses códigos maliciosos se torna mais assustadora. Como pode ser visto na figura 1.1, notícia dada pela AVTEST (2017), a quantidade de *malwares* vem aumentando quase que exponencialmente nos últimos anos.

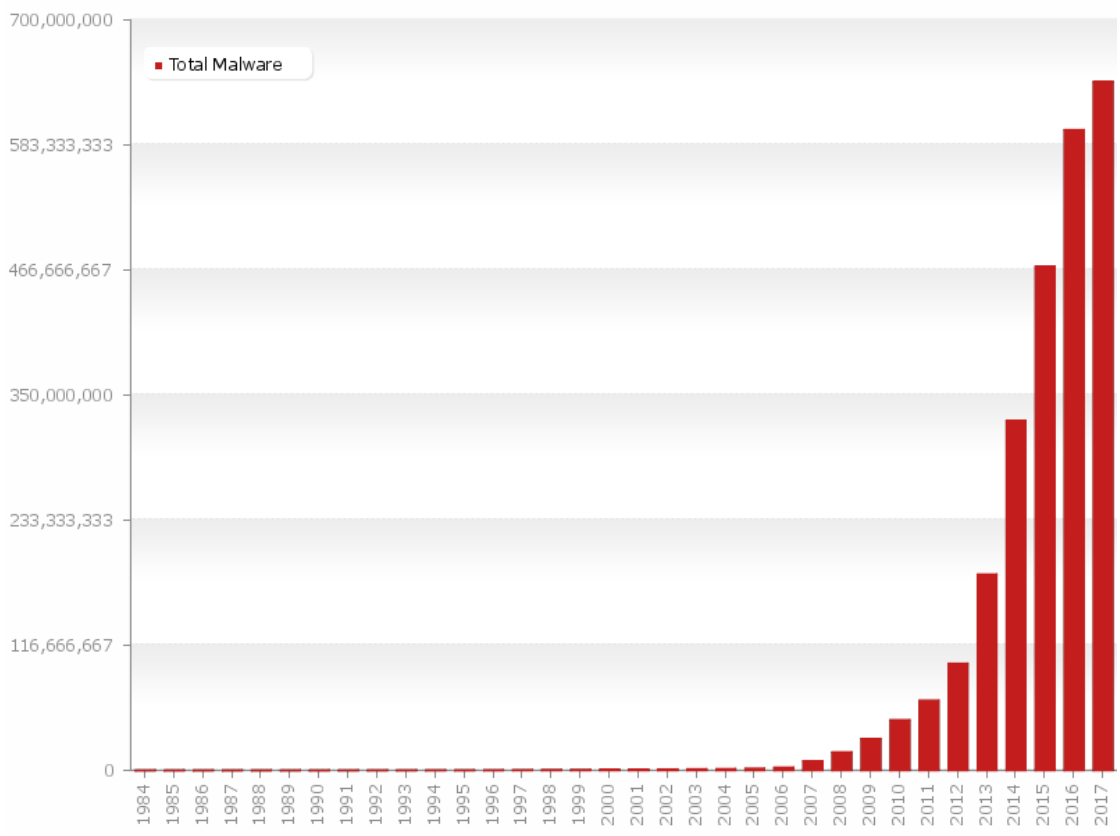


Figura 1.1: Crescimento de malwares (AVTEST, 2017).

1.1.2 Assinatura do *Malware*

Um grande problema enfrentado pelos sistemas de detecção atuais, os chamados antivírus, é a necessidade de uma assinatura do arquivo, algo que o identifique pelas características de seus binários. Então uma vez conhecida a assinatura de um determinado código malicioso, ele é facilmente detectado. No entanto, isso implica em outras dificuldades, como a necessidade de um grande banco de dados para o armazenamento dessas assinaturas já conhecidas, o que não resolve o problema de vulnerabilidades *zero-day*[2], pois ainda não há registros no banco de dados.

Outro método usado por estes programas é o da análise heurística. A heurística é uma tecnologia de detecção de vírus, que não podem ser encontrados no banco de dados do antivírus. Esta técnica permite a detecção de objetos suspeitos de estarem infectados por um *malware* desconhecido ou uma nova modificação de um outro previamente registrado (KASPERSKY-LAB, 2013). Apesar da técnica heurística ter, frequentemente, resultados positivos ao identificar códigos maliciosos desconhecidos, ela ainda pode ignorar *malwares* únicos e novos (SIKORSKI; HONIG, 2012).

[2]Vulnerabilidades *Zero-day* ou *zero-days*: são vulnerabilidades que foram detectadas mas ainda são desconhecidas para desenvolvedor do *software* e para a indústria de segurança da informação (ELISAN, 2015).

1.2 Objetivos

Este trabalho visa detectar *worms* automaticamente e agilizar a análise dos *malwares* utilizando de técnicas de análise dinâmica e estática além de algoritmos de aprendizado de máquina, para que um analista possa alocar mais recursos e, consequentemente, tempo em artefatos que se mostrem suspeitos, garantindo uma maior eficiência no tempo utilizado para análise.

1.2.1 Objetivos Específicos

Este projeto tem como objetivo além de automatizar a detecção de um *worm*, gerar também relatórios com conteúdos detalhados tanto da análise dinâmica quanto estática do artefato, com a finalidade de ajudar um especialista a analisar de maneira mais eficiente artefatos suspeitos. Estes relatórios contêm dados relevantes, tais como:

- Uso da rede, por meio de portas, protocolos, *Uniform Resource Locators* (URLs) e *Internet Protocols* (IPs);
- Criação/remoção/modificação de arquivos no computador infectado;
- Quaisquer peculiaridades nos registros do Windows;
- Uso das diversas *Application Programming Interface* (APIs) do sistema operacional;
- Resultados de diversos antivírus dados pelo VirusTotal[3];
- As decisões tomadas pelos diversos algoritmos de aprendizado de máquina com suas chances de achar que uma amostra é ou não maligna.

Estes relatórios serão mostrados em uma interface *web* de fácil uso com o intuito, também, de facilitar a análise feita por um terceiro, seja ele um especialista ou não.

1.3 Justificativa

Com a grande quantidade de *malwares* sendo criados dia a dia e com a dificuldade de antivírus detectarem arquivos maliciosos que não possuem assinaturas, embora já se trabalhe com heurística e outras formas de garantir uma maior precisão, ainda assim é difícil determinar se um novo artefato (*zero-day*) é malicioso, sendo necessárias outras formas de detecção de códigos maliciosos.

Existem também outras maneiras de analisar um arquivo malicioso. Uma forma bastante utilizada são as *sandboxes*, porém estas não determinam automaticamente a procedência de um dado artefato. Em suma, são utilizadas para gerar relatórios de análises dinâmicas, principalmente, cabendo assim ao especialista identificar ações e características suspeitas no artefato.

[3]Uma descrição mais detalhada do VirusTotal pode ser vista na seção 3.5.

Utilizar funcionalidades de uma *sandbox* e acrescentar uma maneira de automatizar o processo de detecção possibilita reduzir o tempo gasto para o estudo e pesquisa de novos *malwares* e torna possível também a pessoas que não possuam expertise neste assunto a capacidade de identificar se um determinado artefato é malicioso ou não.

1.4 Estrutura do Trabalho

No Capítulo 2, Fundamentação Teórica, serão abordados os principais fundamentos utilizados nesse projeto, incluindo aqueles que dizem respeito à análise dinâmica e estática, tipos de *malwares* e algoritmos de aprendizagem de máquina.

No Capítulo 3, primeiramente serão descritos os métodos e posteriormente explicadas as ferramentas utilizadas para a execução dos processos propostos.

No Capítulo 4, serão descritas as implementações feitas para a solução deste trabalho, além de identificar as modificações na ferramenta utilizada.

No Capítulo 5, poderão ser encontrados os valores para as diversas métricas de desempenho usadas nos algoritmos de aprendizado de máquina.

Por fim, no Capítulo 6, mostrará a conclusão que os autores deste projeto chegaram e o que pode ser feito para melhorar os resultados ou conseguir analisar em outros ambientes.

Capítulo 2

Fundamentação Teórica

Para uma análise condicente, um analista deve saber os metodos que serão utilizados ao longo da análise do binário. Além disso, é importante saber as diferenças dos *malwares* assim como as técnicas empregadas pelos desenvolvedores de *malwares* para dificultar dita análise.

Outro fator importante, que um analista precisa saber ao utilizar o sistema explicado no capitulo 4, é o funcionamento dos algoritmos de aprendizado de máquina assim como suas métricas de desempenho.

2.1 Análise de *Malware*

É possível analisar um *malware* tanto pelos seus binários quanto pelo seu comportamento, sendo que ambas as análises possuem prós e contras que serão discutidos posteriormente. Além disso, um arquivo malicioso pode ter diversas categorias e, em sua maioria, o comportamento é uma composição ou modificação dessas categorias.

2.1.1 Análise Estática

Análise estática é um processo que analisa o código ou a estrutura do programa para determinar suas funções, diferentemente da análise dinâmica na qual se faz necessária a execução do referido código.

Primeiramente, é importante aproveitar o fato de que antivírus possuem assinaturas e heurísticas distintas, portanto, um primeiro passo seria a análise do *malware* em diversos antivírus. Uma ferramenta bastante utilizada com esse propósito é o VirusTotal (VIRUSTOTAL, 2017), é gratuito e analisa arquivos em mais de 60 bases de antivírus diferentes.

Uma abordagem muito utilizada é a de analisar um arquivo a fim de procurar por *strings* que sejam suspeitas, abordagem que pode parecer simples, mas muitas vezes auxilia a encontrar padrões de funcionamento ou até mesmo funcionalidades importantes. Por exemplo, ao encontrar *strings* referentes a alguma URL, ou ainda funções de sistema conhecidas. Infelizmente, esse método

apresenta dificuldades, pois muitos atacantes ofuscam seus executáveis, tornando a análise mais complexa. Por isso, existem algumas abordagens específicas para esse tipo de arquivo.

Outro ponto de vista que pode ser utilizado durante a análise de um *malware* é verificar as bibliotecas utilizadas, visto que não há intenção de reimplementar funcionalidades do sistemas já dispostas. Normalmente, essas bibliotecas aparecem no cabeçalho de cada programa, mas é evidente que existem diversas maneiras de se utilizar dessas funcionalidades, umas mais simples de detecção do que outras. O *Windows*, por exemplo, permite a utilização de bibliotecas não listadas no cabeçalho durante a execução do programa, o que dificulta a análise.

Métodos de análise estática efetivos exigem, muitas vezes, conhecimento aprofundado no que diz respeito à engenharia reversa e conhecimento específico de sistemas operacionais, como *Windows*, alvo da maioria dos ataques (SIKORSKI; HONIG, 2012).

2.1.2 Análise Dinâmica

Para que seja possível determinar o comportamento preciso de um arquivo malicioso, deve-se analisá-lo dinamicamente, pelo simples fato de que, embora características da análise estática (*strings* ou informações de cabeçalho) possam dar muitas informações sobre a provável ação daquele *malware*, é com a análise dinâmica que se determina o que efetivamente o arquivo malicioso pode fazer.

De todo modo, é de extrema importância que seja preparado um ambiente seguro para uma análise de *malware*, tendo em vista que não se quer pôr em risco a integridade de sistemas em produção. Frequentemente, a análise é feita em máquinas virtuais, porquanto o ambiente se torna mais flexível e seguro. Por outro lado, como será tratado na seção 2.1.4, existem *malwares* que conseguem detectar o uso de máquinas virtuais e se comportam diferente do que fariam em máquinas reais, impossibilitando uma análise precisa.

Análise dinâmica utiliza, na maioria das vezes, um *debugger*[1] para traçar os passos do arquivo executado, o qual trará informações que seriam dificilmente percebidas por outro método, e, em geral, complementar outros tipos de análise (SIKORSKI; HONIG, 2012).

2.1.3 Taxonomia de *Malwares*

Nos últimos anos, aumentou-se muito a complexidade de arquivos maliciosos, tornando sua detecção e descrição mais complexa. Hoje, dificilmente é possível determinar um *malware* como um tipo específico, pois ele "não se encaixa cuidadosamente em um tipo de categoria. O atacante não escreve o *malware* para seguir a um tipo de classe por si só, ou para seguir uma descrição especifica de uma classe." (ELISAN, 2015). Por isso, um *malware* pode ter mais de uma classe com características diferentes, mas ainda distinguíveis.

[1] *Debugger*: Programa usado para fazer depuração normalmente com o objetivo de encontrar erros.

Não há uma definição aceita universalmente para termos como vírus e *worm*, muito menos uma definição regulamentada de taxonomia, apesar de ocasionais tentativas de impor um formalismo matemático em *malwares* (AYCOCK, 2006). Por exemplo, alguns autores podem citar *worms* e vírus como sinônimos (SIKORSKI; HONIG, 2012), enquanto outros separam estes em categorias diferentes (AYCOCK, 2006), e há ainda aqueles que não classificam vírus como um *malware* específico (ELISAN, 2015).

A. *Worm*

Worm é um programa capaz de se propagar automaticamente pelas redes, enviando cópias de si mesmo de computador para computador. Diferente do vírus, o *worm* não se propaga por meio da inclusão de cópias de si mesmo em outros programas ou arquivos, mas sim pela execução direta de suas cópias ou pela exploração automática de vulnerabilidades existentes em programas instalados em computadores. *Worms* são notadamente responsáveis por consumir muitos recursos, devido à grande quantidade de cópias de si mesmo que costumam propagar e, como consequência, podem afetar o desempenho de redes e a utilização dos computadores (CERT-BR, 2017).

Um *worm* pode ter subclassificações como: *worm* de mensagem instantânea (IM), *worm* de *Internet Relay Chat* (IRC), *worm* de compartilhamento de arquivos, entre outros (ELISAN, 2015).

B. Vírus

Vírus é um programa ou parte de um programa de computador, normalmente malicioso, que se propaga inserindo cópias de si mesmo e se tornando parte de outros programas e arquivos. Para que possa se tornar ativo e dar continuidade ao processo de infecção, o vírus depende da execução do programa ou arquivo hospedeiro, ou seja, para que o seu computador seja infectado, é preciso que um programa já infectado seja executado (CERT-BR, 2017).

Um modo de entender a diferença entre vírus e *worm* é que o primeiro varre o computador infectado para atingir novos alvos, enquanto o segundo varre a rede para procurar novos dispositivos para serem infectados.

C. Cavalo de Troia ou *Trojan Horse*

O cavalo de Troia, além de executar as funções para as quais foi aparentemente projetado, executa ainda outras funções, normalmente maliciosas, e sem o conhecimento do usuário. *Trojans* podem ser instalados por atacantes que, após invadirem um computador, alteram programas já existentes para desempenhar as funções originais e as ações maliciosas (CERT-BR, 2017).

Cavalos de Troia podem ter inúmeras finalidades (todas maliciosas). Deste modo, um *trojan* pode fazer do computador infectado de *proxy* para o uso do atacante, enquanto outro cria um *backdoor*, e até um terceiro que pode simplesmente apagar arquivos importantes do dispositivo.

D. *Ransomware*

Ransomware é um programa malicioso que bloqueia dados, acesso a sistemas ou recursos. Estes dados são mantidos como "reféns", a não ser que a vítima pague pelo resgate (ELISAN, 2015) (*ransom* em inglês). Os resgates, muitas vezes, são feitos em *cryptomoedas*[2].

Para a extorsão virtual, este *malware* usa técnicas como cifragem de dados, destruição de dados ou negação de acesso ao usuário. A primeira técnica cifra arquivos do usuário e fornece a chave, se e somente se a vítima pagar pelo resgate no tempo estipulado pelo atacante. No entanto, na segunda técnica, o usuário infectado pode ter os arquivos deletados caso o pagamento não seja feito a tempo. No último exemplo, o usuário perde acesso ao sistema operacional, impossibilitando-o de efetuar o *login* na máquina (ELISAN, 2015).

Segundo o relatório "State of Malware", do *Malwarebytes LABS*, a quantidade de *ransomwares* subiu cerca de 267% entre janeiro e novembro de 2016 (MALWAREBYTES-LABS, 2017).

E. *Backdoor*

Backdoor, do inglês, significa porta dos fundos. *Backdoor* é um programa que permite o retorno de um invasor a um computador comprometido, por meio da inclusão de serviços criados ou modificados para este fim. Após incluído, o *backdoor* é usado para assegurar o acesso futuro ao computador atacado, permitindo que ele seja acessado remotamente, sem que haja necessidade de repetir os métodos utilizados na realização da invasão ou infecção e, na maioria dos casos, sem que seja notado (CERT-BR, 2017).

F. *Bot* e *Botnet*

Bot é um programa que dispõe de mecanismos de comunicação com o invasor para permitir que ele seja controlado remotamente (CERT-BR, 2017). Este programa utiliza os métodos de propagação de um *worm*, para assim infectar o máximo possível de dispositivos. Um grupo de *bots* é chamado de *botnet*, isto é, uma rede de *bots*. Um sistema infectado por este tipo de *malware* também é conhecido na literatura como zumbi, pois quando "compromissado, pode ser usado pelo atacante para uma variedade de tarefas, sem o conhecimento do dono legítimo" (AYCOCK, 2006).

A comunicação entre o invasor e o computador infectado pelo *bot* pode ocorrer via canais de IRC, servidores *web* e redes do tipo *Peer-to-Peer* (P2P), entre outros meios. Ao se comunicar, o invasor pode enviar instruções para que ações maliciosas sejam executadas, como desferir ataques, furtar dados do computador infectado e enviar spam (CERT-BR, 2017).

G. *Spyware*

Spyware é um programa que coleta informação de um computador e a envia para o atacante. A informação exata que um *spyware* coleta pode variar, porém, pode conter dados de potencial valor, como:

[2] *Cryptomoeda*: recurso digital que fornece um meio de troca descentralizada usando criptografia para aumentar a segurança da transação(KNUTSON et al., 2016), além de evitar que a moeda seja rastreada.

- Par usuário e senha;
- Endereços de e-mail;
- Dados bancários e de cartões de crédito;
- Licença de *softwares*;
- Conversas em redes sociais;
- Histórico de navegadores;
- Entre outros.

Este tipo de programa também pode ser legítimo, com o objetivo de verificar se outras pessoas o estão utilizando de modo abusivo ou não autorizado. Isto é, quando o usuário tem consentimento do que este programa faz (CERT-BR, 2017).

H. Rootkit

Rootkit é um conjunto de programas e técnicas que permite esconder e assegurar a presença de um invasor ou de outro código malicioso em um computador comprometido (CERT-BR, 2017). Apagar trechos de *logs* do sistema, usar um *backdoor*, varrer e monitorar a rede para novos ataques, entre outras são técnicas comuns que um atacante usa em conjunto com um *rootkit* (SIKORSKI; HONIG, 2012; CERT-BR, 2017).

Após uma invasão bem sucedida, os *rootkits* eram instalados para manter o acesso privilegiado, sem precisar recorrer aos mesmo métodos utilizados na invasão, e para esconder suas atividades da vítima. Atualmente, *rootkits* têm sido também utilizados e incorporados por outros códigos maliciosos para ficarem ocultos e não serem detectados (CERT-BR, 2017).

2.1.4 Dificuldade de Detecção

Da mesma forma que os "defensores" se preocupam com novas maneiras de detectar e combater um *malware*, os atacantes preocupam-se cada vez mais em como não ser identificados, criando métodos que dificultam a detecção de um ataque baseados em ferramentas específicas. Em suma, como Sikorski e Honig (2012) dizem, "análise de *malware* é como um jogo de gato-e-rato. Conforme novas técnicas de análises são desenvolvidas, autores de *malware* respondem com novas técnicas para dificultar as análises". Dentre estes métodos, estão:

- *Entry-Point Obscuring* (EPO), em português, ofuscação do ponto de entrada. O ponto de entrada é um ponteiro para a localização da primeira instrução de um executável, ou seja, onde o programa formalmente começa (ELISAN, 2015). No entanto, arquivos infectados por um *malware* podem ter mais de um ponto de entrada, pois um *malware* cria o seu ponto de entrada dentro de um programa benigno[3]. O EPO consiste em apontar o ponto de entrada do *malware* para um código benigno, dificultando desta maneira as verificações de antivírus, *disassembler* e *debuggers*.

[3]Programa benigno: É um programa com funcionalidades legítimas que não produz ações maliciosas.

- Cifragem é usada em programas benignos para evitar pirataria e cópias com *crack*[4], mas também é utlizada por atacantes para proteger seu código de verificações por assinatura. Este processo faz com que a cifragem e decifragem sejam diferentes a cada infecção; no entanto, o *engine* que cifra e decifra se mantém constante (ELISAN, 2015). significa dizer que os produtos de segurança podem atuar neste ponto, gerando a assinatura a partir deste momento (depois da decifragem e antes da cifragem).

- Polimorfismo é uma evolução da cifragem anterior. Neste, tem-se o *mutation engine* que faz parte do código malicioso e, basicamente, altera o código de outra aplicação sem mudar suas funções (ELISAN, 2015), ou seja, o código do *engine* que cifra e decifra pode ser mudado sem alterar sua funcionalidade. Dessa forma, gera múltiplas assinaturas para cada infecção, exatamente no ponto que é estático na cifragem.

- Metamorfismo é o "contra-ataque" dos atacantes quando as empresas fabricantes de antivírus começaram a combater o polimorfismo. Este combate era feito por meio da análise do código na memória dos dispositivos infectados, que ainda era o mesmo em todas as infecções. Neste método, o *mutation engine* agora pode simplesmente mudar todo o código do *malware*. Desta forma, cada infeção é diferente da anterior, tanto no disco quanto na memória de um dispositivo infectado (ELISAN, 2015).

- Atacantes usam ferramentas de *Anti-Reversing* para, normalmente, atrasar o processo de descompilar ou de *disassembly*. Dentre estas ferramentas, têm-se os empacotadores de código (por exemplo o *Ultimate Packer for Executables* (UPX)).

- *Anti-Debugging* tem como objetivo principal enganar o analisador a seguir o fluxo de execução que não leva a conclusão alguma (ELISAN, 2015). Esta técnica não impede o *debugging*, mas atrasa o processo de remediação.

- *Anti-sandboxing* verifica se o *malware* está em uma máquina virtual, pode ser um sinal de *sandboxing*. Há também a verificação de arquivos e ferramentas dentro do sistema, que podem indicar se o dispositivo infectado está sendo monitorado por ferramentas de *sandboxing*. Caso qualquer uma dessas verificações tenha resultado positivo, o programa deixará de executar suas funções maliciosas, ou até mesmo não fará nada quando executado (ELISAN, 2015).

- Trava de Ambiente ou *Enviroment Lock*, em inglês, é a técnica de fazer o código malicioso funcionar exclusivamente em um único ambiente (ELISAN, 2015). Para isto, um atacante programa seu código para capturar dados únicos do dispositivo e então o *malware* poderá fazer a verificação. Se a verificação for falha, o código malicioso não é executado, dificultando com sucesso a análise do *malware* em qualquer outro dispositivo que não o paciente zero[5].

- Anti-Varredura de Antivírus (*Anti-AV Scan*) é a técnica em que o *malware* novo, que ainda não foi catalogado no banco de dados do antivírus, adiciona-se à lista de programas isentos

[4] *Crack*: Um *software* com *crack* tem funções indesejadas pelo usuário removidas, como a prevenção de cópias.

[5] Paciente Zero: O primeiro código ou dispositivo a ser infectado por algum tipo de *malware*.

de varreduras de segurança dos antivírus (ELISAN, 2015). Um outro método mais agressivo é o de desligar funcionalidades do AV; no entanto, este é mais perceptível a um usuário leigo.

- Proteção de Comportamento de Rede é quando um código malicioso toma atitudes para que seus rastros na rede não sejam facilmente percebidos. As técnicas mais comuns usadas por atacantes são a mudança de domínio constante (*Domains Fluxing*), domínios que respondem por IPs que mudam constantemente (*Fast Fluxing* ou *IP Fluxing*) e abuso de tráfego de dados de serviços legítimos (i.e IM, Redes Sociais, P2P) (ELISAN, 2015).

2.1.4.1 Sistema de Detecção e Prevenção de Intrusão (IDPS)

Além disso, uma ferramenta amplamente usada em redes privadas é o IDPS (do inglês, *Intrusion Detection and Prevention System*) "que tipicamente grava informações relacionadas a eventos na rede, notifica o administrador da rede de importantes eventos observados e então produz relatórios. Muitos IDPS podem também responder a ameaças detectadas tentando prevenir um ataque"(KAREN; PETER, 2007). Isto quer dizer que um equipamento como este precisa estar constantemente monitorando uma rede para gerar estatísticas de seu uso e então possivelmente detectar atividade suspeita, criando assim um *overhead* no ambiente.

2.1.4.2 *Sandbox versus Malware*

Apesar da técnica de *sandboxing* ser bem detalhada com relação à análise estática, dinâmica, de rede e de memória, ainda há alguns problemas que este tipo de técnica de inspeção de artigos maliciosos possui.

Um *malware* pode fazer uso de engenharia social para que sua tarefa seja cumprida. Em um sistema automático de *sandboxing*, não é possível ter a interação do usuário com o *malware* propriamente dito. Logo, esta análise será prejudicada, já que o programa pode não ser executado em sua totalidade.

Códigos maliciosos podem estar preparados para ambientes virtualizados e de *sandboxing*. Um atacante, sabendo que uma análise tem um tempo máximo de execução, pode colocar em seu *malware* métodos que atrasam o início do código ou funções não legítimas. Assim, não acontece nada de suspeito na máquina monitorada e, consequentemente, gera um relatório contendo um falso-negativo.

Uma *sandbox* não pode dizer o que o *malware* faz. Ela pode relatar funcionalidades básicas, mas não pode dizer se o *malware* é um utilitário de *dump* de *hashes* de um gerenciador de contas de segurança ou um *backdoor keylogger* encriptado, por exemplo (SIKORSKI; HONIG, 2012).

2.2 Aprendizado de Máquina

Aprendizado de máquina extrai ideias de diversas disciplinas, incluindo inteligência artificial, probabilidade e estatística, teoria da informação, psicologia, entre outros (TOM M., 1997). O

aprendizado de máquina envolve o processo de coleta de dados e informações de um determinado cenário para otimização de futuros cenários, semelhantes ou não. É dito que um programa de computador aprende de uma experiência E, com respeito a algumas classes de tarefa T, e medição de performance P, se seu desempenho na função T, que é medida por P, melhorar com a experiência E (TOM M., 1997).

Por exemplo, um programa de computador que aprende a jogar damas pode melhorar seu desempenho, cuja medida é sua habilidade de ganhar na classe de tarefas envolvendo o ato de jogar damas, através de experiência obtida jogando damas (TOM M., 1997).

Há, neste contexto, quatro tipos de aprendizagem, que serão descritas nas próximas seções.

2.2.1 Aprendizagem Supervisionada

Nesta aprendizagem, o objetivo é inferir ou mapear os dados de treinamento que já estão classificados (MOHSSEN et al., 2017). Este é o cenário mais comum associado a classificação, regressão, e ranqueamento de problemas (MOHRI et al., 2012).

Métodos supervisionados são métodos que tentam descobrir a relação entre atributos de entrada (às vezes chamados de variáveis independentes) e um atributo alvo (também conhecido por variável dependente). A relação descoberta é representada em uma estrutura referida como Modelo. Normalmente, modelos descrevem e explicam o fenômeno, que está escondido no *dataset*, e podem ser usados para predição de valor do atributo alvo, sempre que os valores dos atributos de entradas são conhecidos (ROKACH; MAIMON, 2015). Por exemplo, na tabela 2.1, o resultado da última coluna, a variável dependente, é dado pela combinação dos valores das quatro primeiras colunas, as variáveis independentes.

Tabela 2.1: *Dataset* sobre a ocorrência de jogos de tênis (MOHSSEN et al., 2017).

Clima	Temperatura	Umidade	Ventoso	Jogo
Ensolarado	Quente	Alta	Falso	Não
Ensolarado	Quente	Alta	Verdadeiro	Não
Nublado	Quente	Alta	Falso	Sim
Chuvoso	Amena	Alta	Falso	Sim
Chuvoso	Agradável	Normal	Verdadeiro	Sim
Chuvoso	Agradável	Normal	Verdadeiro	Não
Nublado	Agradável	Normal	Verdadeiro	Sim
Ensolarado	Amena	Alta	Falso	Não
Ensolarado	Agradável	Normal	Falso	Sim
Chuvoso	Amena	Normal	Falso	Sim
Ensolarado	Amena	Normal	Verdadeiro	Sim
Nublado	Amena	Alta	Verdadeiro	Sim
Nublado	Quente	Normal	Falso	Sim
Chuvoso	Amena	Alta	Verdadeiro	Não

2.2.1.1 Classificação e Regressão

Em aprendizagem supervisionada, tem-se a divisão de algoritmos para classificação, nos quais a variável de saída é um rótulo (*label* em inglês), e também algoritmos para regressão, que tratam a saída como valores contínuos. Um exemplo para classificação pode ser visto na tabela 2.1 da seção 2.2.1.7, em que as variáveis de saída são "sim" e "não". A técnica de regressão é utilizada em previsão do tempo, onde os valores de saída são a temperatura no dado período.

2.2.1.2 Dilema Viés e Variância

Um resultado teórico importante de estatística e aprendizado de máquina é o erro de generalização do modelo, que pode ser expresso como a soma de três tipos de erro (AURÉLIEN, 2017): viés, variância e erro irredutível.

Tabela 2.2: Viés, Variância e Erro irredutível.

Viés	Este erro é comum quando se pressupõe erroneamente. Por exemplo, imaginar que as variáveis são independentes quando na verdade não são. Esse tipo de erro, quando alto, comumente sub-ajusta os dados de treinamento.
Variância	Quando o modelo possui muitos graus de liberdade, pode acabar tão bem treinado para o conjunto de dados de treinamento que se torna muito sensível a pequenas mudanças causando sobre-ajuste.
Erro irredutível	Este faz parte do próprio ruído do *dataset*, só é possível reduzi-lo limpando os dados que estão "poluindo" as amostras. Ou seja, arrumando a fonte de dados ou removendo amostras que estão completamente fora do padrão imaginado.

O grande dilema tratado entre viés e variância é que, ao aumentar a complexidade do modelo de predição, reduz o viés. Porém, aumenta a variância e, por outro lado, ao reduzir a complexidade, aumenta-se o viés e reduz a variância.

2.2.1.3 *Overfitting* e *Underfitting*

Overfitting refere-se à situação em que um algoritmo classifica muito bem o *dataset* de treinamento, mas não consegue prever corretamente dados fora do conjunto de treinamento. Em outras palavras, o algoritmo memorizou o conjunto de treinamento quando ele deveria ter aprendido (ROKACH; MAIMON, 2015). O contrário ,*underfitting*, tem-se o algoritmo que não conseguiu aprender a estrutura básica do *dataset* (AURÉLIEN, 2017).

Para se evitar ambos, deve-se observar, ao implementar um algoritmo de aprendizado de máquina, o viés, a variância e o erro irredutível como descrito na seção anterior.

2.2.1.4 Regressão Linear

Regressão linear é uma abordagem linear simples e direta para prever uma resposta quantitativa Y com base em uma única variável preditora X. Assume-se que existe aproximadamente uma relação linear entre X e Y (JAMES et al., 2013). Como pode-se notar na figura 2.1, tem-se uma aproximação por uma reta mostrando a relação linear entre o gasto com orçamento em propagandas exibidas na TV com a venda de um produto.

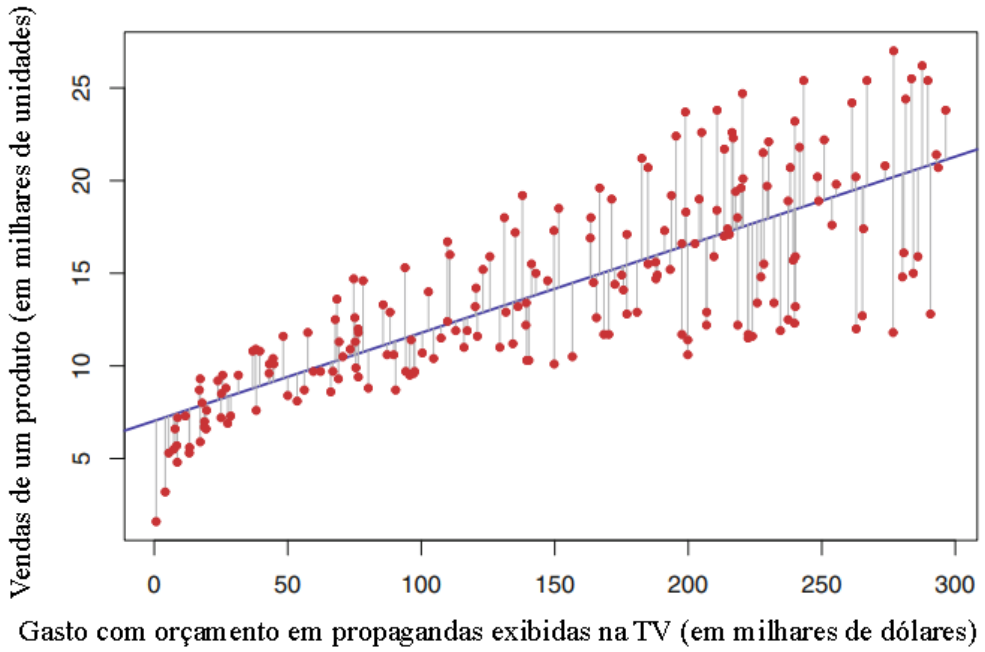


Figura 2.1: Regressão Linear aplicada em propagandas em TV. Adaptado de (JAMES et al., 2013).

2.2.1.5 Regressão Logística (LR)

Ao invés de modelar a resposta diretamente, a regressão logística modela a probabilidade de a resposta pertencer a uma categoria em particular (JAMES et al., 2013). A regressão logística é essencialmente idêntica a regressão linear. A única diferença é que há a necessidade de se ter um limiar para as saídas poderem produzir as predições de classes (CONWAY; JOHN MILES, 2012).

2.2.1.6 Análise do Discriminante Linear (LDA) ou Discriminante Linear de Fisher

Durante o treinamento, o algoritmo aprende o eixo mais discriminativo entre as classes, e estes podem ser usados para definir um hiperplano no qual os dados são projetados. O benefício é que a projeção irá manter as classes o mais distante possível. Em suma, a análise do discriminante linear é uma boa técnica para reduzir a dimensionabilidade antes de usar outro algoritmo de classificação, como o Máquina de Vetores de Suporte (SVM), tratado na seção 2.2.1.10 (AURÉLIEN, 2017). Na figura 2.2 há duas classes representadas junto com os histogramas resultantes da projeção na linha que une as médias das classes. Veja que não há sobreposição de classes no espaço projetado baseado

no discriminante linear de Fisher.

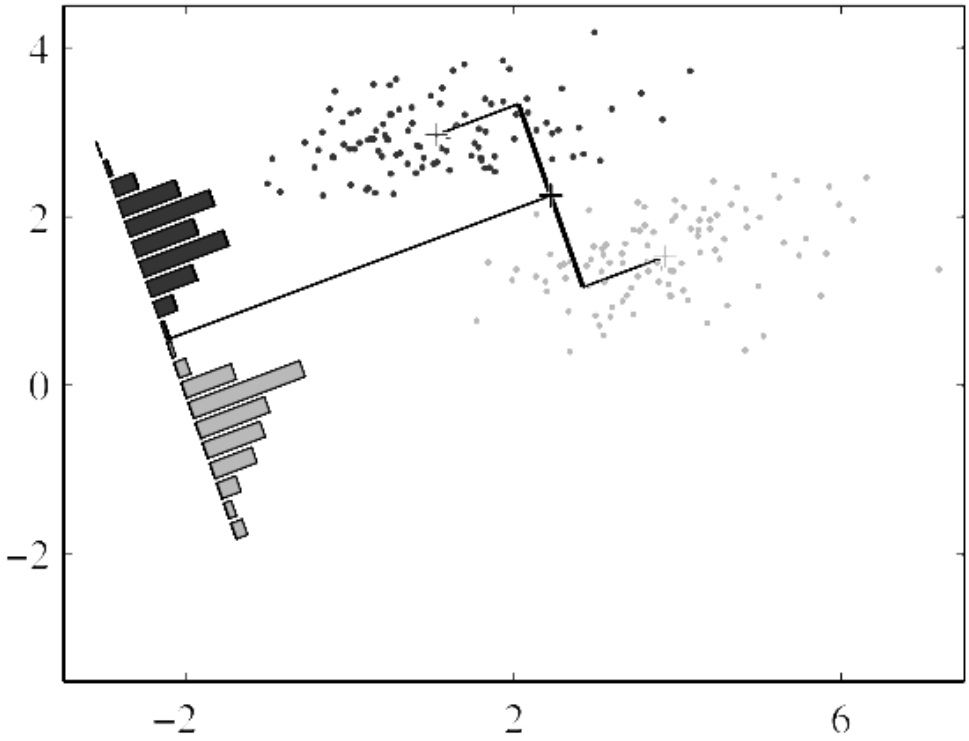


Figura 2.2: Análise do Discriminante Linear (BISHOP, 2011).

2.2.1.7 Naive Bayes (NB)

Este algoritmo é baseado no Teorema de Bayes para tomar suas decisões, necessitando de uma forte independência de suas variáveis, daí o nome *naive*, que significa ingênuo. A equação a seguir demonstra o teorema de Bayes em termos matemáticos (MOHSSEN et al., 2017):

$$P(A|B) = \frac{P(A)P(B|A)}{P(B)} \tag{2.1}$$

O *dataset* representado na tabela 2.1 será usado para melhor explicar o funcionamento desse algoritmo. A principio, já se percebe que a probabilidade do evento B ser "Sim" é de $\frac{9}{14}$. E de ser "Não" é de $\frac{5}{14}$. É necessário calcular a probabilidade condicional de cada uma das opções da tabela, sendo assim, pode-se considerar um evento X = (Clima = Nublado, Temperatura = Amena, Umidade = Normal, Ventoso = Falso).

Porém, é possível que ocorram algumas probabilidades condicionais inexistentes, que levariam a expressão para zero. Para evitar que isso ocorra, é utilizado o chamado *estimador de Laplace* (ROKACH; MAIMON, 2015), que retira a possibilidade de uma probabilidade ser zero.

2.2.1.8 *Perceptron* de Múltiplas Camadas (MLP)

A estrutura biológica de sistema neural e como ela executa suas funções inspiraram a ideia de redes neurais artificiais. Estas redes seguem o conceito de neurônio como uma célula individual que se comunica com outras células em uma rede. Esta célula recebe dados de outras células, processa as entradas e passa as saídas para outras células (MOHSSEN et al., 2017).

Um *perceptron* é uma rede neural que tem um único neurônio que recebe múltiplas entradas, mas produz apenas uma saída. Este *perceptron* pode solucionar problemas de classificação para classes linearmente separáveis. No entanto, em casos não lineares, um *perceptron* de uma camada irá falhar na classificação (MOHSSEN et al., 2017).

Em uma rede neural de MLP, cada *perceptron* recebe um conjunto de entradas de outros *perceptrons* e, então, dispara ou não se a soma ponderada das entradas passou de um certo limiar. Assim como em uma rede de um *perceptron*, o viés (que determina o limiar), além dos pesos, são ajustados durante a fase de treinamento (MOHSSEN et al., 2017). Na rede MLP, tem-se a camada de entrada, uma ou mais camadas ocultas (*hidden*) e uma camada final chamada de camada de saída (AURÉLIEN, 2017).

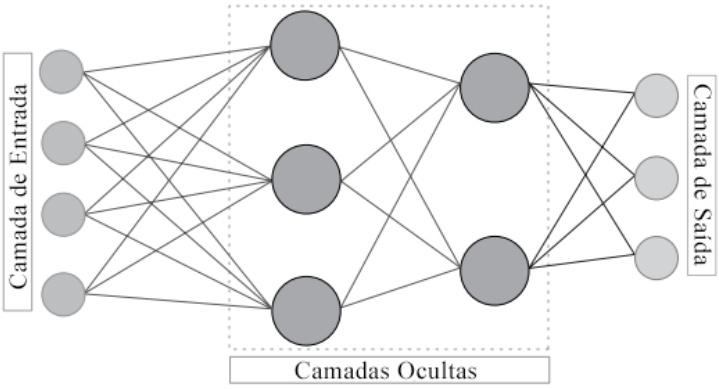


Figura 2.3: Uma Rede Neural MLP. Adaptado de (MOHSSEN et al., 2017).

Para aprender uma rede neural, são gerados pesos e vieses aleatórios. Então, uma instância de treinamento é passada para a rede neural, onde a saída de cada camada é enviada para a próxima camada até o processamento (de acordo com os pesos iniciais) da saída prevista na camada de saída. E o erro na última camada é a diferença entre as saídas reais e previstas. A partir deste ponto, os pesos entre a camada de saída e as camadas ocultas são corrigidos e, em seguida, os pesos entre a camada oculta e a camada de entrada são ajustados de forma inversa. Por fim, outra instância de treinamento é endereçada para a rede neural e para o processo de avaliação do erro na camada de saída, corrigindo, deste modo, os pesos entre as camadas diferentes a partir da camada de saída até a camada de entrada (MOHSSEN et al., 2017).

2.2.1.9 K Vizinhos Mais Próximos (KNN)

O KNN, um dos algoritmos mais simples de aprendizado de máquinas, é um tipo de aprendizado baseado em instância, ou aprendizado preguiçoso, onde a função é apenas aproximada localmente e toda computação é diferida até a classificação (MOHSSEN et al., 2017).

Em reconhecimento de padrões, o algoritmo k vizinhos mais próximos é um método não paramétrico usado para classificação e regressão. Em ambos os casos, a entrada consiste dos k próximos exemplos de treinamento no espaço de características (MOHSSEN et al., 2017). O k é escolhido para ser um número impar em um problema de duas classes e, em geral, não ser múltiplo do número de classes (THEODORIDIS; KOUTROUMBAS, 2003). O resultado depende se KNN foi usado para classificação ou regressão.

- Na classificação, um objeto é classificado pela maioria dos seus vizinhos e classe do objeto é dada pela classe mais comum entre os k vizinhos próximos. Isto é, a classe será igual à da maioria em sua volta. Em caso de k = 1, o objeto tem classe igual ao seu vizinho mais próximo. Nota-se que, na figura 2.4, a classe prevista para a amostra representada por uma estrela será "quadrado" quando k = 3, já que, dos seus três próximos vizinhos, a maioria deles são da classe "quadrado".
- Na regressão, a saída é dada pela média dos valores de seus k próximos vizinhos e não mais o valor mais comum.

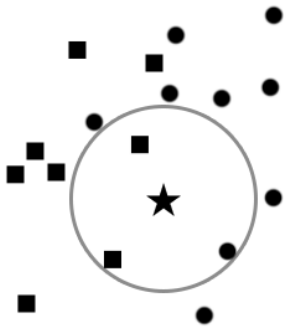


Figura 2.4: Exemplo de KNN para k = 3.

Para ambos, pode ser útil dar peso às contribuições dos vizinhos, deste modo os vizinhos mais próximos tem um impacto maior na classificação quando comparados com os mais distantes. Uma métrica comum é usar o inverso da distância entre o objeto e o vizinho (MOHSSEN et al., 2017).

2.2.1.10 Máquina de Vetores de Suporte (SVM)

Neste algoritmo é criado um hiperplano que tem como objetivo separar as classes, uma em cada lado de seu vetor. Para o melhor resultado possível, deseja-se que a margem (distância entre o hiperplano e o ponto mais próximo de cada classe) seja a maior possível, veja na figura 2.5 dois

exemplos de hiperplanos, onde a direção 1 tem uma margem menor que a direção 2. Além disto, tal margem deve ser igual para os dois lados para permitir que novos dados sejam classificados com a menor taxa de erros possível. Veja na figura 2.5 que ambas as direções têm margens iguais para os dois lados e que a direção 2 tem uma margem mais ideal para o problema do que a direção 1 (THEODORIDIS; KOUTROUMBAS, 2003).

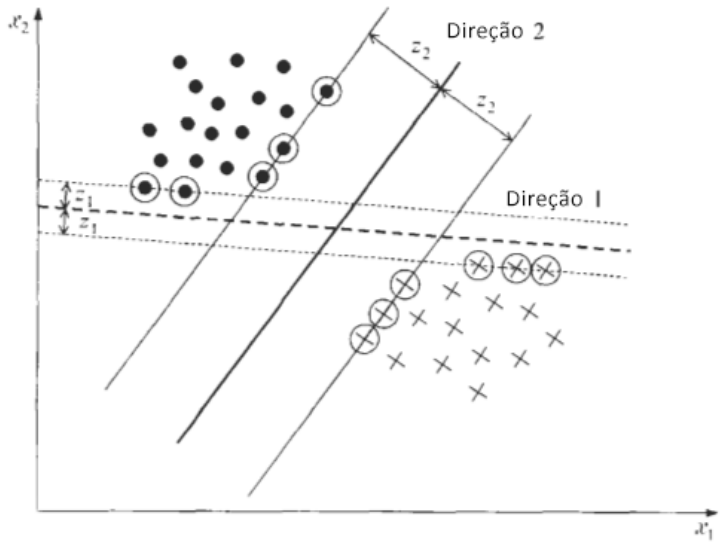


Figura 2.5: SVM com classes separáveis. Adaptado de (THEODORIDIS; KOUTROUMBAS, 2003).

No entanto, o caso anterior só funciona corretamente se as classes forem separáveis. Isto quer dizer que não há um hiperplano que separe os pontos de cada classe em grupos homogêneos de cada lado, ou seja, haverá um hiperplano que minimize a heterogeneidade dos dois grupos. O objetivo agora é fazer a margem o mais larga possível, mas que ao mesmo tempo mantenha o número de pontos "do lado errado" o mais próximo de zero que se consiga (THEODORIDIS; KOUTROUMBAS, 2003). Na figura 2.6 pode-se observar que pontos de ambas as classes estão dos dois lados do hiperplano, mas a quantidade de pontos no lado oposto é o menor possível.

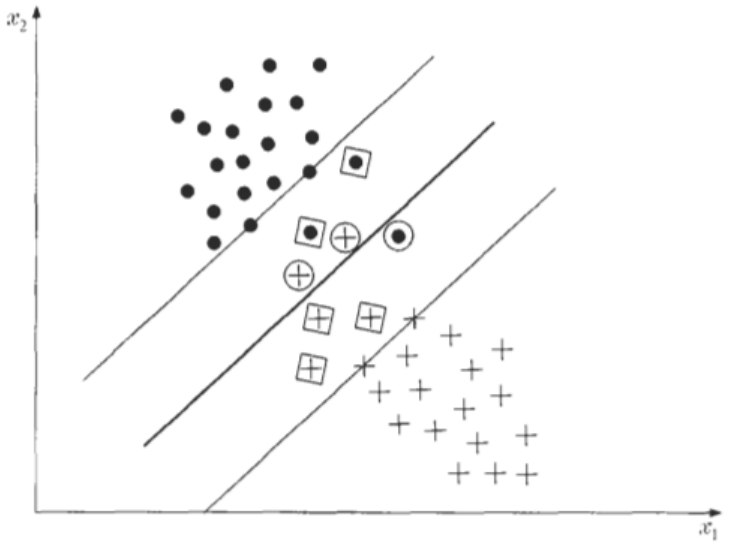


Figura 2.6: SVM com classes não separáveis (THEODORIDIS; KOUTROUMBAS, 2003).

2.2.1.11 Árvores de Decisão

Como a máquina de vetores de suporte, árvores de decisão são algoritmos versáteis de aprendizado de máquinas que podem fazer tarefas de classificação e regressão, e até tarefas de múltiplas saídas (AURÉLIEN, 2017). Este modelo classifica dados de um *dataset* ao fazer uma série de consultas a partir da raiz (*root* em inglês) até chegar às folhas. Cada uma destas folhas representa uma classe e a raiz representa o atributo de função principal na classificação. Um modelo de árvore de decisão segue os seguintes passos, segundo (MOHSSEN et al., 2017):

- Todas as amostras de treinamento são situadas na raiz da árvore;
- Estas são separadas a partir de atributos selecionados baseados em medidas estatísticas, que serão discutidas a seguir;
- O particionamento recursivo continua até não existir amostras de treinamento, ou até não ter atributos, ou até a quantidade restante das amostras de treinamento pertencerem à mesma classe.

Critérios de Divisão de Árvore

Na maior parte dos indutores de árvores de decisão, as funções de divisão discreta são univariáveis, isto é, um nó interno na árvore é dividido de acordo com um único atributo. Consequentemente, um indutor procura pelo melhor atributo para melhor performance desta divisão (ROKACH; MAIMON, 2015). O resultado será mais preciso se este for dividido de maneira que os outros *datasets* resultantes tenham menos incertezas ou entropia menor. Isto é, para a classificação em árvores de decisão, a seleção de atributos é um passo vital (MOHSSEN et al., 2017).

Há vários critérios de uma única variável que podem ser caracterizados de maneiras diferentes, como a origem de sua medição e a estrutura de sua medição (i.e. critério baseado em impureza e critério binário) (ROKACH; MAIMON, 2015).

Árvores de Classificação e Regressão (CART)

O CART (do inglês, *Classification and Regression Trees*) separa o conjunto de treinamento em dois subconjuntos usando um critério de impureza e um único atributo das amostras, veja na figura 2.7 que a primeira divisão gera uma folha com 0 de impureza de Gini e um nó com 0,5, menor que o inicial de 0,6667, nota-se que para isto o CART usou a característica de comprimento da pétala da flor íris. O critério de impureza do CART é variável de acordo com sua configuração. Assim que sua divisão em dois do conjunto de treinamento for bem sucedida, outra divisão ocorre usando a mesma lógica e assim por diante, recursivamente. Estas divisões param apenas quando a sua profundidade máxima é atingida ou quando o algoritmo não consegue mais achar uma divisão que diminua a impureza. Existe, no entanto, outros critérios para a finalização da execução do algoritmo. Por padrão, a medida impureza de Gini é usada, mas pode ser usada a impureza de entropia (de Shannon), em vez do primeiro (AURÉLIEN, 2017).

Como pode ser visto, o CART é um algoritmo "guloso" (ou míope): procura de maneira gananciosa por uma divisão ótima no alto nível, e então repete o processo a cada nível. Ele não checa se a divisão irá levar na menor impureza possível nos níveis abaixo ou não (AURÉLIEN, 2017). Produzindo assim uma solução decentemente boa, já que a solução ótima cai no problema NP-Completo (COURNAPEAU, 2017).

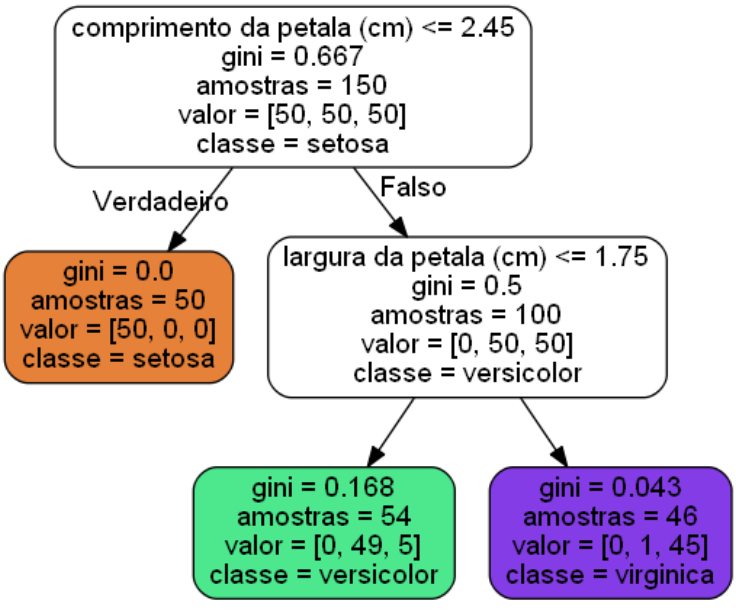


Figura 2.7: Árvore de Decisão gerada a partir do algoritmo CART. Adaptado de (AURÉLIEN, 2017).

As árvores de decisão geradas pelo CART são binárias, isto é, todo nó sempre terá apenas duas subdivisões, sendo estas folhas ou outros nós. Veja na figura 2.7 um pequeno exemplo de uma árvore de decisão para flores íris gerada pelo CART usando a biblioteca do python *Scikit-Learn*, tratada na seção 3.3.1. Note que os valores de impureza de Gini são sempre menores a cada nível que a árvore desce e que todo nó (em branco) só tem duas saídas para o próximo nível.

2.2.2 Outras Formas de Aprendizagem

Além da aprendizagem supervisionada, existem também outros tipos de aprendizagem que não são utilizadas neste trabalho, dentre elas:

- Aprendizagem Não Supervisionada: Contrária da forma de aprendizagem apresentada na seção 2.2.1, a aprendizagem não supervisionada contém apenas os vetores dos atributos e suas classificações são ausentes;
- Aprendizagem Semi-supervisionada: Esta é a junção tanto da aprendizagem supervisionada quanto da não supervisionada, normalmente a quantidade de dados classificados é bem menor que a de não classificados (AURÉLIEN, 2017);
- Aprendizagem Reforçada: O agente, que é o sistema que está aprendendo, pode observar o ambiente, selecionar e executar ações e assim receber recompensas ou penalidades (AURÉLIEN, 2017). Com estas informações, o agente cria políticas para ditar suas próximas ações, melhorando assim o seu resultado.

2.2.3 Métricas de Desempenho de um Classificador

Medir o desempenho de um algoritmo de aprendizagem de máquinas é fundamental para compreender sua qualidade e, consequentemente, realizar mudanças que possam gerar resultados melhores.

2.2.4 *White Box* e *Black Box*

Como pode ser observado, árvores de decisão são intuitivas em suas decisões e de fácil interpretação. Modelos são comumente chamados de modelos *white box* (AURÉLIEN, 2017) quando é possível saber porque uma certa amostra foi classificada corretamente ou não, isto é, ver as decisões tomadas dentro do algoritmo. No entanto, modelos *black box* são, usualmente, difíceis de explicar em termos simples o por quê uma predição foi correta ou não (AURÉLIEN, 2017).

2.2.4.1 Acurácia

A acurácia de um classificador é medida a partir da razão entre a quantidade de observações corretas sobre a quantidade total de observações, incluindo predições erradas. Para tal medição em aprendizado de máquinas, emprega-se, comumente, a reamostragem (ou *resampling*) dos

dados para se fazer uma média de acurácias. Isto é, os dados são separados em conjuntos de treinamento e de teste de maneiras diferentes várias vezes. Subamostragem aleatória e validação-cruzada k-dobras (*k-fold cross-validation* em inglês) são métodos comuns de reamostragem. Na subamostragem aleatória, os dados são aleatoriamente particionados várias vezes em conjuntos disjuntos de treinamento e de teste, e então faz-se a média das acurácias de cada partição. Na validação-cruzada k-dobras, os dados são também divididos aleatoriamente, mas agora em k subconjuntos mutualmente exclusivos de aproximadamente mesmo tamanho. Então há o treinamento k vezes, cada vez é testado uma das k dobras e treinadas as outras k - 1 dobras (ROKACH; MAIMON, 2015).

Outro método de medição de qualidade de um algoritmo é o de *holdout*, o qual diz que um dado *dataset* deve ser dividido aleatoriamente em dois grupos, o de treinamento e o de teste. Normalmente, se aloca de 66% a 80% do *dataset* para o grupo de treinamento e o restante para o grupo de teste. Com a divisão feita, pega-se o grupo com mais amostras e o usa para o treinamento do algoritmo, em seguida, usa-se o grupo menor para medir a acurácia do classificador. Como apenas uma fração do *dataset* foi utilizada, tende-se a ter uma valor de acurácia abaixo do esperado em relação à validação-cruzada k-dobras (ROKACH; MAIMON, 2015).

Todavia, a acurácia nem sempre é uma boa medida para a análise de um modelo com classes não balanceadas (ROKACH; MAIMON, 2015), tornando-se importantes outras medidas para ajudar nessas medições, tais como precisão, *recall* e *F-measure*.

2.2.4.2 Matriz de Confusão

A matriz de confusão é usada como um indicador das propriedades de uma regra (discriminante) de classificação. Esta matriz contém o número de elementos que foram corretamente ou incorretamente classificados por cada classe. Sua diagonal principal é o número de observações que foram corretamente classificadas para cada classe, e a diagonal secundária indica o número de observações erroneamente classificadas (ROKACH; MAIMON, 2015).

Tabela 2.3: Outras Métricas.

	Predição Positiva	Predição Negativa
Amostras Positivas	Verdadeiro Positivo	Falso Negativo
Amostras Negativas	Falso Positivo	Verdadeiro Negativo

A partir dos valores da tabela 2.3 é possível calcular outras métricas, incluindo a acurácia (ROKACH; MAIMON, 2015):

- A taxa de classificações erradas (TCE) é dada pela equação 2.2:

$$TCE = \frac{FalsoPositivo + FalsoNegativo}{AmostrasPositivas + AmostrasNegativas} \tag{2.2}$$

- A taxa de falsos negativos (TFN) é dada pela equação 2.3:

$$TFN = \frac{FalsoNegativo}{AmostrasPositivas} \tag{2.3}$$

- Taxa de falsos positivos é dada pela equação 2.4:

$$TFP = \frac{FalsoPositivo}{VerdadeiroNegativo + FalsoPositivo} \tag{2.4}$$

- A precisão mede quantas amostras classificadas como classe "positiva" são realmente desta classe (ROKACH; MAIMON, 2015). Esta medida é tipicamente usada com o *recall* (AURÉLIEN, 2017). E pode ser vista na equação 2.5:

$$Precisão = \frac{VerdadeiroPositivo}{PrediçõesPositivas} \tag{2.5}$$

- O *recall* (ou taxa de verdadeiros positivos) avalia o quão bem o classificador pode reconhecer amostras positivas e as defini-las (ROKACH; MAIMON, 2015). Seu valor é dado pela equação 2.6:

$$Recall = \frac{VerdadeiroPositivo}{AmostrasPositivas} \tag{2.6}$$

- A especificidade (*Specificity* em inglês), mostrada na equação 2.7, mede quão bem o classificador pode reconhecer amostras negativas (ROKACH; MAIMON, 2015), ou seja, a taxa de verdadeiros negativos:

$$Especificidade = \frac{VerdadeiroNegativo}{AmostrasNegativas} \tag{2.7}$$

- A acurácia citada na seção 2.2.4.1 pode ser calculada a partir de algumas destas medidas anteriormente citadas, gerando a equação 2.8:

$$\begin{aligned} Acurácia = & Recall \times \frac{AmostrasPositivas}{AmostrasPositivas + AmostrasNegativas} \\ & + Especificidade \times \frac{AmostrasNegativas}{AmostrasPositivas + AmostrasNegativas} \end{aligned} \tag{2.8}$$

2.2.4.3 *Tradeoff* Entre Precisão e *Recall*

Normalmente, tem-se um *tradeoff* entre as medidas de precisão e de *recall*. Tentar melhorar uma medida muitas vezes resulta na deterioração da outra medida (ROKACH; MAIMON, 2015).

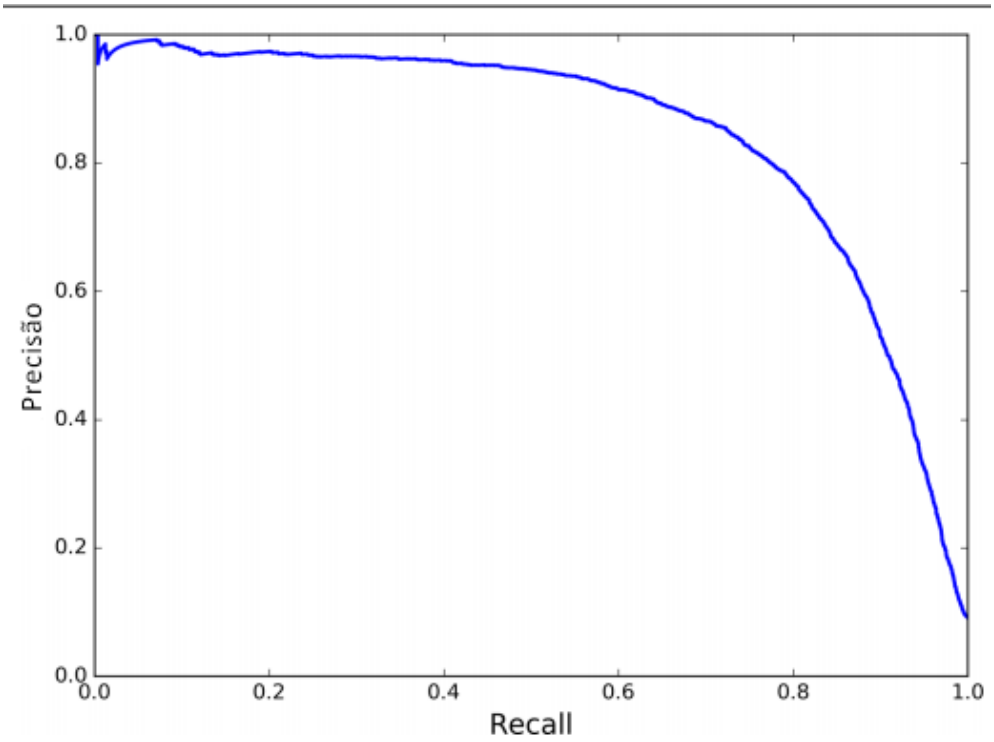


Figura 2.8: Diagrama precisão-*recall*. Adaptado de (AURÉLIEN, 2017).

Dado um classificador probabilístico, este gráfico de *tradeoff* da figura 2.8 pode ser obtido a partir de diferentes valores de limiares de decisão. Por exemplo, em um problema de classificação binária, o classificador prefere a classe "não passar" sobre a classe "passar" se a probabilidade para "não passar" for maior que 0,5. De outra forma, ao usar outro limiar que não 0,5 (ROKACH; MAIMON, 2015), tem-se outro valor para este gráfico.

Este problema é descrito como *multi-criteria decision-making* (MCDM). O método mais simples e comum usado para solucionar o MCDM é o modelo de soma ponderada. Esta técnica combina os critérios em um único valor ao aplicar a ponderação apropriada, já que a média aritmética pode resultar em valores pouco conclusivos (ROKACH; MAIMON, 2015). Matematicamente, o valor desta medida conhecida por *F-Measure* (ou *F-score*) é de:

$$\begin{aligned} F_1 &= \frac{2}{\frac{1}{precisão} + \frac{1}{recall}} \\ &= 2 \times \frac{precisão \times recall}{precisão + recall} \end{aligned} \tag{2.9}$$

O *F-measure* tem valores entre 0 e 1. Ele terá seu maior valor quando precisão e *recall* forem iguais. Percebe-se que em cada ponto da curva viés precisão-*recall* há valores diferentes para o par, resultando em diferentes valores para o *F-measure*. Significa que classificadores diferentes têm valores diferentes do gráfico da figura 2.8 (ROKACH; MAIMON, 2015).

Pode-se ver que, para averiguar a qualidade de um algoritmo, é necessário o uso de diversas métricas de desempenho diferentes e não só da acurácia. Além disso, existem outros critérios de medição de desempenho da predição de um certo algoritmo de aprendizado de máquina,

como, por exemplo, a complexidade computacional ou a compreensibilidade de um certo classificador(ROKACH; MAIMON, 2015).

Capítulo 3

Métodos Propostos e Ferramentas Utilizadas

Diversas ferramentas são utilizadas para a execução deste projeto. A principal delas é o *Cuckoo*, usada para fazer análises de *malwares*, enquanto outras são empregadas para dar suporte à ela, como o *Virtualenv*, o *VirtualBox* e o *VirusTotal*. Além das ferramentas usadas para o sistema *web* e criação da etapa de aprendizado de máquina.

3.1 Descrição dos Métodos

No primeiro momento, um banco de *malwares* e arquivos benignos conhecidos são testados na *sandbox* que, então, gerará relatórios padronizados para a criação de um *dataset*. Ela poderá concluir se o artefato analisado é malicioso ou não. Em uma segunda instância, o processo é repetido, mas agora com um analisador automático, utilizando aprendizado de máquina, o qual será treinado a partir do *dataset*, anteriormente gerado, para que assim seja capaz de interpretar a procedência de um determinado arquivo.

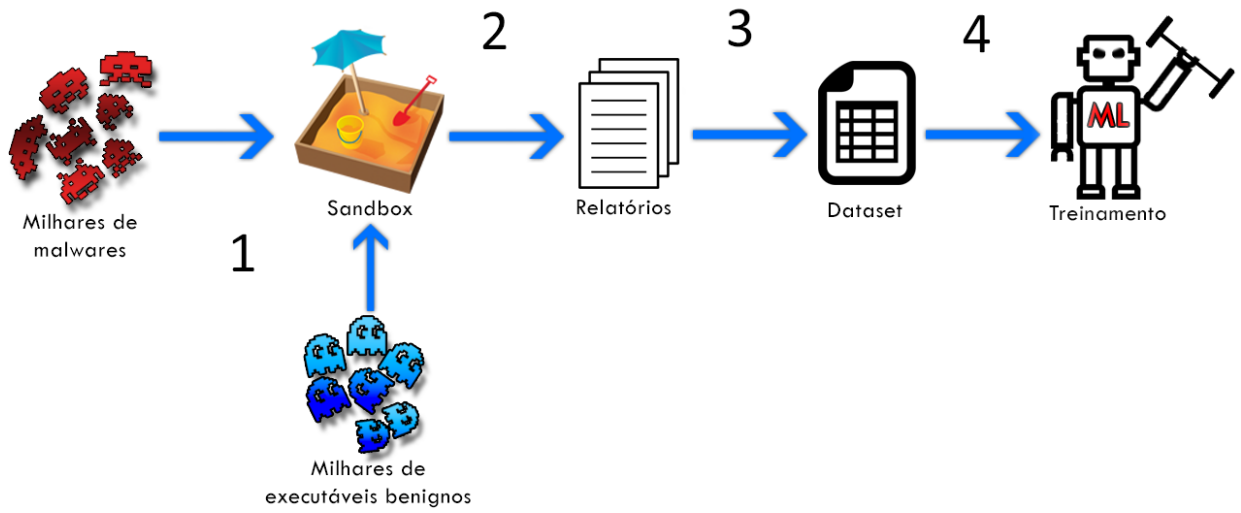


Figura 3.1: Primeira Etapa do Projeto.

Cada caminho apresentado na figura 3.1 demonstra uma etapa relevante no desenvolvimento da primeira parte deste projeto, portanto, será descrita detalhadamente:

1. Milhares de artefatos, previamente escolhidos e cujo comportamento é conhecidamente malicioso, são lançados em diversas *sandboxes* para serem analisados de maneira que a máquina de aprendizagem possa ser treinada com o volume de relatórios que serão gerados;

2. A *sandbox*, uma vez que tem a posse do artefato, gera relatórios bem detalhados, os quais são otimizados e adequados com informações relevantes e não redundantes (pré-processamento), que serão tratados na seção 4.1.1.1;

3. Com base nos dados relevantes, após o pré-processamento, um *dataset* é gerado contendo os dados de todos os artefatos submetidos nesta etapa de treinamento;

4. Por fim, o *dataset* será usado para treinamento dos algoritmos de aprendizado de máquina.

Com o intuito de mostrar uma aplicação prática dessa solução, foi proposto um sistema *web* capaz de submeter arquivos para essa *sandbox*, para ser usado como facilitador da análise de artefatos por um analista. Dessa maneira, o analista poderá ter acesso tanto aos relatórios gerados como à previsão da procedência dos artefatos determinada pelos algoritmos de aprendizado de máquina.

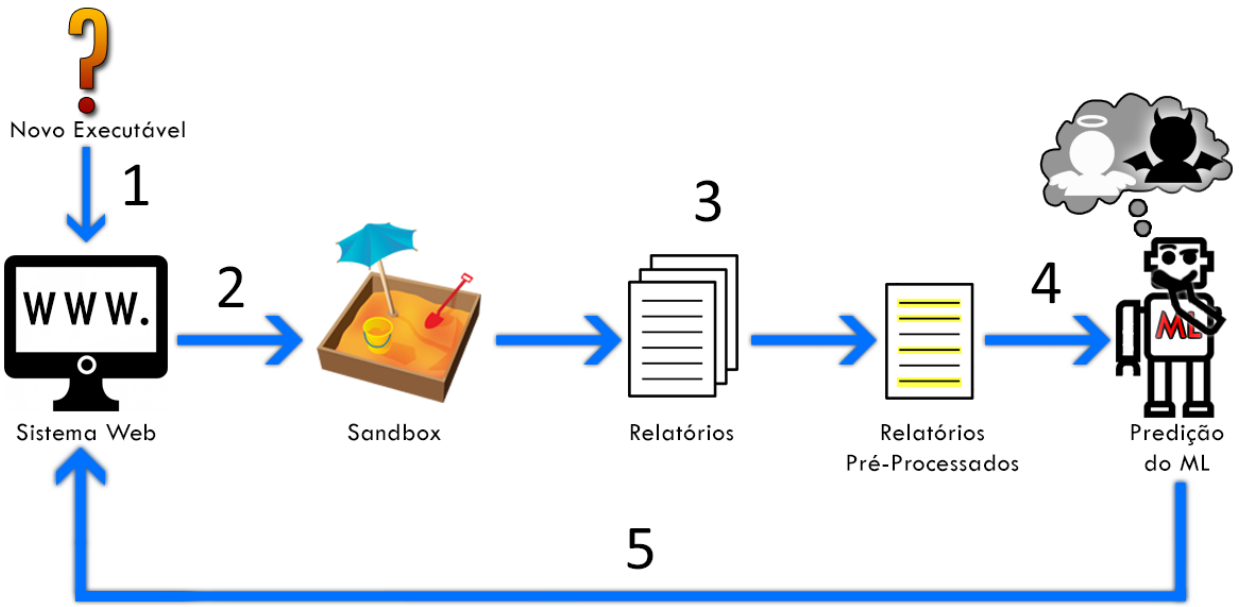


Figura 3.2: Segunda Etapa do Projeto.

1. A princípio, será submetido um novo arquivo ao sistema *web*;
2. O artefato então será submetido à *sandbox* para ser analisado e gerados os diversos relatórios;
3. Ocorrerá em seguida o pré-processamento dos relatórios produzidos, para que assim sejam enviados ao aprendizado de máquina;
4. Nesta etapa, os relatórios recebidos serão analisados por diversos algoritmos de ML;
5. Por fim, as predições resultantes e os relatórios anteriormente produzidos serão enviados para o sistema *web*, completando assim o ciclo.

3.2 *Cuckoo Sandbox*

Uma *sandbox* é um mecanismo de segurança para executar programas não confiáveis em um ambiente seguro sem o risco de danificar sistemas "reais". *Sandboxes* envolvem ambientes virtualizados que comumente simulam serviços de tal forma que permitam ao programa testado, legítimo ou não, funcione normalmente (SIKORSKI; HONIG, 2012).

Cuckoo é um sistema de código aberto de análise de *malwares* automatizada. É usado para executar automaticamente e analisar arquivos, para assim coletar informações das análises que destacam o que um arquivo malicioso faz, ou tenta fazer, enquanto está em execução em um sistema operacional (SO) isolado (CUCKOO-FOUNDATION, 2017). Dentre os resultados que o *Cuckoo* gera, tem-se:

- Traços das chamadas de todos os processos feitos pelo arquivo;
- Arquivos criados, deletados e baixados durante a execução da máquina virtual;

- *Dumps* de memória dos processos do artefatom além do *dump* completo de memória da máquina virtual. Uma análise mais completa desta característica pode ser encontrada na seção 5.3.2;
- Tráfico de rede no formato .PCAP;
- Capturas de tela obtidas durante a execução do arquivo.

3.2.1 Arquitetura

O *Cuckoo* foi projetado para ser usado como uma aplicação *standalone*, assim como para ser integrado em grandes *frameworks*, graças ao seu design extremamente modular. Esta arquitetura permite que um usuário possa customizá-lo conforme queira (CUCKOO-FOUNDATION, 2017).

Esta ferramente permite a análise de diversos tipos de arquivos, alguns deles são: executáveis (.EXE), Dlls do Windows, documentos em PDF, URLs, *scripts* em Visual Basic, PHP, Python, e "quase todo o resto" (CUCKOO-FOUNDATION, 2017).

Cuckoo Sandbox consiste em um programa de gerenciamento central que controla a execução das amostras e análises. Cada análise é feita em uma máquina isolada, virtual ou física. Uma máquina hospedeira executa o *Cuckoo* (chamada de *Cuckoo Host*) e gerencia os dispositivos a serem infectados. Isto quer dizer que o *Cuckoo Host* será responsável por grande parte do processo de análise, enquanto as máquinas *Guests* (visitantes em português) executam os artefatos em um ambiente isolado para evitar que este se propague. Assim, permite a analisado de maneira segura (CUCKOO-FOUNDATION, 2017). A figura 3.3 exemplifica a arquitetura descrita.

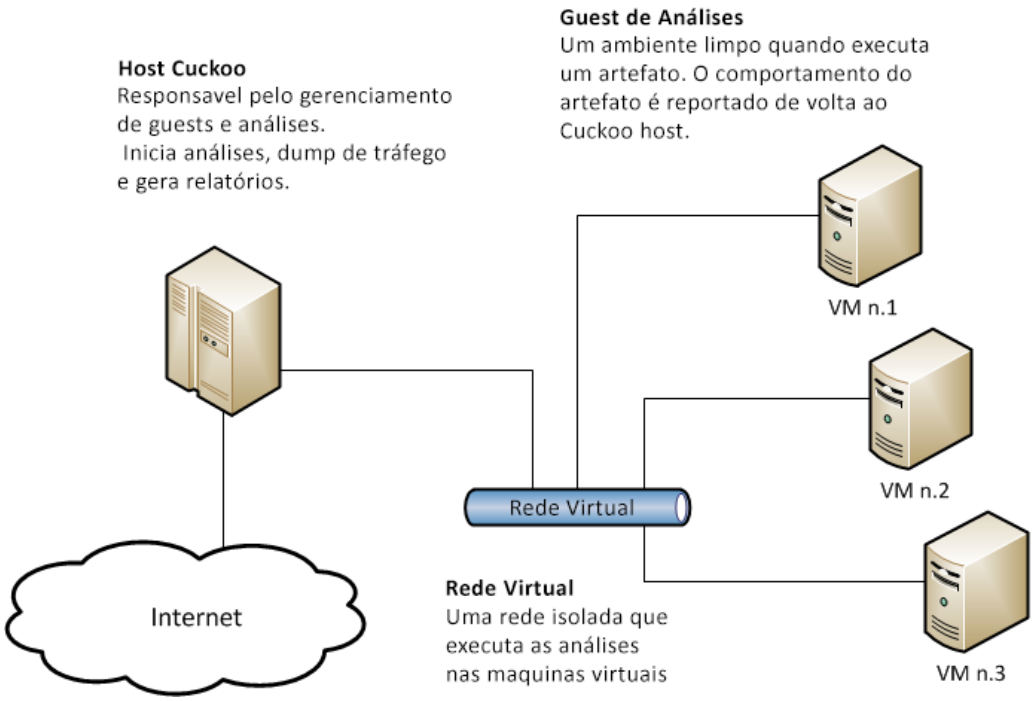


Figura 3.3: Arquitetura do *Cuckoo* (CUCKOO-FOUNDATION, 2017).

3.2.2 Funcionamento

O funcionamento da ferramenta também pode ser dividido em duas grandes etapas, demonstradas na figura 3.4 que, de certa forma, resumem o seu funcionamento, *Processing* e *Reporting*.

- *Processing*: Nesta etapa, são feitos os processamentos no artefato submetido, entre eles, análise estática, análise dinâmica, análise de rede, o resultado de cada um desses módulos é unificado em um grande relatório (dicionário do *python*) e passado para os módulos de *reporting*;
- *Reporting*: Nesta etapa, são gerados os relatórios, tais como, página HTML com alguns resultados gerados ou até mesmo um objeto JSON com todos os resultados do processamento. Também podem ser persistidos em um banco de dados, como, por exemplo, o *MongoDB*, lembrando que a aplicação é modular, podendo ser adaptadas muitas outras formas de relatório.

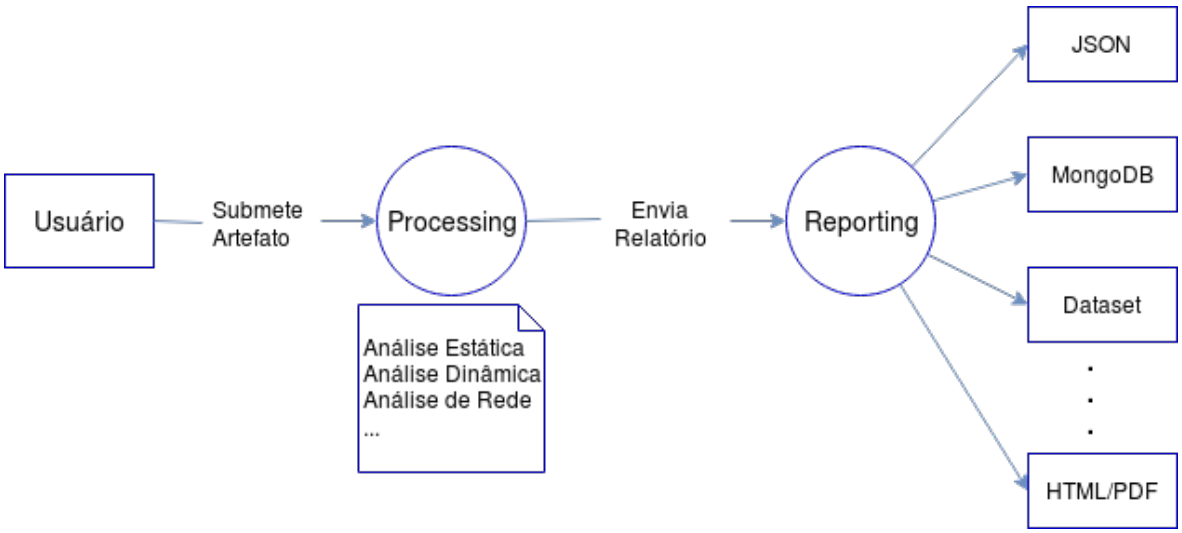


Figura 3.4: Lógica Principal do *Cuckoo Sandbox*.

3.2.3 *Cuckoo* API

Cuckoo Sandbox tem também uma API implementada em *Flask* (que será abordada na seção 3.3.4) com uma malha de recursos prontos para serem utilizados. Dentre suas utilidades, tem-se (CUCKOO-FOUNDATION, 2017):

- Enviar novos artefatos para a fila para serem analisados;
- Remover artefatos da fila;
- Mudar a posição na fila de certo artefato;
- Enviar novos artefatos para a fila para serem analisados;

- Adquirir um ou todos os *screenshots* de uma análise já feita;
- Requisitar o *report* a partir de um ID;
- Ver informações do *Cuckoo Sandbox*;
- Listar as máquinas *guest* e suas respectivas informações.

3.2.4 *Cuckoo Working Directory* (CWD)

O *Cuckoo* possui um usuário próprio para administrar seus recursos, evitando assim executar o programa com o usuário *root*, o que traria algumas vulnerabilidades. Além de possuir um usuário próprio, o *Cuckoo* possui também um diretório de trabalho, onde há os arquivos gerados e as configurações feitas. Mais sobre o funcionamento é encontrado em (CUCKOO-FOUNDATION, 2017). Os principais arquivos que estão localizados neste diretório são:

- Arquivos de configuração;
- Armazenamento dos resultados das análises;
- O *agent* do *Cuckoo* que roda nas Máquinas Virtuais.

3.2.5 *Cuckoo Agent*

O *agent* do *Cuckoo* é executado dentro da *guest*. Ele gerencia a comunicação e troca de dados com o *host* através de uma pequena API. Este *agent* foi desenvolvido para ser *cross-platform*, significando que ele é capaz de funcionar em diversos sistemas operacionais como Windows, Android, Linux e Mac OS X. Para fazer esta ferramenta funcionar corretamente, há a necessidade da instalação do *Python* nas máquinas *guests*, pois este é um *script* desenvolvido nesta linguagem de programação (CUCKOO-FOUNDATION, 2017).

3.3 *Python*

Python é uma linguagem de programação poderosa e fácil de aprender. Tem estruturas de dados de alto nível eficientes e uma simples, mas eficiente, abordagem à programação orientada a objeto. Esta linguagem permite a divisão de programas em módulos que podem ser reusados em outros programas em *Python* (PYTHON-SOFTWARE-FOUNDATION, 2017), como é o caso do *Flask*, usado na API do *Cuckoo*.

Python é uma das linguagens de programação mais populares da atualidade, sendo ranqueada em 2017 pela IEEE Spectrum como a mais popular entre as linguagens utilizadas em projetos *open source* (DIAKOPOULOS; CASS, 2016). Inclusive, *softwares* essenciais para este projeto são totalmente escritos em *Python* na versão 2.7.

3.3.1 *Scikit-Learn*

Scikit-learn, ou *sklearn*, possui tanto os algoritmos para gerar modelos de aprendizagem quanto para a análise deles. É extremamente simples de ser usado e o fato de possuir diversos algoritmos de ML inseridos é um ótimo ponto de partida para o aprendizado de máquinas (AURÉLIEN, 2017). Para a análise do desempenho dos diferentes modelos, esta ferramenta possui algumas funções facilitadoras, que auxilia a ter não só a visualização mas também o valor estatístico das medidas.

3.3.2 *croos_ val_ score* e *cross_ val_ predict*

Assim como descrito na seção 2.2.4.1, o *sklearn* implementa duas funções baseando-se na validação cruzada e nas k-dobras, *cross_ val_ score* e *cross_ val_ predict*. A primeira é utilizada quando se possui uma menor quantidade de amostras, tornando pior a divisão do *dataset* em treinamento e validação, pois implicaria em resultados piores devido à baixa quantidade de amostras. Este método pode ser utilizado para encontrar acurácia, entropia, *recall* ou *F-measure*. De maneira semelhante, a *cross_ val_ predict* também se baseia na validação cruzada, porém no lugar de retornar uma pontuação, retorna um conjunto de predições (AURÉLIEN, 2017), podendo depois ser utilizado para determinar a matriz de confusão do modelo analisado.

3.3.3 *Pandas*

Pandas é uma biblioteca *Python* feita para prover estrutura de dados rápidas, flexíveis e expressivas, trabalhando com essas estruturas de maneira intuitiva (MCKINNEY, 2017). Existe um grande ganho ao se trabalhar com estruturas performáticas, principalmente ganho de processamento, podendo assim resolver problemas em um menor espaço de tempo. Por obter performances melhores que muitas outras estruturas utilizadas em *Python*, o *Pandas* tem sido frequentemente utilizado para análises financeiras.

As estruturas formadas são normalmente indexadas, o que facilita na velocidade de busca dos dados, usualmente são utilizadas estruturas como dicionário do *Python* e transformadas nessas estruturas tabulares do pandas, trazendo um ganho de desempenho em todo o código. O fato de trabalhar com entradas como CSV e SQL faz com que essa ferramenta possa ser usada para diversas aplicações. Portanto, soluções mais performáticas, como o *scikit-learn*, que necessita de velocidade para excutar algoritmos, são utilizadas nessas estruturas.

3.3.4 *Flask*

Flask é um *microframework* desenvolvido em *Python*, utilizado para desenvolvimento de APIs e sistemas WEB. Considerado por muitos como o *framework* mais "pythonico", ou seja, o que melhor utiliza das funcionalidades do *Python*, linguagem que preza por simplicidade e facilidade. Para o seu funcionamento, é utilizada a ferramenta *Jinja2* como *template* para as paginas HTML e a *Werkzeug*, uma espécie de kit de ferramentas *Web Server Gateway Interface* (WSGI).

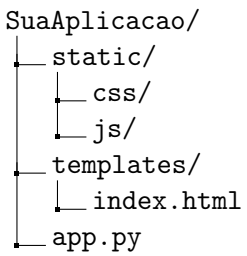


Figura 3.5: Árvore de Diretório do *Flask*.

O *Flask*, com auxílio de algumas outras ferramentas, possui uma estrutura padrão descrita na figura 3.5, onde é apresentada a árvore de diretórios padrão do *Flask*. As páginas HTML são naturalmente procuradas na pasta *template* e todos os outros atributos que só precisam ser carregados uma vez, como *scripts* em *JavaScript* e o CSS, ficam na pasta *static*. Por fim, as rotas e as lógicas da aplicação ficam no arquivo em *Python*.

3.3.5 *Virtualenv*

Virtualenv é uma ferramenta para criar ambientes *Python* isolados. O problema básico a ser solucionado é o de dependências, versões e indiretamente permissões. Estes problemas ocorrem quando aplicações diferentes precisam de, por exemplo, versões diferentes de uma mesma biblioteca *Python*, ou então alguma aplicação ainda não foi testada com a nova versão de tal biblioteca (BICKING, 2017). O último problema, o de permissões, envolve o fato de que um certo usuário não pode instalar pacotes no diretório global do *Python*. Para todos estes casos, o *Virtualenv* tem os seus próprios diretórios de instalação que não compartilham bibliotecas com outros ambientes *Virtualenv* e, opcionalmente, com o ambiente global (BICKING, 2017).

3.3.6 Jinja2

É uma linguagem moderna de "*templating*" para *Python* e *designer-friendly*, modelado a partir dos *templates Django*. É rápido, amplamente usado e seguro com o ambiente opcional de execução de *template* em *sandbox* (RONACHER, 2017). Dentre suas características, vale a pena citar seu sistema de prevenção de XSS, sintaxe configurável, herança de *templates* e facilidade de *debugging*.

3.4 *Oracle VM VirtualBox*

Virtualbox é uma aplicação de virtualização *cross-platform*. Pode ser instalado em dispositivos dos mais diversos OS, por exemplo, Windows, Mac e Linux. O *Virtualbox* estende as capacidades de um computador existente para que este possa funcionar com múltiplos sistemas operacionais (dentro de múltiplas máquinas virtuais) ao mesmo tempo. Também é a única solução profissional que está disponível gratuitamente (ORACLE, 2017).

3.5 *VirusTotal*

O *VirusTotal* inspeciona arquivos e URLs com mais de 60 antivírus e serviços de *blacklisting* de URLs e domínio, em adição a inúmeras ferramentas, para extrair sinais do conteúdo estudado. Qualquer usuário pode selecionar um arquivo de seu computador usando seu navegador e mandá-lo para o *VirusTotal*. Há também outros métodos além da paginas web pública, como extensão de navegadores e uma API programática. Contudo, a interface web tem a maior prioridade em escaneamento de itens sobre os outros métodos (VIRUSTOTAL, 2017).

Ao enviar um arquivo ou URL para análise, o resultado desta análise será compartilhado com os parceiros do *VirusTotal* para que estes possam melhorar seus próprios sistemas. Como um resultado, um usuário acaba contribuindo para a melhoria dos níveis globais de segurança em tecnologia da informação com a sua nova submissão à ferramenta (VIRUSTOTAL, 2017). Atualmente, esta ferramenta é gratuita e consegue fazer análise estática do artefato submetido. Além disso, qualquer usuário pode votar se um item submetido tem caráter malicioso ou não.

3.6 *Materialize*

Materialize é um *framework* responsivo de *front-end* baseado em Material Design do Google (GOOGLE, 2017). Seu objetivo é dar suporte aos desenvolvedores *front-end* para facilitar a criação de páginas *web*. Para isto, o *Materialize* dispõe de elementos de CSS prontos como os *helpers* para alinhamento de texto, classes de tabelas já definidas e responsivas, sistema de *grid* que suporta até quatro resoluções de tela diferentes (desde um celular até telas de *desktops*) e diversas animações para melhor visualização da página *web*.

Capítulo 4

Implementação

Para que o sistema funcione corretamente, é preciso fazer alterações e adições no código fonte do *Cuckoo*, além de configurações da máquina virtual *guest* e do *host* que terá esta ferramenta instalada. Em adição a essas modificações e configurações, ainda é preciso um sistema *web* que faça uso da API do *Cuckoo* para que o analista veja os dados necessários de maneira prática e rápida.

4.1 Modificações no *Cuckoo Sandbox*

Como já descrito anteriormente, na seção 2.1.4.2, o *Cuckoo* não faz nenhum tipo de predição de resultado. Embora o *Cuckoo* tenha diversas funcionalidades, tais como assinaturas criadas pela comunidade que ajudam na determinação de características do artefato submetido e utilização do *VirusTotal* para a inspeção de diversos antivírus, ainda não é possível identificar *zero-days*. Portanto, a criação de um método de detecção automática de um arquivo malicioso é necessária.

Uma vez que o *Cuckoo* faz, de maneira profunda, inspeção dinâmica, estática e até mesmo de rede de todo executável submetido, essa variedade de dados torna possível modelar de diversas maneiras os dados obtidos. É possível por se tratar de uma ferramenta em código aberto. Para acrescentar novas funcionalidades é importante entender o funcionamento de todo código que pode ser melhor estudado na documentação do *Cuckoo* (CUCKOO-FOUNDATION, 2017). Uma vez feito isso, pode-se agregar novas características ao software para a resolução do problema.

Como as funcionalidades não fazem parte do pensamento atual empregado no *Cuckoo* e envolve a mudança na estruturação lógica do código, foi feito um *fork*[1] do repositório atual do *Cuckoo* versão 2.0.4.4. A implementação desta nova biblioteca que, embora use boa parte da lógica empregada pelo *Cuckoo*, ainda assim se comporta diferente. Por outro lado, tanto a instalação como a configuração seguem o mesmo padrão empregado pelo *Cuckoo*.

[1] *Fork*: termo usado no protocolo de versionamento *git* para representar uma cópia do repositório.

4.1.1 Gerando um *Dataset*

Lembrando o que foi citado no capítulo 3, o *Cuckoo* possui a função de *reporting* e por ser uma ferramenta modular é possível criar qualquer tipo de relatório. Imaginando o *dataset* como uma forma de relatório, apenas com a diferença que neste caso será armazenado o conjunto dos resultados de todos os artefatos, o padrão é um relatório por arquivo submetido. Portanto, foi criado o módulo de geração de *dataset*, o qual é gerado em um diretório específico, para que assim possa ser tratado por diversos algoritmos de aprendizado de máquina.

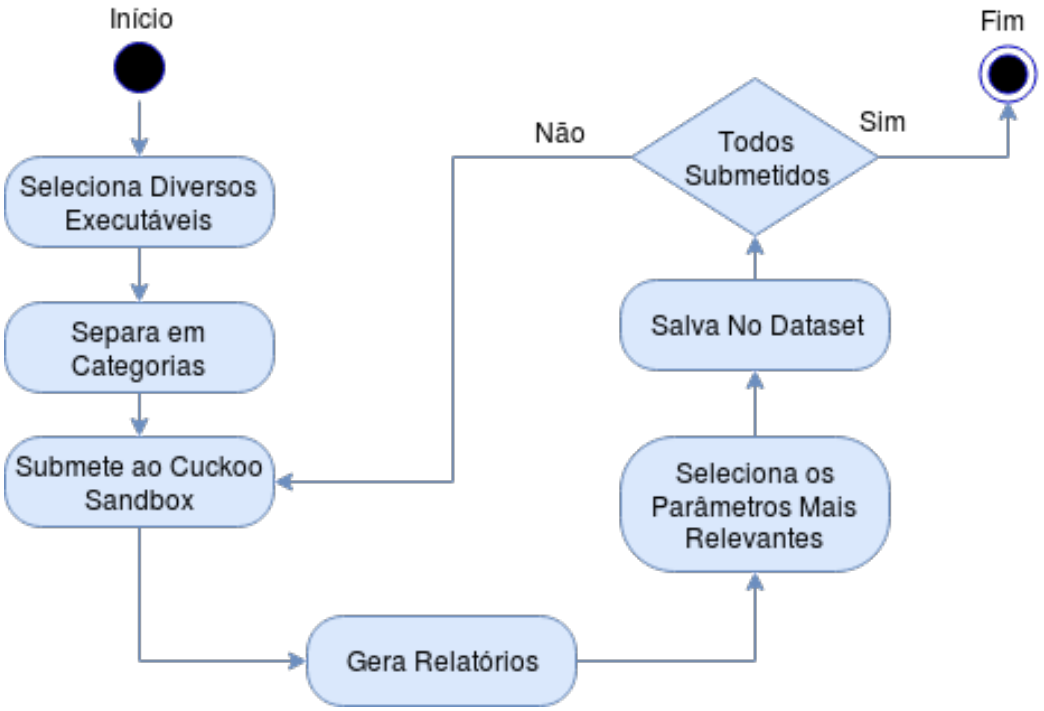


Figura 4.1: Geração de *Dataset*.

A figura 4.1 mostra o funcionamento da geração do *dataset*, popula um arquivo CSV linha por linha. É necessário que na submissão do artefato seja descrito o conhecimento prévio de qual categoria ele faz parte, seja benigno ou maligno, essa submissão é feita conforme a seguinte instrução:

```
(venv)$ cuckoo submit --custom benign <caminho/para/o/arquivo>
```

A classe que gera o *dataset* no *Cuckoo* está presente no anexo II.2, ou pode ser encontrada no repositório do *Github*[2] para maiores detalhes. O *dataset* é salvo no CWD de forma que a etapa de aprendizado de máquina, que será descrita a seguir, tenha acesso a ele facilmente.

[2] `https://github.com/relisfleuris/cuckoo-learning`.

4.1.1.1 Parâmetros Analisados

(SIKORSKI; HONIG, 2012) mostra em seu livro as 121 APIs mais comumente utilizadas por *malwares* em computadores com sistema operacional Windows e que essas deveriam ser as principais analisadas, no entanto para algoritmos de aprendizado de máquina 121 parâmetros acarreta no problema da multidimensionalidade, que aumenta a complexidade do algoritmo tornando o tempo de processamento maior.

Como estratégia para seleção dos dados são selecionadas 20 APIs que descrevem a maior parte do comportamento de um artefato, além destas, foram acrescentadas outras duas características, número de processos criados e número de arquivos baixados, essa abordagem foi baseada na tese de mestrado de (César Augusto Borges de ANDRADE, 2013). O autor desta tese conclui que algoritmos de aprendizado de máquina conseguem classificar *worms* e benignos com até 93,6% de acurácia usando um *dataset* de 10000 amostras. Desta forma, pode-se ver que os atributos escolhidos por ele são suficientemente bons por mostrar resultados acima de 90% de acurácia. No entanto, sua análise foi focada no comportamento dos *malware*, isto é, apenas na análise dinâmica, logo, foram acrescidos outros dois atributos, número de requisições para outros *hosts* (análise de rede) e a entropia do arquivo executável (análise estática).

A escolha da entropia como um atributo a mais nos modelos, se deve ao fato que grande parte dos arquivos maliciosos são empacotados aumentando assim a sua entropia, como mostrado em Elisan (2015), e embora alguns arquivos benignos possuam esta característica também esta característica é mais frequentemente encontrada em *malwares*. Outro atributo acrescido foi o número de requisições de *hosts*, pois durante as análises foi observado que alguns *worms* requisitam diversas URLs, padrão pouco comum em arquivos benignos.

Tabela 4.1: Atributos Considerados no *Dataset*.

Análise	Atributos
APIs (César Augusto Borges de ANDRADE, 2013)	*CreateFile, CreateMutant, CreateProcess, CreateRemoteThread, CreateService, DeleteFile, FindWindow, OpenMutant, OpenSCManager, ReadFile, ReadProcessMemory, RegDeleteKey, RegEnumKeyEx, RegEnumValue, RegOpenKey, ShellExecute, TerminateProcess, URLDownloadToFile, WriteFile, WriteProcessMemory*
Outras Análises Dinâmicas (César Augusto Borges de ANDRADE, 2013)	Número de processos criados e número de arquivos baixados
Análise de Rede	Número de requisições para outros *hosts* (novo)
Análise Estática	Entropia do Arquivo (novo)

4.1.2 Etapa de Aprendizado de Máquina

Uma vez em posse de um *dataset* no CWD para ser analisado, será possível assim rodar todos os algoritmos presentes no *sklearn* para diferentes formas de predição de um artefato. A predição de resultado foi pensada como uma etapa diferente na dualidade *reporting* e *processing*, sendo criada uma terceira etapa *learning*, a qual de maneira intermediária roda os algoritmos de ML no *dataset* e acrescenta no relatório as predições que então são enviadas para classe de *reporting*.

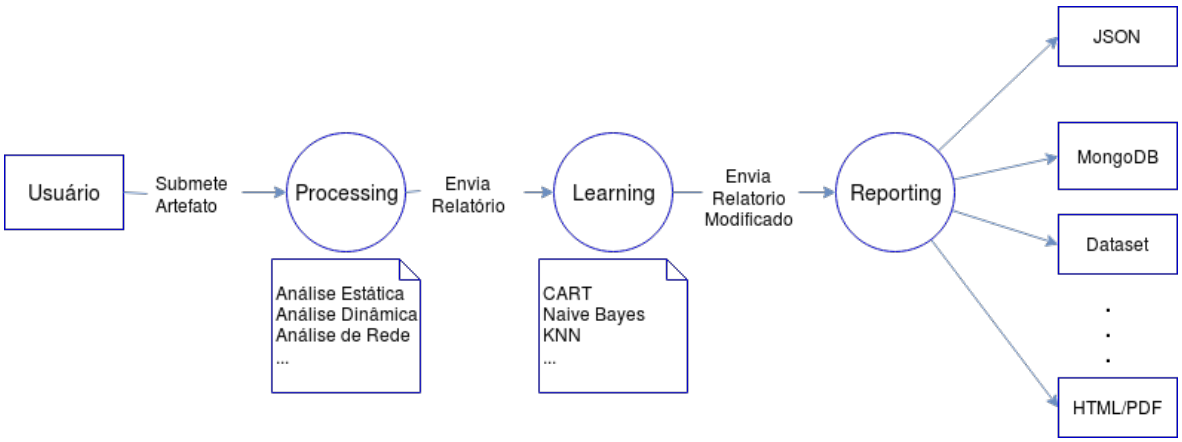


Figura 4.2: Predição de Malware.

Na figura 4.2 é possível observar a etapa de aprendizado sendo intermediária entre o processamento e a geração de relatório, a princípio a etapa de *reporting* possuiria informações como: análise dinâmica, análise estática, análise de rede, dentre outras. Com a adição da etapa de aprendizado é gerado mais um campo, o de *learning* o qual possui a predição e a probabilidade de ser uma classe (benigno ou *worm*). Desta maneira, pode-se manter o funcionamento anterior do *Cuckoo*, explicado no capitulo 3, para o caso em que não há nenhum algoritmo de ML habilitado, pois não haverá nenhuma alteração nas informações da etapa de *reporting*.

Esta nova funcionalidade foi idealizada, mantendo o mesmo padrão das etapas de *reporting* e *processing*, até mesmo a adição de um novo módulo segue o padrão já empregado no *Cuckoo*, ou seja, de maneira semelhante em que é acrescido um novo módulo na etapa de *reporting* é acrescido também nessa nova etapa, a qual foi construída desta maneira para que a solução possa ser mais flexível e de fácil adaptação. Pode-se ver no anexo II.1 como criar um novo módulo nesta etapa de *learning*.

4.2 Aplicação *Web*

Esta aplicação foi desenvolvida com o intuito de mostrar uma solução prática para sistemas de detecção automática de *malwares*, de maneira a prover uma solução intuitiva e de fácil uso para diversos públicos, principalmente para analistas que podem utilizá-lo para reduzir o tempo gasto em artefatos que a priori não possuem características de arquivos maliciosos. Como a missão da aplicação é capturar *malwares*, a solução foi batizada de *Venus Malware Trap*[3] (VMT) uma alusão à planta carnívora Vênus papa-moscas (*Venus flytrap* em inglês), que se assemelha a uma armadilha para insetos.

A parte de *Backend* da aplicação foi toda desenvolvida em *Flask*, *framework* explicado em 3.3.4, já a parte de *Frontend* foi feita com o auxilio do *Materialize* e do *Jinja2*, também detalhados

[3]O site pode ser acessado pelo link `https://venusmalwaretrap.com`

em 3.6. Todo o código desta aplicação está disponível no gitlab.com (link).

4.2.1 Estrutura da Aplicação

Esta aplicação foi desenvolvida utilizando a estrutura divisional especificada em (PICARD, 2017), normalmente usada para soluções de médio e grande porte, uma vez que a estrutura natural do *Flask* não comporta grandes aplicações. Esta pode ser visualizada na figura 3.5.

```
venus/
├── data/
│   └── model.py
├── api/
│   └── TODO
├── admin/
│   ├── templates/
│   └── controller.py
├── main/
│   ├── templates/
│   └── controller.py
├── profile
│   ├── templates/
│   └── controller.py
├── static/
├── templates/
└── app.py
```

Figura 4.3: Árvore de Diretório da Aplicação.

Nesta abordagem utilizando *Blueprints*[4], tem-se os controles das rotas individualmente em cada contexto, mantendo a lógica anteriormente explicadas do *Flask* na seção 3.3.4 onde em *templates* são colocadas as páginas HTML. Para cada contexto podemos ter páginas diferentes, o que auxilia na modularização da aplicação e também no seu controle. A estrutura da aplicação é dividida em 4 partes são elas:

- *Data*: aqui está presente o modelo para o banco de dados, podendo ser facilmente migrado para qualquer outro banco de dados pois utiliza *SQLAlchemy* (BAYER, 2017) como *framework* para banco de dados;
- *Main*: aqui fica o controle das rotas comuns a todos os usuários tais como login, página principal, dentre outras;
- *Profile*: nesta parte estão as rotas específicas para cada usuário, onde é preciso ter cuidado com controle de usuários e permissões;

[4] *Blueprints*: utilizado para separar a aplicação em módulos, ex: /admin /user

- *Admin*: nesta parte estão as rotas específicas para o controle dos usuários, tais como excluir e modificá-los.

Uma vez entendido o funcionamento do *backend* da aplicação, é importante agora observar o seu funcionamento como aplicação final, o qual será detalhado na próxima seção.

4.2.2 Funcionamento

O sistema *web* visa oferecer um design simples e de fácil navegação para os visitantes. O usuário registrado terá acesso a um sistema completo para análise e detecção de *malwares*, como também para o provedor do serviço que terá a possibilidade de montar o seu próprio banco de arquivos maliciosos com o auxílio da comunidade que utilizará o sistema. As principais etapas do funcionamento serão descritas a seguir:

Assim como muitos sistemas, o VMT também possui um sistema de *login* e registro de usuários para melhor integração com a submissão de artefatos, como será discutido a seguir.

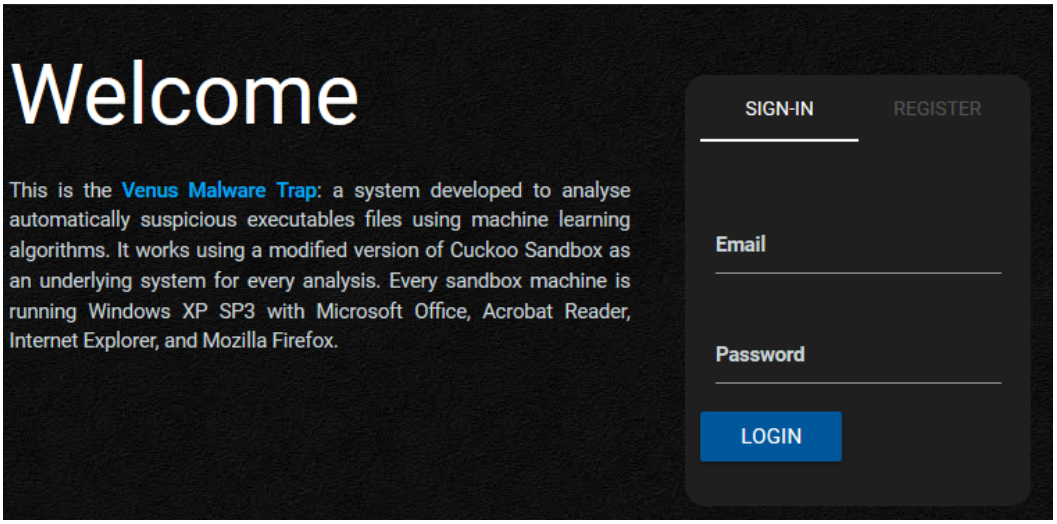


Figura 4.4: Página de *Login* e Registro.

Cada usuário poderá submeter executáveis para serem analisados, os quais são armazenados e atribuídos ao usuário que os submeteu. Existindo um controle de submissões que facilita a visualização de análises previamente submetidas. A página de envio de arquivos pode ser vista na figura a seguir.

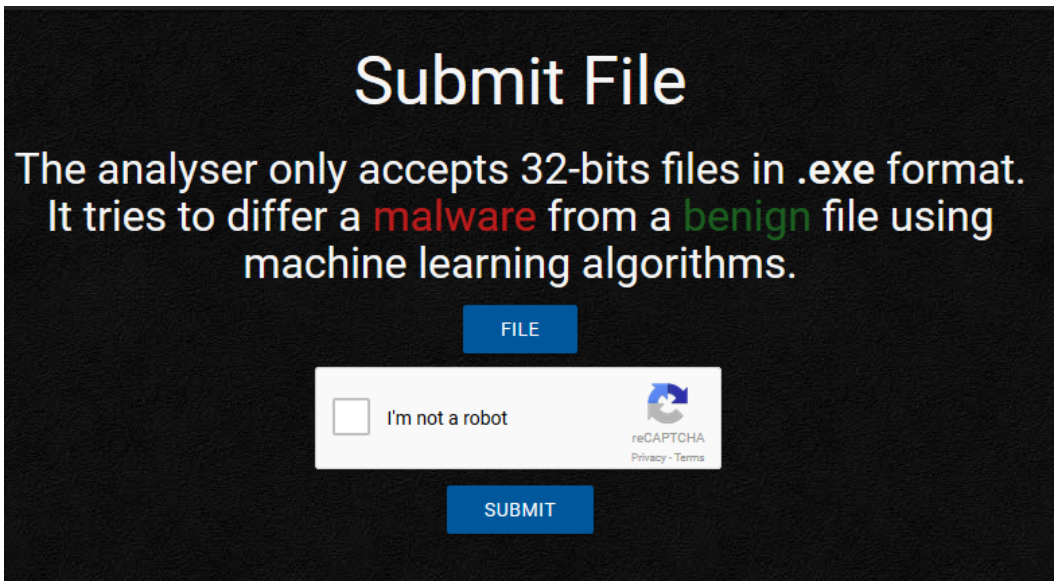
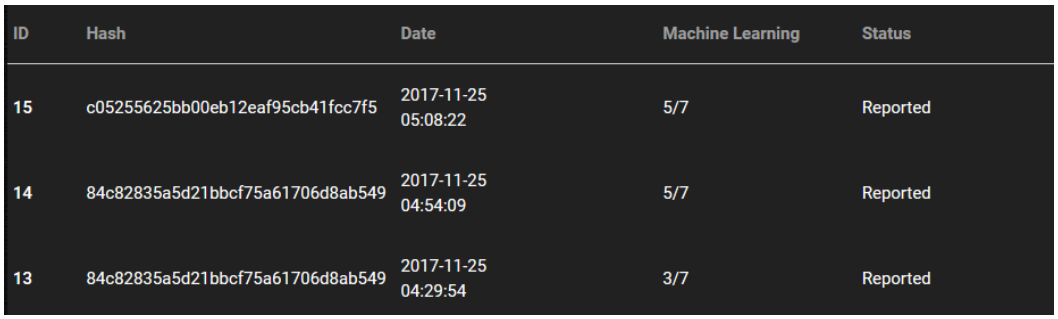


Figura 4.5: Página de Submissão.

Como dito anteriormente, o fato de possuir usuários facilita na separação dos artefatos submetidos, podendo tanto mostrar os artefatos enviados pelo usuário, quanto os artefatos submetidos na plataforma, como é possível ver na figura 4.6



ID	Hash	Date	Machine Learning	Status
15	c05255625bb00eb12eaf95cb41fcc7f5	2017-11-25 05:08:22	5/7	Reported
14	84c82835a5d21bbcf75a61706d8ab549	2017-11-25 04:54:09	5/7	Reported
13	84c82835a5d21bbcf75a61706d8ab549	2017-11-25 04:29:54	3/7	Reported

Figura 4.6: Páginas de Lista de Artefatos.

Existem alguns status para serem usados como *feedback* ao usuários no intuito de informar em que etapa do processo o artefato se encontra.

- *pending*: nesta etapa o executável ainda não foi submetido à *sandbox*;
- *running*: nesta etapa o executável está sendo processado pela *sandbox*;
- *completed*: nesta etapa o processamento já foi finalizado, porém os módulos de geração de relatório ainda não foram finalizados;
- *reported*: nesta etapa já é possível ver as análises realizadas pelo *Cuckoo*.

Ao clicar em um artefato dessa tabela é encaminhado para a página de análise do mesmo.

Nesta página é possível ver informações sobre análise estática, dinâmica e de rede além das predições desenvolvidas. Estas estão divididas em cinco abas, uma para cada, além da aba de sumário que contém informações mais relevantes de cada análise. Na figura 4.7 tem-se a página principal de cada análise, a página de sumário. E na figura 4.8 têm-se as informações de conexões feitas pelo *guest* a outros *hosts* tanto na rede que ele está quanto na Internet.

Já na figura 4.9, pode-se ver as assinaturas desenvolvidas pelas comunidades a partir do comportamento do artefato analisado durante sua execução na VM *guest*, e também é visível a árvore de processos criada na execução do item a ser analisado, ou seja, a análise dinâmica. Na quarta aba, representada pela figura 4.10, está a análise estática que mostra as DLLs importadas pelo executável, suas *strings*, os resultados do *VirusTotal*, entropia das seções, entre ouras informações relevantes. Por último, a aba de aprendizado de máquina, mostrada na figura 4.11, que mostra a predição de cada algoritmo de ML e seus valores de certeza em suas classificações, além de tabelas e gráficos para auxiliar o analista a avaliar os resultados a ele mostrados.

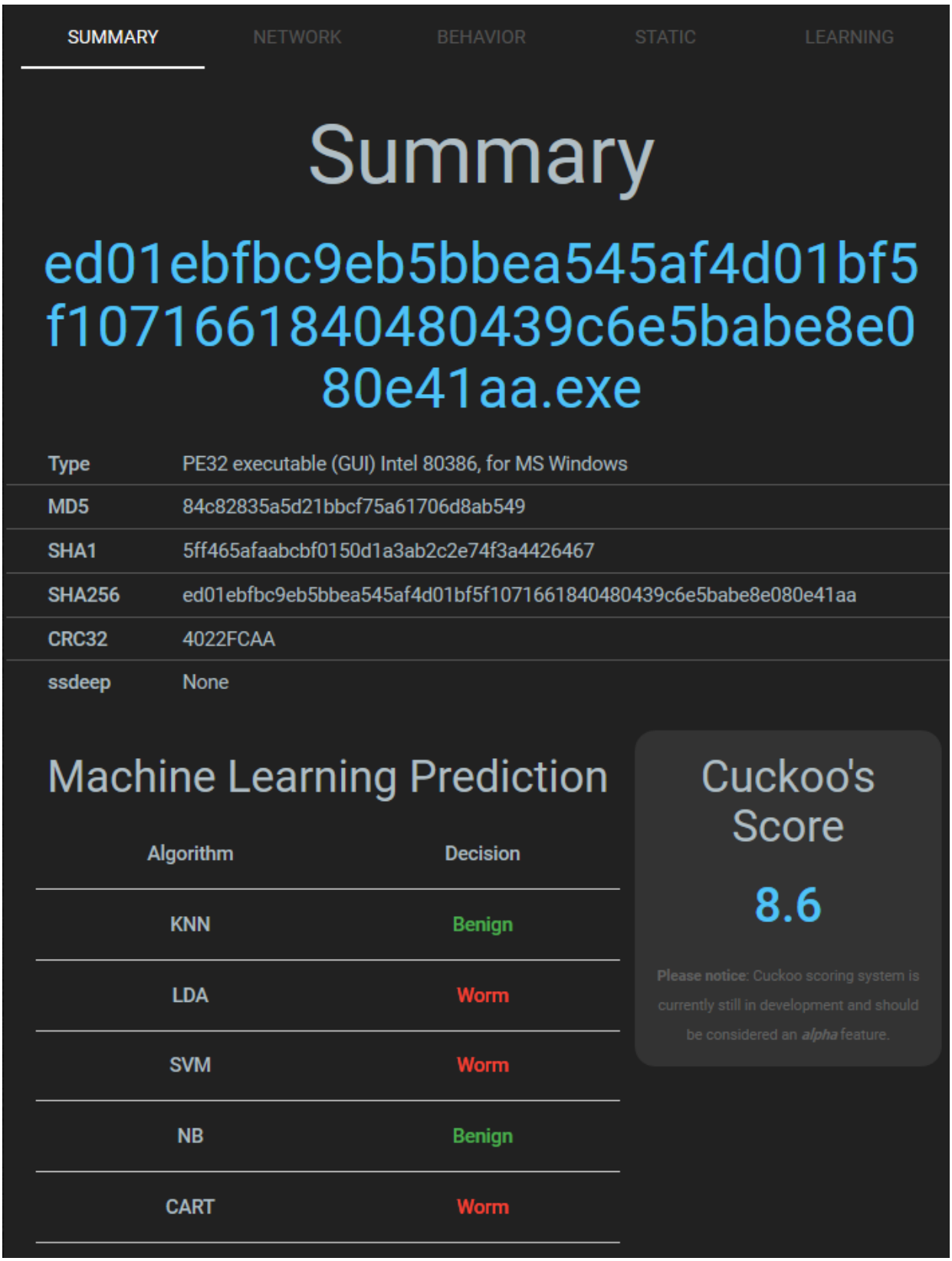


Figura 4.7: Exemplo de página de análise dos artefatos: Sumário.

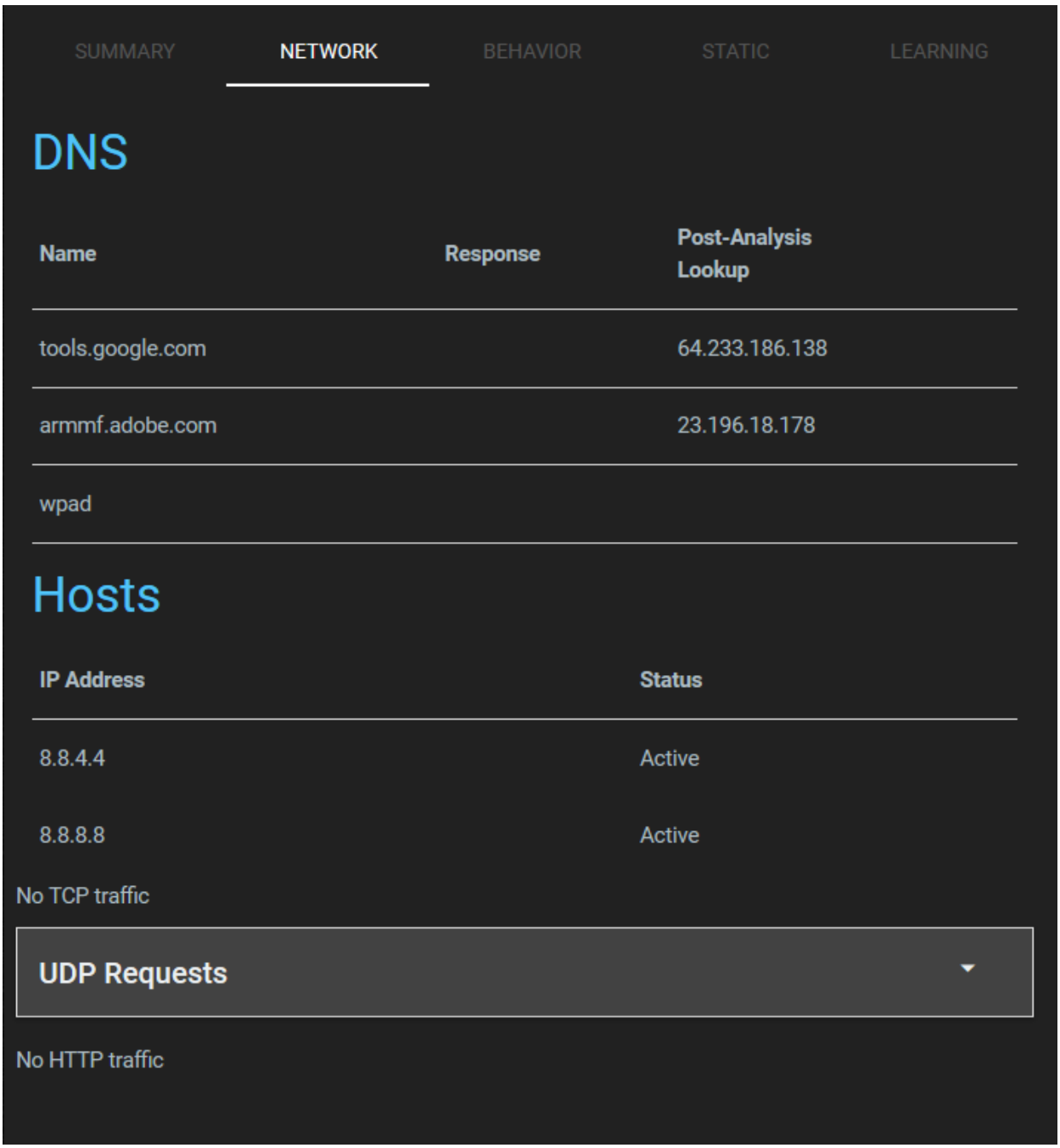


Figura 4.8: Exemplo de página de análise dos artefatos: Rede.

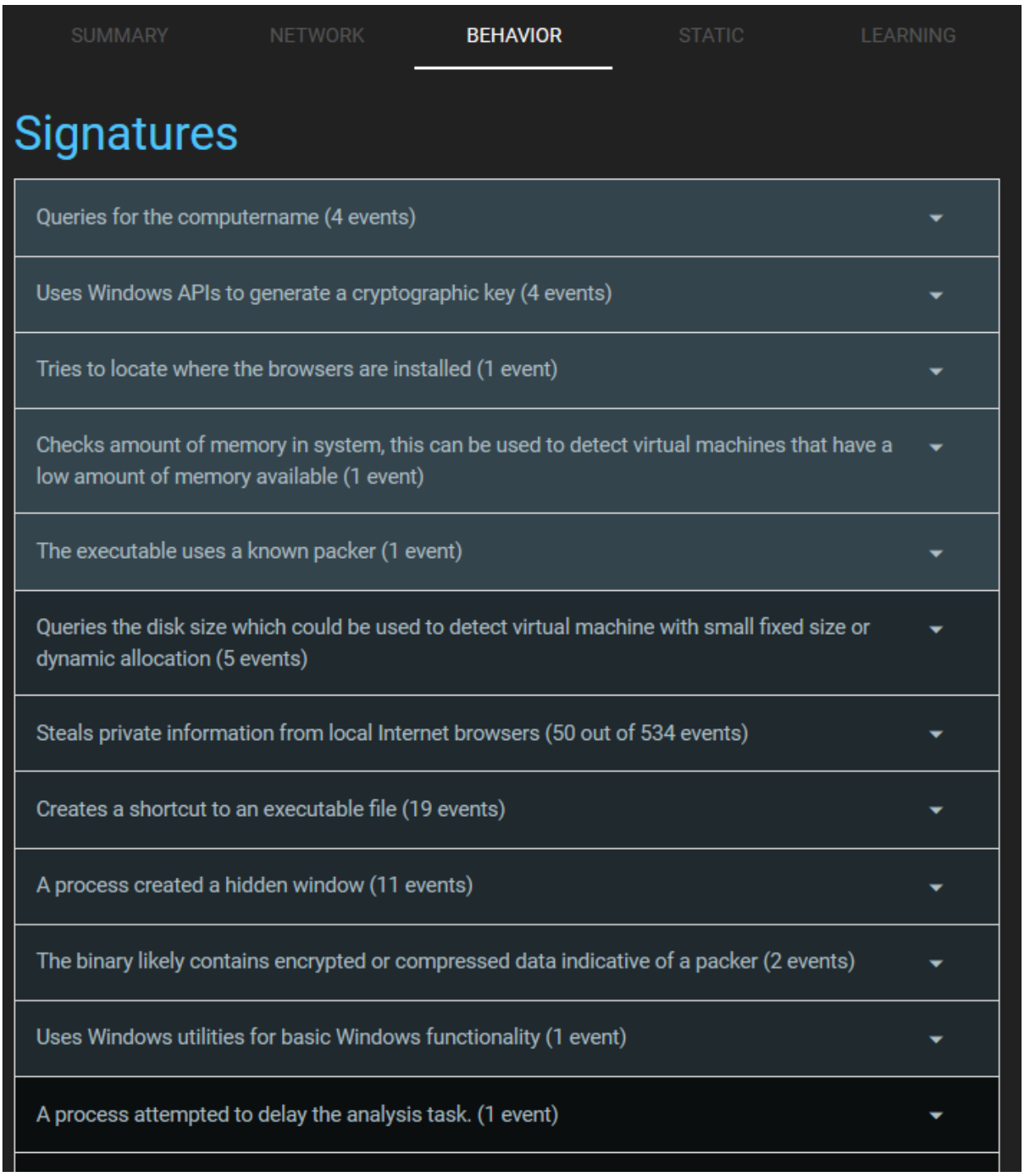


Figura 4.9: Exemplo de página de análise dos artefatos: Dinâmica.

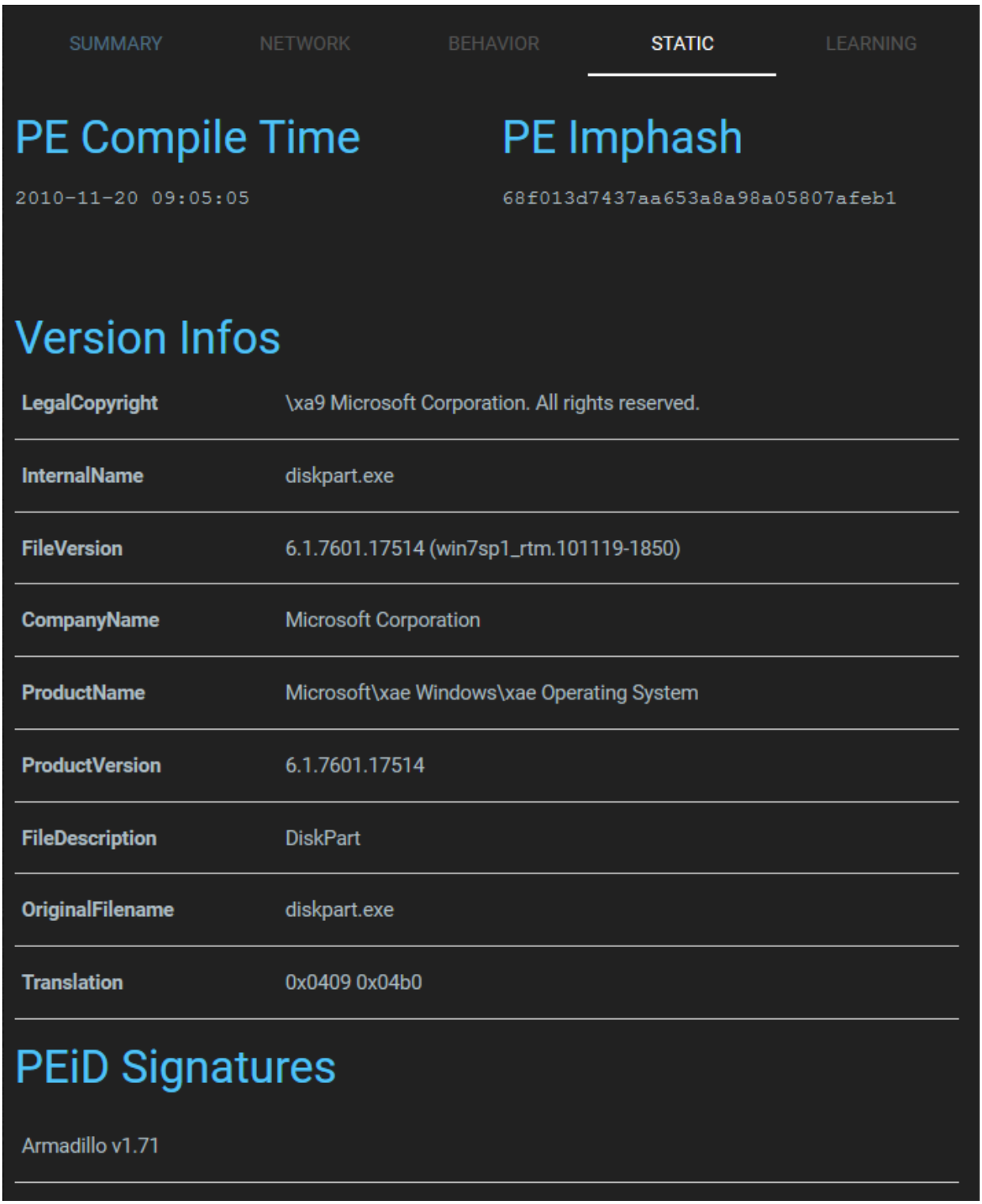


Figura 4.10: Exemplo de página de análise dos artefatos: Estática.

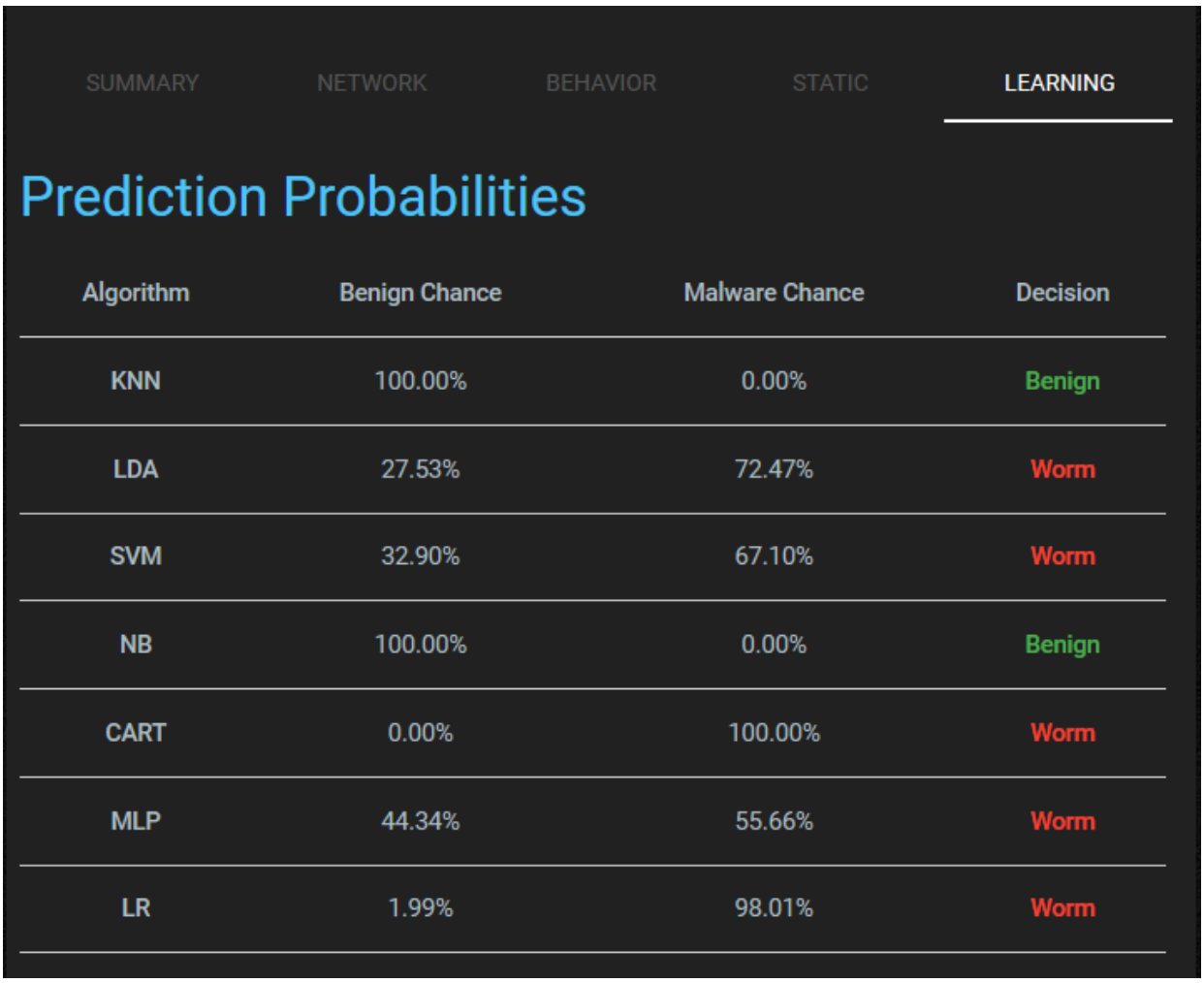


Figura 4.11: Exemplo de página de análise dos artefatos: Aprendizado de Máquina.

4.3 Arquitetura e Configuração do Ambiente Desenvolvido

A comunicação entre o sistema *web* e o *Cuckoo* é dada pela API, requisições de GET e POST são feitas pelo *backend* do VMT, então o retorno é tratado e são geradas as páginas HTML utilizando o *Jinja2*.

Tabela 4.2: Requisições Feitas na API do *Cuckoo*.

Recurso	Requisição HTTP	Descrição
/tasks/view/<id>	GET	Retorna os detalhes de uma análise a partir de seu ID..
/tasks/report/<id>	GET	Retorna o relatório gerado da análise a partir de seu ID. Pode-se opcionalmente especificar qual formato de relatório deve ser retornado, caso nenhum seja mencionado, o relatório retornado será o JSON.

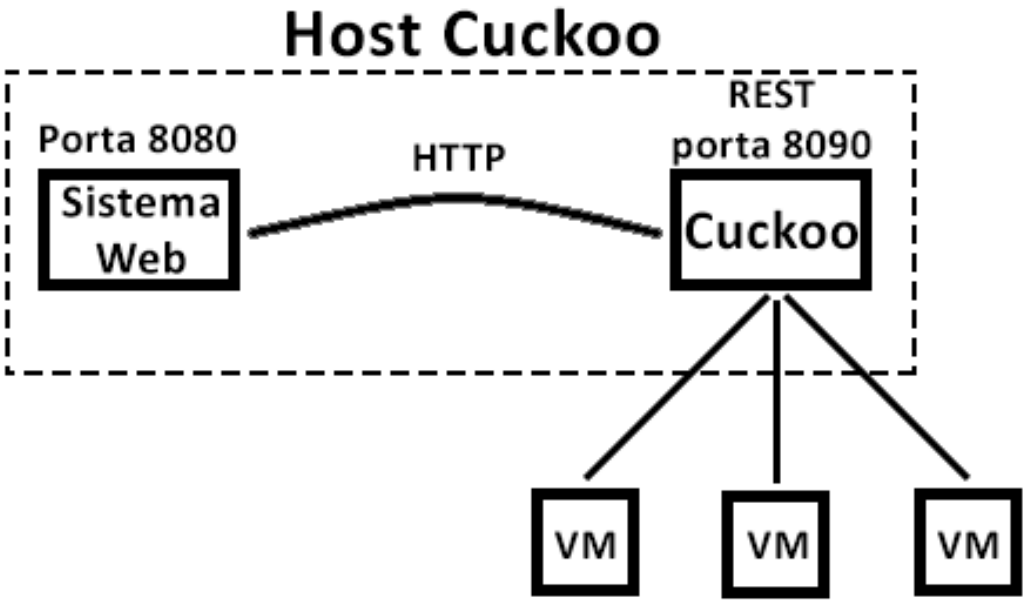


Figura 4.12: Arquitetura Venus Malware Trap com o Cuckoo.

Na arquitetura descrita na figura 4.12, é possível notar que tanto o VMT quanto o *Cuckoo* funcionam separadamente, embora a aplicação faça requisições HTTP (GET), de maneira que é possível tanto criar uma nova aplicação que consulte via API a versão modificada do *Cuckoo* quanto trocar a *sandbox* por alguma outra solução que também retorne os parâmetros necessários para o funcionamento do VMT.

4.3.1 Configuração para o *Cuckoo*

Além da instalação do *Cuckoo*, existe a necessidade de configurar as máquinas *guest* e *host* para que possa haver conexão entre elas. Além disso, arquivos de configurações do *Cuckoo* devem ser ajustados após sua instalação, podendo assim permitir funções específicas ou não, incluindo funções customizadas.

4.3.1.1 Máquina *Guest*

Usando o *VirtualBox* da *Oracle*, foi criada uma máquina virtual com as configurações mostradas na tabela 4.3. Há a possibilidade da adição de novas VMs a partir de novas instalações ou clones, tendo em mente que o IP deverá ser mudado para não haver conflito entre eles, permitindo assim que o *Cuckoo* possa executar múltiplas análises ao mesmo tempo.

Tabela 4.3: Configurações da VM no *VirtualBox*.

Sistema Operacional	Windows XP Service Pack 3 (32-bits)
Memoria RAM	0.5 GB
Memoria de Vídeo	18 MB
Storage	10,00 GB
Rede	*Host-Only*
IP	192.168.56.101

Para a melhor execução de *malwares*, um dispositivo necessita estar o mais desprotegido possível, isto é, ferramentas de segurança como o *firewall* e antivírus não instaladas, ou mal configuradas, ou desatualizadas. Há também a necessidade de programas comumente explorados por atacantes como o *Acrobat Reader*, *Internet Explorer*, pacote *Office* da *Microsoft*, *Mozzila Firefox*, *Google Chrome* e *Windows Media Player*. Deste modo, foi criada uma máquina virtual *guest* para o *Cuckoo* com o sistema operacional Windows XP SP3 com as configurações e programas supracitados além do *agent* do *Cuckoo*, conforme orientado pela documentação do (CUCKOO-FOUNDATION, 2017).

4.3.1.2 Máquina *Host*

Assim como a máquina *guest*, o *host* precisa ser configurado para o bom funcionamento do *Cuckoo*. Além da instalação das dependências do *Cuckoo* e do *Flask* (que pode ser vista no anexo deste documento), existe a necessidade de configurar a rede do *host* para permitir conexões à internet dos *guests*. Neste projeto, será usado o roteamento global simples instruído, que faz uso do iptables[5] do Linux. Há também configurações mais complexas como o roteamento por redes Tor, por *Virtual Private Network*[6], entre outras além do bloqueio de toda a conexão com a rede externa (CUCKOO-FOUNDATION, 2017).

4.3.1.3 Arquivos de Configurações do *Cuckoo*

O arquivo principal de configuração, o "cuckoo.conf" é para a configuração geral de comportamento deste e opções de análises (CUCKOO-FOUNDATION, 2017). Neste, as únicas configurações diferentes das padrões modificadas foram as de banco de dados e a de *timeout* padrão. As informações da VM discutida na seção 4.3.1.1 são passadas para o "virtualbox.conf", assim o *Cuckoo* saberá o IP, a interface de rede da *host* que será usada para comunicar-se com a *guest*, o sistema operacional da *guest*, entre outras informações. Na configuração do "reporting.conf" foi adicionado o campo *dataset* para que o usuário possa escolher se uma nova análise terá seus dados colocados no *dataset* ou não. Por último, um novo arquivo de configuração foi criado, o "learning.conf",

[5]Iptables: Ferramenta administrativa para filtragem de pacotes IPv4 e NAT.

[6] *Virtual Private Network* ou VPN: é um túnel seguro entre dois ou mais dispositivos.

nele pode-se colocar o nome do arquivo do *dataset*, dos atributos do deste e habilitar ou não cada algoritmo de aprendizado de máquina usados neste projeto.

Detalhes dos arquivos de configurações ".conf" do *Cuckoo* podem ser encontradas nos anexos I.3 e II.1.

Com as configurações feitas no *Cuckoo*, o sistema *web* integrado e funcionando, a máquina virtual *guest* instalada com acesso à internet e as configurações de rede da máquina *host* em funcionamento, pode-se gerar o *dataset*. Após a criação do *dataset* completo, é permitida a análise de artefatos pelo sistema *web*.

Capítulo 5

Análises e Resultados

O projeto consiste em analisar apenas *worms* e códigos benignos, isto é, os algoritmos classificariam uma nova amostra como *worm* ou benigno. Para verificar se o modelos gerados são bons os suficientes, foram enviados dois *ransomwares* com características de *worms* para a análise no sistema VMT.

Os artefatos a serem analisados para a criação dos *datasets* foram enviados para o Cuckoo, que fez o paralelismo em quatro máquinas virtuais com o sistema operacional Windows XP SP3 como descritas na seção 4.3.1.1. Com os *datasets* em CSV completos de amostras, pode-se começar a fazer o treinamento dos diversos algoritmos de aprendizado de máquinas assim como suas devidas análises. A figura 5.1 mostra um trecho do *dataset* do primeiro experimento.

```
26,4,4,0,1,2,78,4,2,14,2,0,0,0,146,0,9,0,11,4,5,3,2,7.9337,worm,Worm.Win32.AutoRun.bco.
19,4,3,0,1,0,78,4,3,12,2,0,0,0,106,0,6,0,10,4,5,0,1,6.4339,worm,Worm.Win32.AutoRun.bcy.
25,4,4,0,1,2,78,4,2,14,2,0,0,0,146,0,9,0,10,4,5,2,2,6.2473,worm,Worm.Win32.AutoRun.bgc.
1,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0,3,0,0,0,5,0,2,5.916,worm,Worm.Win32.AutoRun.bhq.
18,0,1,1,1,5,192,0,4,5,0,0,0,0,96,0,3,0,5,9,5,2,8,7.9724,worm,Worm.Win32.AutoRun.bfs.
26,4,4,0,1,2,78,4,2,14,2,0,0,0,145,0,9,0,11,4,5,3,2,6.3063,worm,Worm.Win32.AutoRun.bgd.
1,0,0,0,0,27,2647,0,0,0,0,0,0,0,2,0,0,0,0,0,5,0,0,6.0036,worm,Worm.Win32.AutoRun.bhr.
25,4,4,0,1,2,78,4,2,14,2,0,0,0,146,0,9,0,10,4,5,2,8,6.3394,worm,Worm.Win32.AutoRun.bhv.
16,0,4,0,0,2,0,0,0,4,0,0,4,0,70,0,3,0,0,0,5,2,1,7.9254,worm,Worm.Win32.AutoRun.bia.
26,4,4,0,1,2,78,4,2,14,2,0,0,0,145,0,9,0,11,4,5,3,2,6.1631,worm,Worm.Win32.AutoRun.bhw.
25,4,4,0,1,2,78,4,2,14,2,0,0,0,146,0,9,0,10,4,5,2,2,6.2192,worm,Worm.Win32.AutoRun.bhy.
```

Figura 5.1: Parte do *Dataset* do Primeiro Experimento.

5.1 *Worms versus* Benignos

Em primeira instancia, foram enviados para a análise no *Cuckoo* 933 *worms* e 935 programas benignos para as máquinas virtuais *Guests*, gerando um *dataset* balanceado (apenas 2 programas benignos a mais que *worms*).

5.1.1 Validação Cruzada (*Croos-Validation*)

Neste projeto foram utilizados sete algoritmos de aprendizado de máquina, são eles o LR, LDA, SVM, CART, KNN, NB, e MLP. A escolha foi feita para que se tenha uma quantidade considerável de algoritmos diferentes, incluindo árvores de decisão como o CART, redes neurais como o MLP.

5.1.1.1 Acurácia

Utilizando *cross-validation* de 10 dobras (*10 fold*) para avaliar a acurácia dos algoritmos foi obtido os seguintes resultados na tabela 5.1, nela também é possível ver o desvio padrão (DP) de cada acurácia.

Tabela 5.1: Acurácia dos Algoritmos.

Algoritmo	Acurácia	Desvio Padrão
CART	0,878143	0,023884
KNN	0,818595	0,024602
LDA	0,795803	0,028409
LR	0,813879	0,029732
MLP	0,862081	0,030574
NB	0,687396	0,026177
SVM	0,766349	0,033499

Nota-se que entre os sete algoritmos de ML testados, os de melhor desempenho baseado apenas na métrica acurácia foram o CART, KNN, LDA, LR e MLP com valores acima de 79,58% de acurácia. Outro ponto a ser observado é o resultado do Naive Bayes, que obteve apenas 68,74% de acurácia (7,90% a menos que o segundo pior algoritmo, o SVM). Isto se dá pelo fato do algoritmo ser "ingênuo" e considerar que os atributos usados para o experimento são independentes entre si quando, na realidade, não são.

No diagrama de caixa a seguir, tem-se os sete algoritmos de ML com suas acurácias plotadas junto com o desvio padrão e variância. Percebe-se, assim como na tabela 5.1, que o CART tem o melhor desempenho quando usado a acurácia como métrica, além disso, seu desvio padrão é o menor entre os algoritmos, tendo assim o segundo e terceiro quartis pequenos.

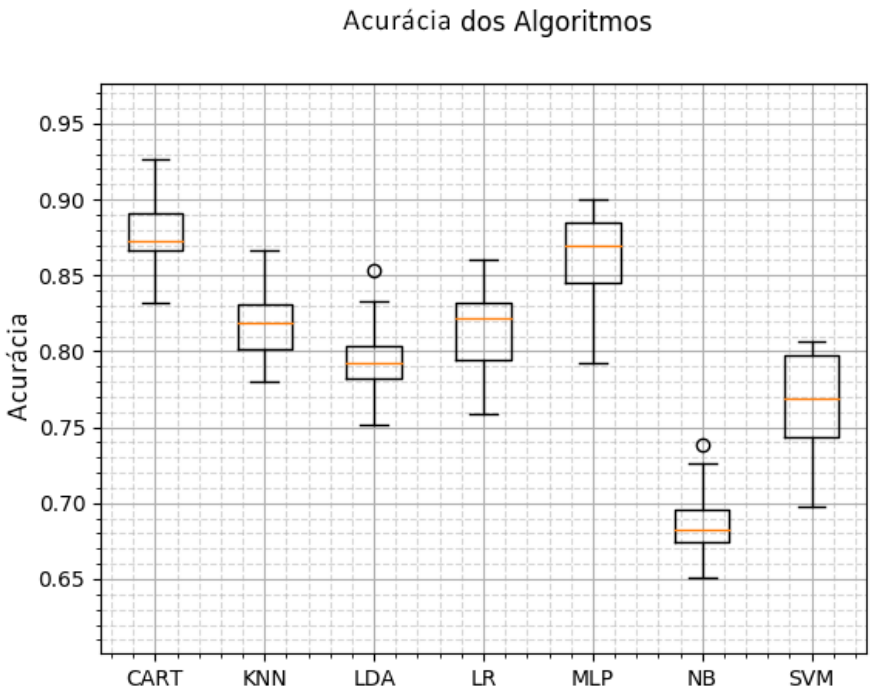


Figura 5.2: Diagrama de caixa dos diferentes algoritmos (Acurácia).

5.1.1.2 Precisão, *Recall* e *F-Measure*

Apesar da acurácia ser uma métrica comum para medição de qualidade de um algoritmo de aprendizado de máquinas em um certo *dataset*, como visto em (KUBAT et al., 1998) e (FREITAG, 2000), deve se considerar outras métricas para medição de desempenho. Neste projeto, um valor relevante a se considerar é o *recall* devido ao fato de que, idealmente, não se pode deixar passar *worms* como benignos na análise.

Nas figuras a seguir têm-se as curvas de precisão-*recall* dos modelos gerados a partir do *dataset*. Nesses gráficos é visível que a precisão decresce quando o *recall* aumenta. Entretanto, a precisão, em todos os casos, tem valores de 0,5 quando o *recall* é 1, significando, deste modo, que todos os *worms* são classificados corretamente, mas todos os benignos também são. Para um melhor balanço entre precisão e *recall*, decidiu-se usar o limiar de decisão de 0,5 na probabilidade de um artefato ser *worm*, resultando nos valores de precisão, *recall* e *F-measure* encontrados na tabela 5.2.

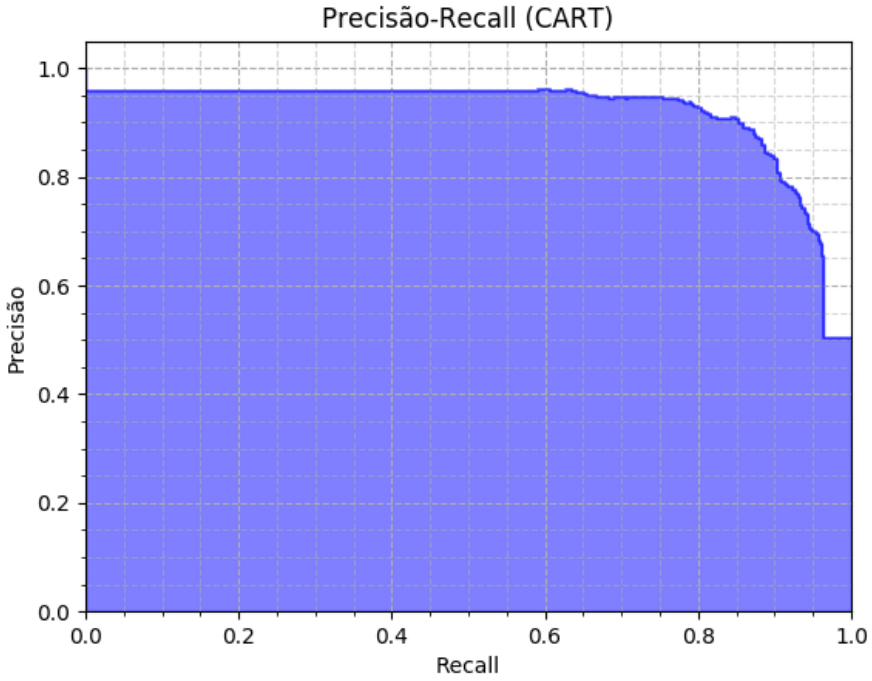


Figura 5.3: Curva de Precisão e *Recall* do CART.

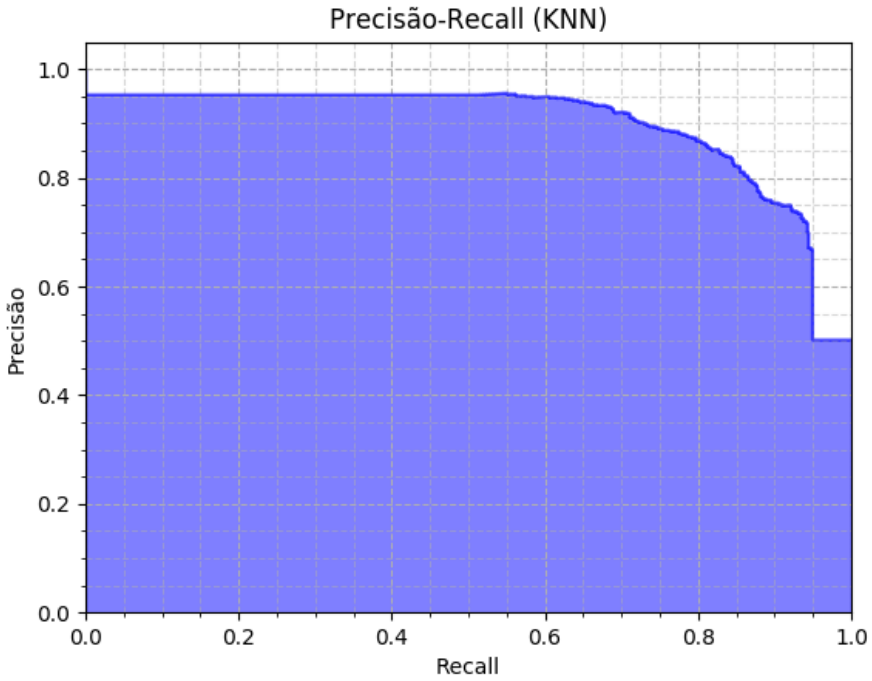


Figura 5.4: Curva de Precisão e *Recall* do KNN.

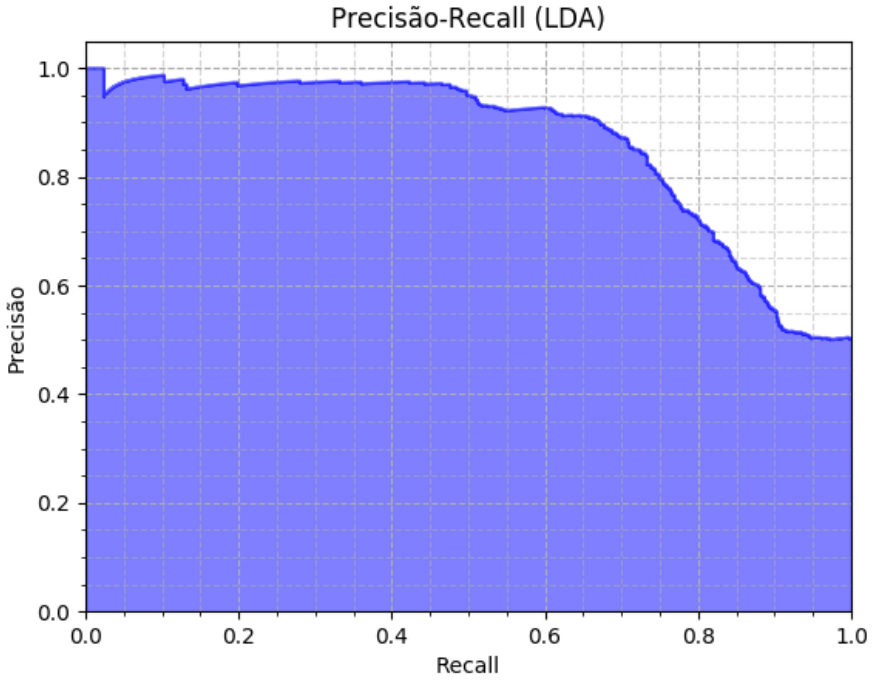


Figura 5.5: Curva de Precisão e *Recall* do LDA.

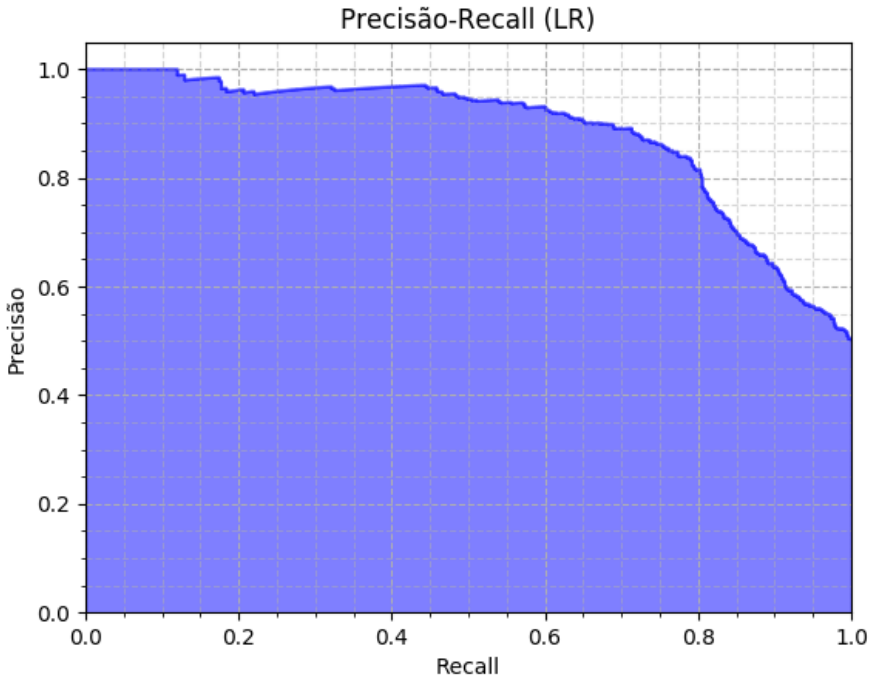


Figura 5.6: Curva de Precisão e *Recall* do LR.

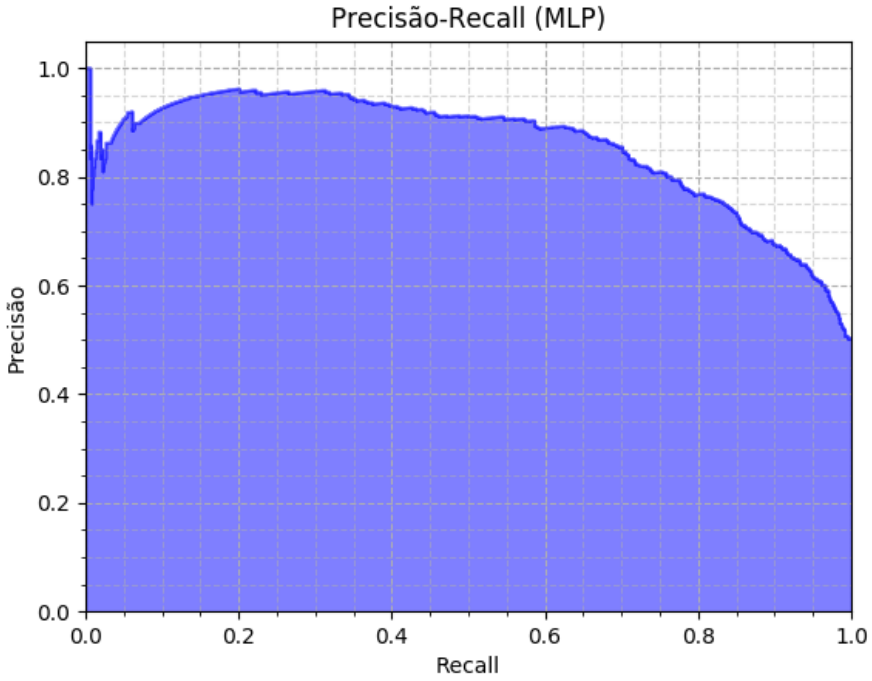


Figura 5.7: Curva de Precisão e *Recall* do MLP.

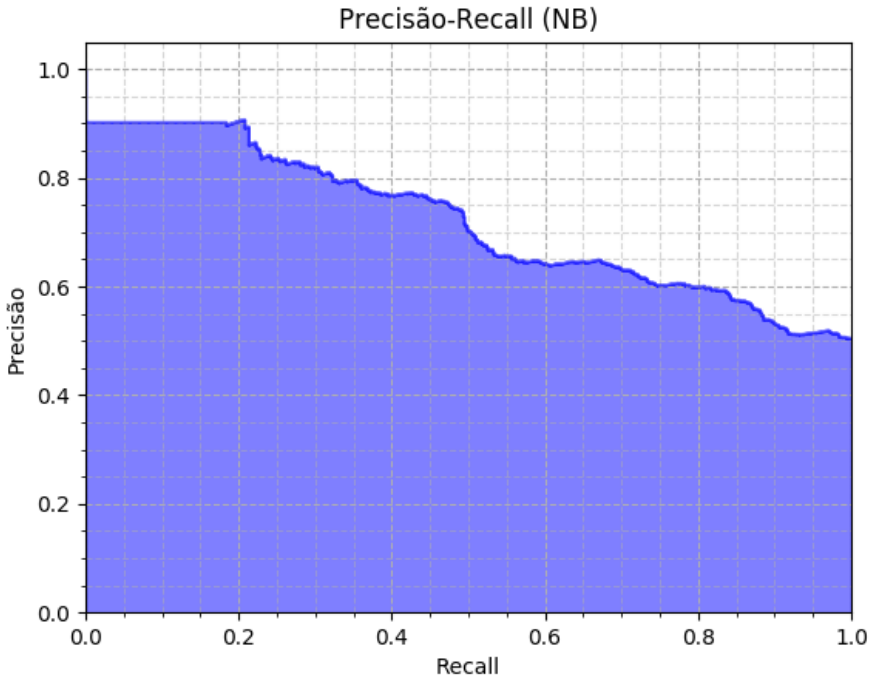


Figura 5.8: Curva de Precisão e *Recall* do NB.

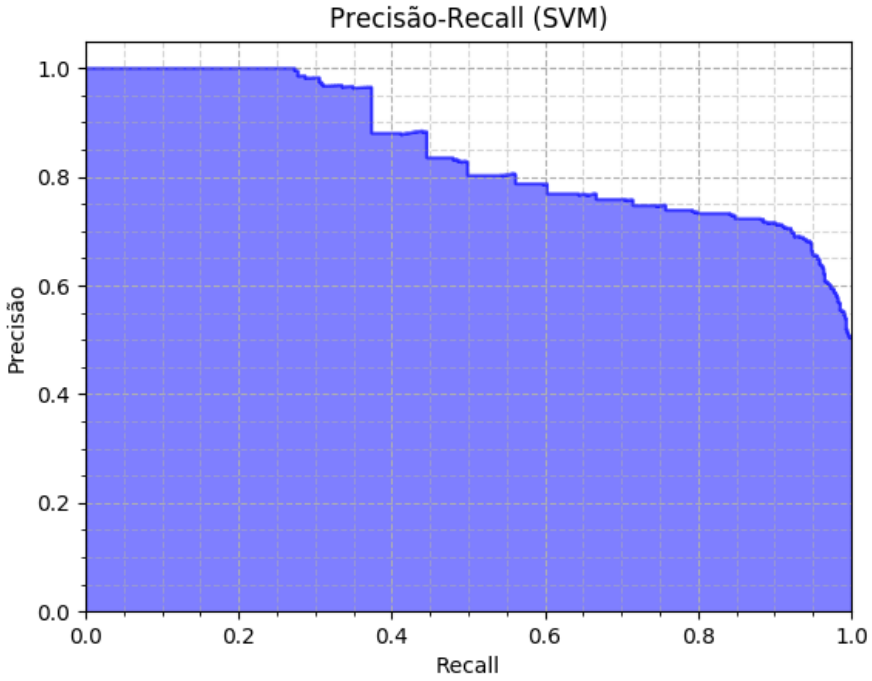


Figura 5.9: Curva de Precisão e *Recall* do SVM.

Os resultados encontrados das métricas de desempenho de algoritmos de aprendizado de máquina são vistos na tabela 5.2. Nota-se que assim como a tabela 5.1, os valores têm seus respectivos desvios padrões representados na tabela. Ao comparar resultados da tabela 5.1 com a tabela 5.2, pode-se concluir que precisão não é um bom indicador por si só, já que o Naive Bayes obteve, para a classe *worm* 92,71% de precisão, um valor considerado alto, mas com *recall* de 41,09%, resultando em um *F-measure* de apenas 0,59996. Além disto, o CART, em geral, obteve os melhores resultados nessas três métricas, assim como para acurácia. Outro algoritmo a ser observado por bons resultados é o MLP, com precisão acima de 90% e *recall*, mesmo com a precisão alta, de 78,39%, consequentemente gerando um *F-measure* de 0,847521.

Tabela 5.2: Precisão, *recall* e *F-Measure* dos algoritmos de ML para a classe *worm*.

Algoritmo	Precisão	DP	*Recall*	DP	*F-Measure*	DP
CART	0,888019	0,033842	0,864197	0,041296	0,875362	0,030293
KNN	0,881109	0,032831	0,737374	0,054717	0,801547	0,036027
LDA	0,845536	0,043613	0,724469	0,056906	0,778909	0,041619
LR	0,876127	0,039946	0,730516	0,057901	0,795602	0,042974
MLP	0,911876	0,043792	0,783931	0,062880	0,847521	0,050397
NB	0,927141	0,054872	0,410893	0,059996	0,565951	0,054657
SVM	0,706978	0,039686	0,914924	0,023557	0,796986	0,029184

Os diagramas de caixas das figuras 5.11, 5.10 e 5.12 representam, respectivamente, a precisão,

o *recall* e o *F-measure* para a classe *worm*. Nos diagramas é mais fácil a visualização dos valores encontrados na tabela anterior, deste modo, é visível que o NB tenha a maior precisão entre os algoritmos comparados e o maior desvio padrão. Já no segundo diagrama, observa-se o baixo *recall* do NB e, concorrentemente, os altos valores de *recall* do CART e do SVM. Por fim, no último diagrama de caixas, que mostra o *F-measure* dos sete algoritmos, tem-se a confirmação da superioridade do CART e do MLP sobre os outros já que seus valores de precisão e *recall* são altos e próximos uns dos outros como discutido na seção 2.2.4.3.

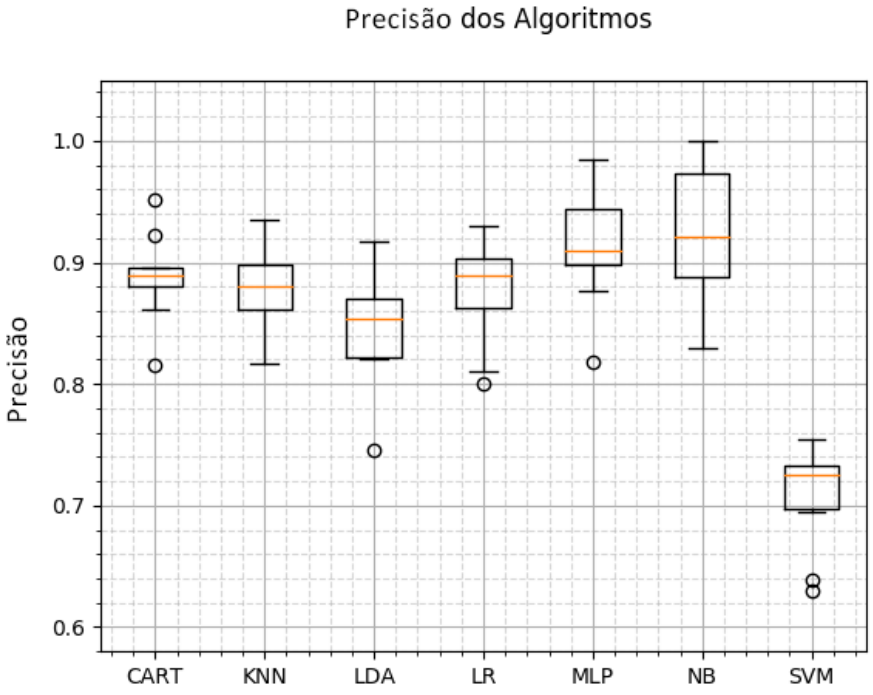


Figura 5.10: Precisão em diagrama de caixas dos diferentes algoritmos.

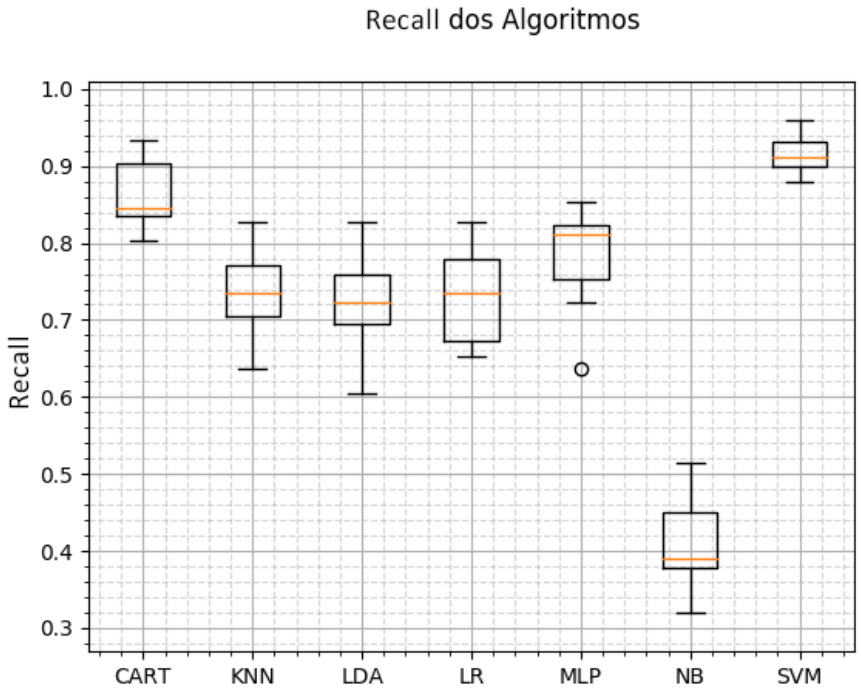


Figura 5.11: *Recall* em diagrama de Caixas dos diferentes algoritmos.

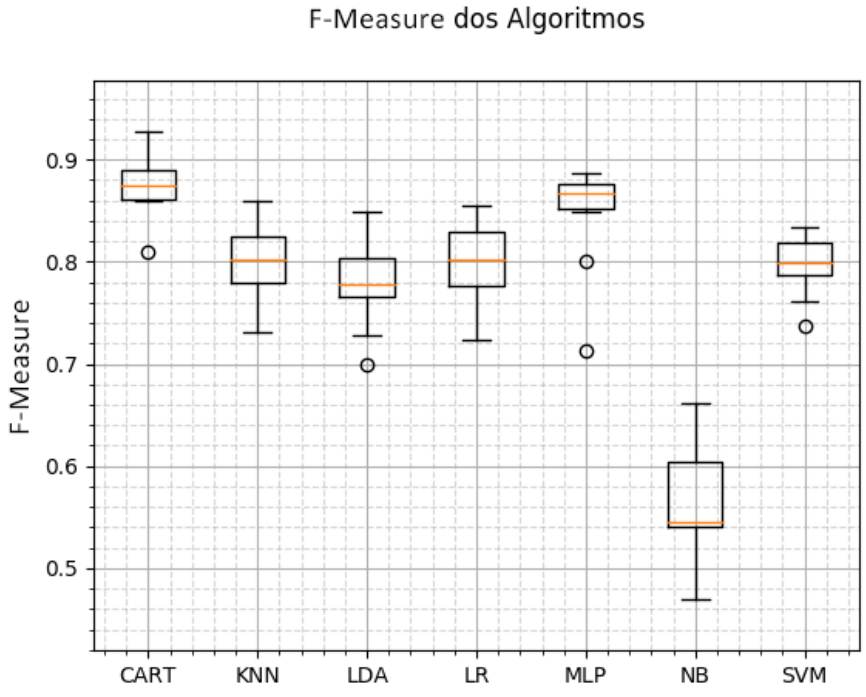


Figura 5.12: *F-Measure* em diagrama de caixa dos diferentes algoritmos.

5.1.2 Validação dos resultados

Nesta seção, para validar os resultados obtidos na seção anterior, foi utilizada a técnica de *holdout* onde os 20% do *dataset* separados para validação foram testados nos modelos gerados com o intuito de verificar se estes estão de alguma forma muito específicos para a coleção de dados de treinamento. Na tabela 5.3 é possível verificar as pontuações obtidas pelos algoritmos de aprendizado de máquina.

Tabela 5.3: Acurácia, precisão, *recall* e *F-measure* dos algoritmos de ML para a classe *worm* com a técnica de *Holdout*.

Algoritmo	Acurácia	Precisão	*Recall*	*F-Measure*
CART	0,87166	0,84615	0,90164	0,87302
KNN	0,84492	0,88820	0,78142	0,83140
LDA	0,79144	0,80347	0,75956	0,78090
LR	0,79947	0,82143	0,75410	0,78632
MLP	0,88235	0,90173	0,85246	0,87640
NB	0,67112	0,88462	0,37704	0,52874
SVM	0.77807	0,70000	0,95628	0,80831

Em geral, os resultados obtidos referentes ao conjunto de teste foram semelhantes aos obtidos pela validação cruzada, mostrando que o treinamento não foi tendencioso. Alguns casos obtiveram resultados até melhores, por exemplo o CART possui um *recall* médio de 86% e foi obtido 90% com as amostras de teste, por outro lado a precisão média é de 88% e o resultado obtido foi de 84%, o que causa 87% de *F-measure* para ambos os casos. Também com base nas amostras de teste, com o intuito de melhorar a visualização dos resultados obtidos, foi gerada a matriz de confusão que será tratada com mais detalhes a seguir.

5.1.2.1 Matriz de Confusão

A matriz de confusão, como discutida na seção 2.2.4.2, mostra valores importantes que ajudará a compreender a qualidade de um certo algoritmo para o *dataset* encontrado neste projeto. Primeiramente, na figura 5.13 tem-se a matriz de confusão do CART que mostra 90% das amostras de *worms* são classificadas corretamente, e 84% das predições das amostras benignas estavam certas.

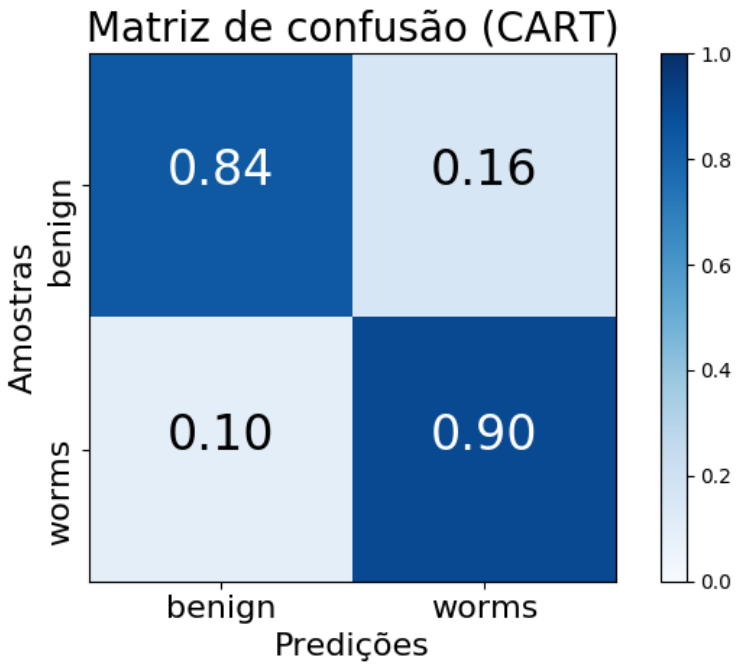


Figura 5.13: Matriz de Confusão do CART.

Na figura 5.14, que mostra a matriz de confusão normalizada para o k próximos vizinhos, é notável que 91% das amostras benignas são classificadas corretamente e das amostras *worms*, 78% delas são classificadas como *worms* também.

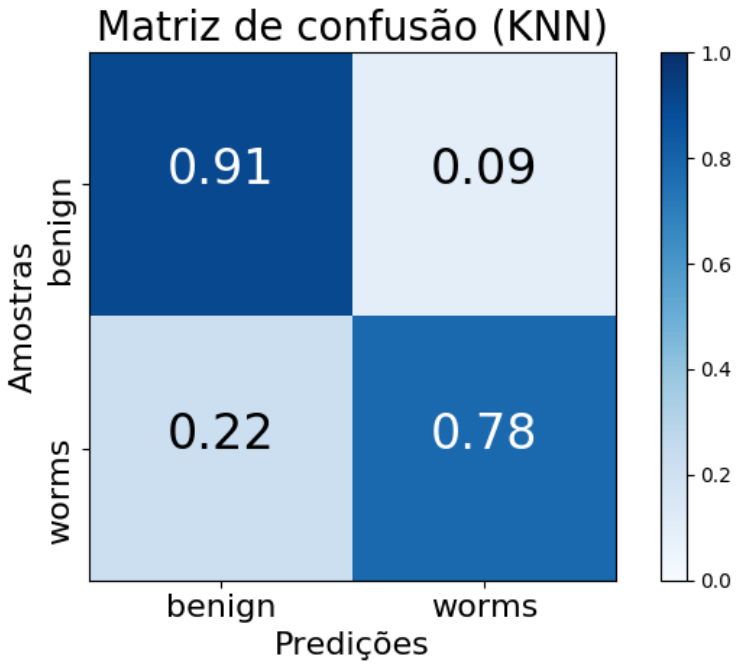


Figura 5.14: Matriz de Confusão do KNN.

A matriz de confusão para o algoritmo LDA, mostrado na figura 5.15, tem resultados piores

que os dois algoritmos anteriores. Isto é, sua classificação de amostras benignas acerta 82%, 9% a menos que o KNN, enquanto para classificação de *worms* o LDA obteve 76%, apenas 2% a menos que o algoritmo anterior.

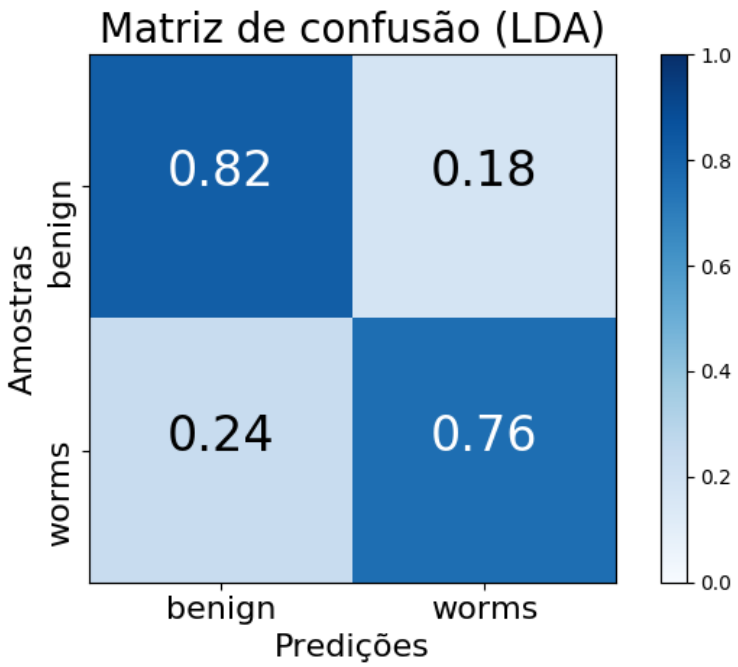


Figura 5.15: Matriz de Confusão do LDA.

Na figura a seguir, tem-se a matriz de confusão para o LR que tem 84% das amostras benignas sendo categorizadas corretamente pelo algoritmo mas apenas 75% dos *worms* classificados realmente como *worms*. Mostrando que o algoritmo esta enviesado, classificando as amostras, benignas ou não, como benignas.

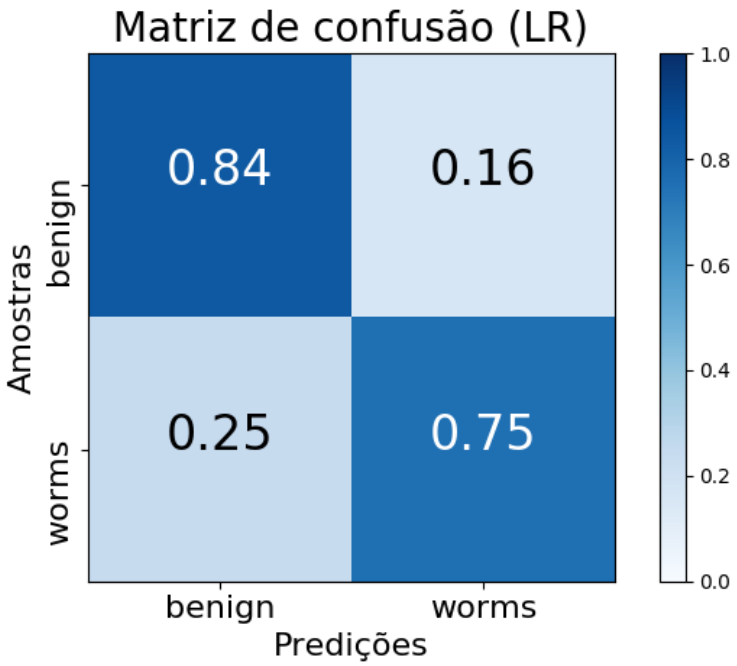


Figura 5.16: Matriz de Confusão do LR.

O único algoritmo de rede neural usado neste projeto, o MLP, tem sua matriz de confusão representada na figura 5.17, classificou 91% das amostras benignas sem erro e 85% para amostras de *worms*.

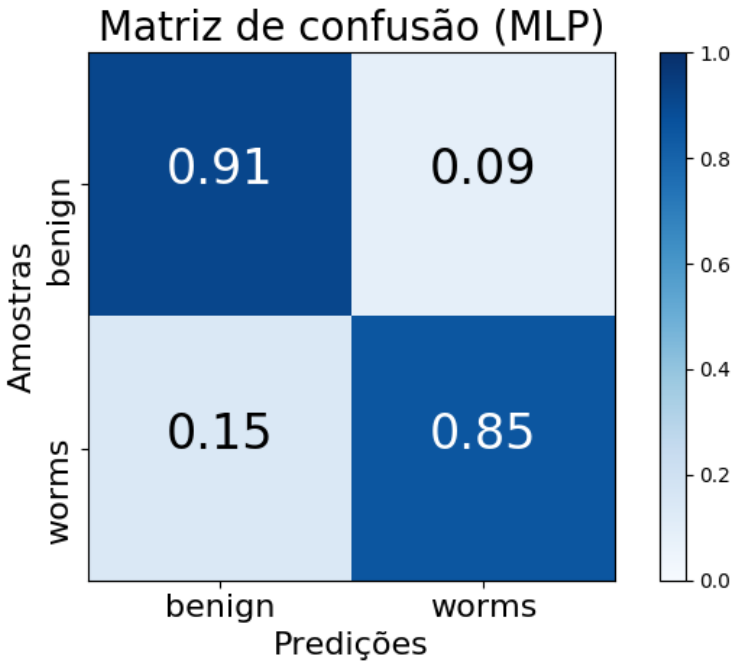


Figura 5.17: Matriz de Confusão do MLP.

Analisando a figura 5.18 pode-se ver que a taxa de verdadeiros positivos para a classe benigna

é de 95%, o que poderia indicar um bom algoritmo de ML. No entanto, as amostras *worms* são classificadas 62% das vezes como pertencentes à classe benigna e apenas 38% destas amostras são corretamente classificadas, ou seja, o Naive Bayes erra suas predições mais que acerta quando o programa é, de fato, um *worm*.

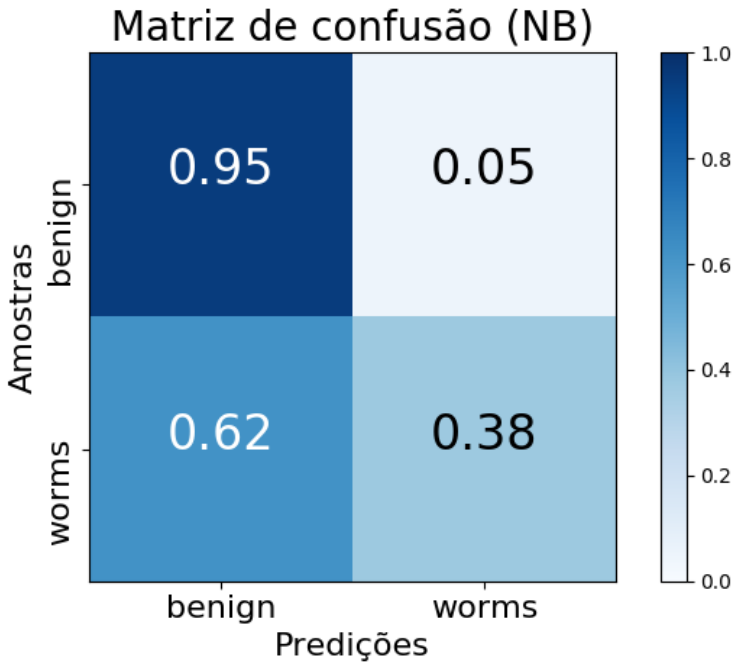


Figura 5.18: Matriz de Confusão do NB.

Por último, a figura 5.19 que mostra a matriz de confusão do algoritmo SVM, nela é visível a classificação desbalanceada do SVM pois este classifica mais amostras como *worms*. Consequentemente, 61% de suas predições para amostras benignas estão corretas, e para amostras *worms* tem-se 96% preditas corretamente.

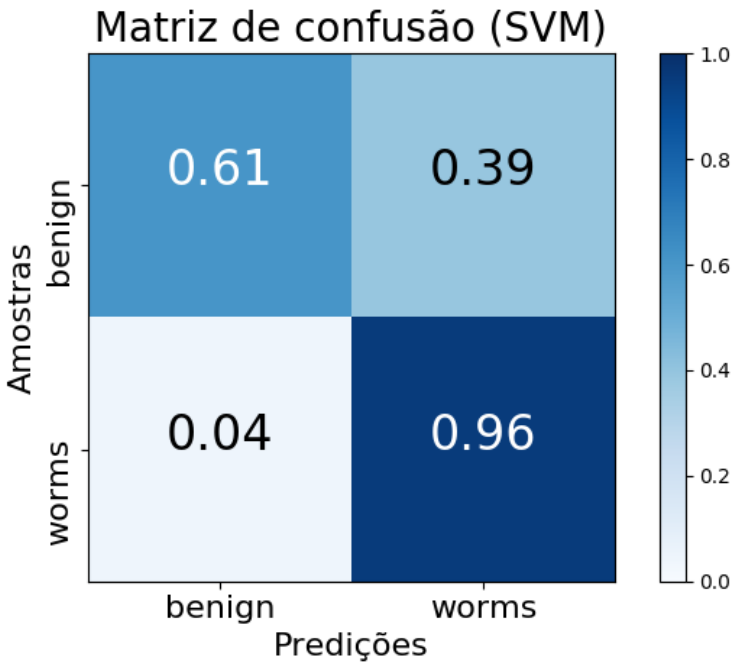
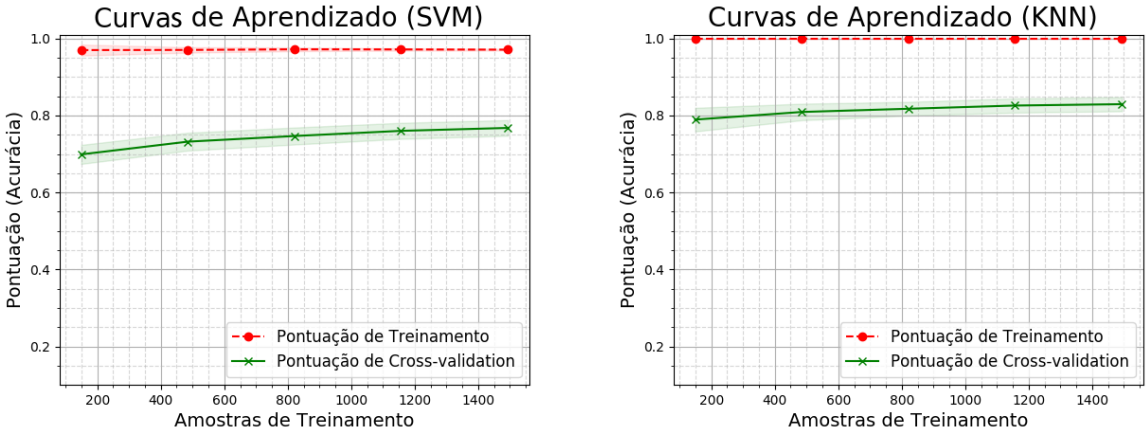


Figura 5.19: Matriz de Confusão do SVM.

Em suma, seis dos sete algoritmos usados para a classificação de *worms* tiveram classificações corretas de amostras *worms* piores que a classificação de programas benignos, isto é, a maioria das amostras testes foram classificadas como benignas sendo elas corretas ou não. Por outro lado, o CART, em geral, obteve resultados melhores que os outros algoritmos em todas as métricas utilizadas para qualificar o desempenho de um algoritmo como a precisão, o *recall*, o *F-Measure* e a acurácia.

5.1.2.2 Curva de Aprendizado



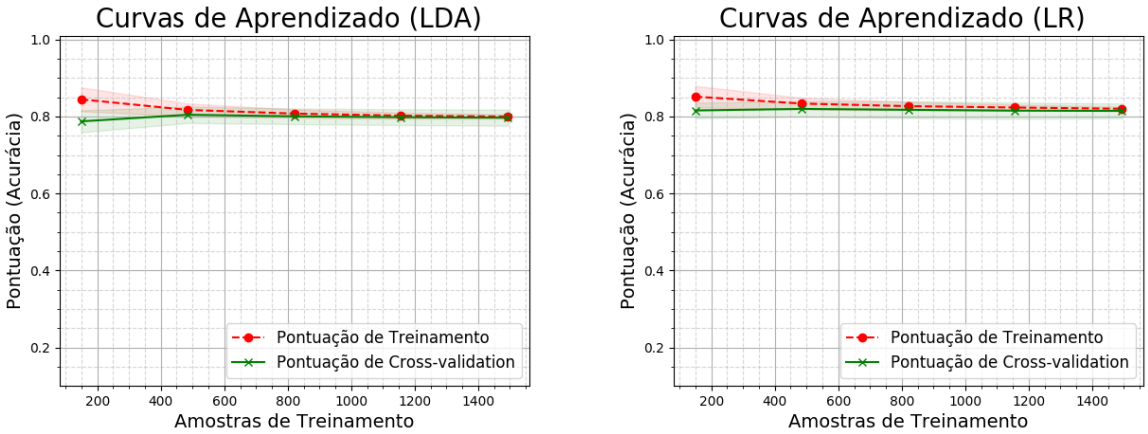
(a) Curva de Aprendizado do LDA.

(b) Curva de Aprendizado do LDA.

Figura 5.20: Modelos com *Overfitting*

Com a curva de aprendizado é possível ver se um determinado modelo está *overfitting* ou *underfitting* ao comparar o conjunto de treinamento com o de *cross-validation*.

Os modelos de aprendizado SVM e KNN, passam pelo fenômeno de *overfitting*, em que as amostras de treinamento possuem uma pontuação muito melhor do que a de validação. Para estes modelos, mostrados na figura 5.20, o aumento na quantidade de amostras acarretaria em maiores pontuações. Para os 2 casos (LR e LDA) mostrados na figura 5.21, a curva de treinamento e



(a) Curva de Aprendizado do LDA.

(b) Curva de Aprendizado do LR.

Figura 5.21: Modelos Lineares

validação convergem, implicando que por mais que ocorra um aumento na quantidade de amostras a tendência é que as pontuações se mantenham as mesmas. Nos casos dos modelos LR e LDA, a aproximação linear não é a melhor alternativa para modelar o *dataset*, nem o aumento das amostras fariam com que a distribuição dos dados se tornasse mais linear.

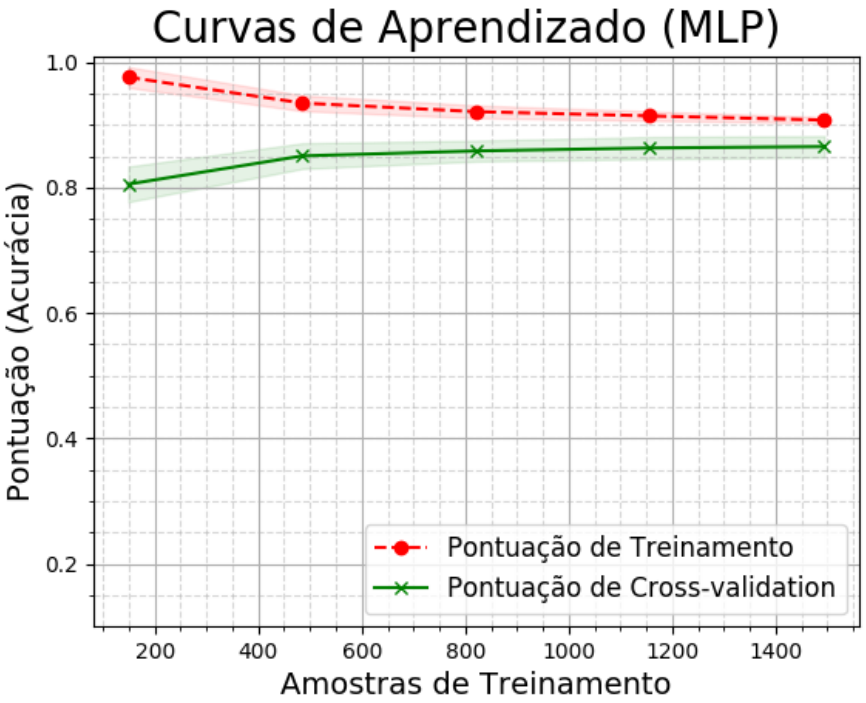


Figura 5.22: Curva de Aprendizado do MLP.

No gráfico da figura 5.22 tem-se a curva de aprendizado plotada para o MLP, nele pode-se notar a convergência das duas curvas para o valor de acurácia de aproximadamente 0.9. Como o MLP é um algoritmo não linear, este se comporta melhor que os lineares supracitados.

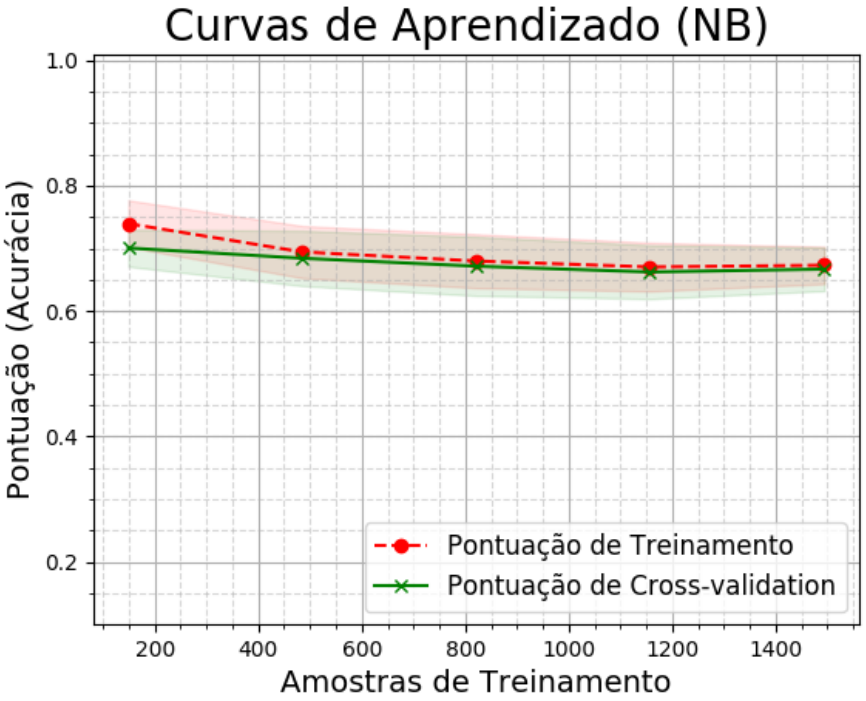


Figura 5.23: Curva de Aprendizado do NB.

A curva de aprendizado do *Naive Bayes* converge com os valores mais baixos do que todos os outros modelos, indicando *underfitting*, ou seja o modelo gerado não se adequa com a distribuição dos atributos. O NB assume que os atributos são independentes entre sí o que é extremamente simples para um *dataset* com esse nível de complexidade, portanto mesmo com o aumento das amostras não há nenhuma melhora significativa nas pontuações.

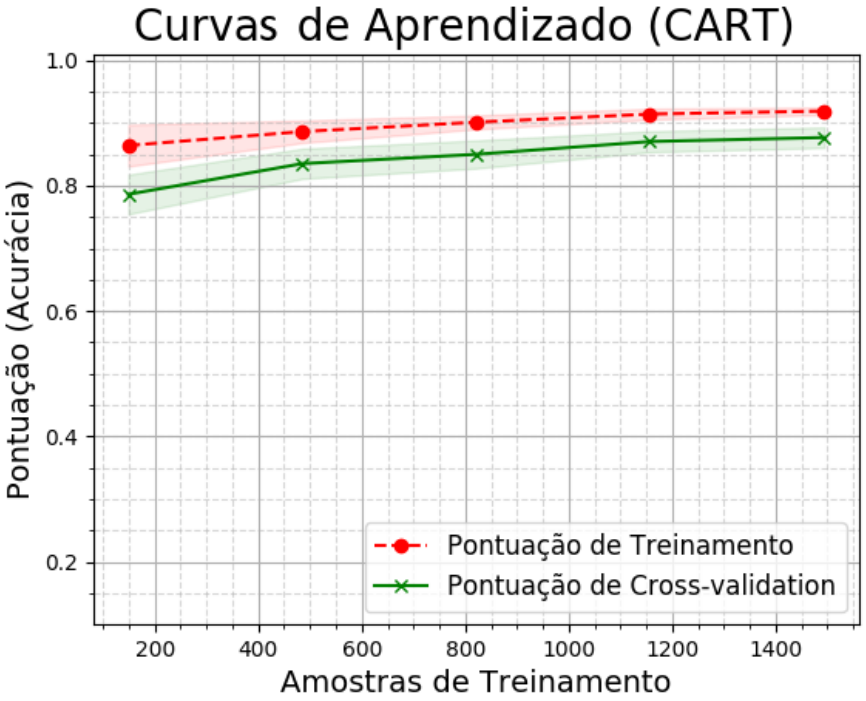


Figura 5.24: Curva de Aprendizado do CART.

Por fim, a curva de aprendizado do CART possui o melhor comportamento, onde tanto a curva de treinamento quanto a curva de aprendizado crescem conforme o número de amostras aumentam.

5.2 Sistema de Detecção

Com o intuito de testar os algoritmos, será usado novos *worms* e programas benignos que não foram submetidos para o aprendizado de máquinas, isto é, não estavam presentes no *dataset* gerado.

5.2.1 Detectando *WannaCry*

Dentre os artefatos a serem testados, foi identificado *WannaCry* que é um *ransomware* recente que atingiu centenas de milhares de computadores pelo mundo desde de 12 de Maio de 2017. Este *malware* é muito mais perigoso que outros tipos de *ransomwares* comuns por causa de sua habilidade de se espalhar pelas redes de organizações ao explorar vulnerabilidades críticas do sistema operacional Windows (SYMANTEC-CORPORATION, 2017a) como o *EternalBlue* que explora o protocolo *Server Message Block* (SMB). Tal vulnerabilidade foi corrigida em Março de 2017. Na figura a seguir, pode-se ver a tela de um computador infectado pelo *ransomware* análisado.

Figura 5.25: Tela de um computador ao ser infectado pelo *WannaCry* (FORCEPOINT-SECURITY-LABS, 2017).

O executável do *WannaCry* foi enviado para a análise automática no VMT que retornou as análises estática, dinâmica, de rede e as decisões de seus sete algoritmos de ML. O resultado dos algoritmos de ML podem ser encontrados na tabela 5.4, nela pode-se ver que cinco dos sete algoritmos retornaram que o *WannaCry* é, de fato, um *worm*. O CART tem 100% de certeza que o arquivo é maligno pois é a fração de amostras da mesma classe que pertence a mesma folha da árvore (COURNAPEAU, 2017) significando que todas as amostras nesta folha são *worms*. Um *dataset* com mais amostras ou um valor de k maior (no projeto usou-se 5) poderia levar a um resultado melhor. Outro algoritmo que obteve resultado anormal foi o NB, que como discutido na seção 5.1.2.1 tende a prever uma amostra como benigna. Infelizmente, não é possível mostrar quais valores do *WannaCry* no *dataset* fez os algoritmos KNN, MLP e NB errarem pois estes são algoritmos *black box*.

Tabela 5.4: Decisão para o WannaCry.

Algoritmo	Chances de ser benigno	Chance de ser *worm*	Decisão
CART	0,00%	100,00%	*Worm*
KNN	78,90%	21,10%	Benigno
LDA	29,38%	70,62%	*Worm*
LR	35,32%	64,68%	*Worm*
MLP	100,00%	0,00%	Benigno
NB	100,00%	0,00%	Benigno
SVM	32,79%	67,21%	*Worm*

Uma vez tendo a suspeita de que executável é malicioso, cabe ao especialista averiguar a procedência do artefato.

5.2.1.1 Análise Dinâmica

Durante sua execução na *sandbox*, pode-se ver o processo de cifragem de diversos arquivos. Na figura 5.26, tem-se arquivos ".zip" sendo modificados com a adição do sufixo ".WNCYR" para mostrar à vítima que seus arquivos foram, de fato, cifrados.

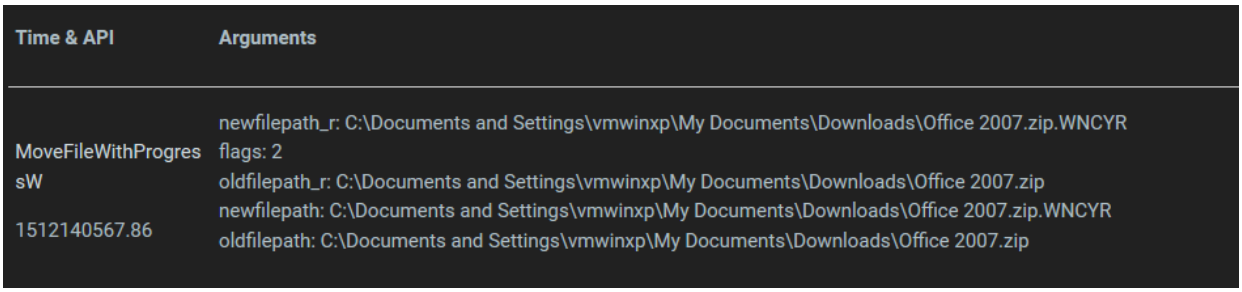


Figura 5.26: Ação tomada pelo *WannaCry*.

5.2.1.2 Análise Estática

Grande parte do *WannaCry* foi cifrada, impossibilitando uma análise estática mais completa do arquivo sem o uso de um decriptador. Seus altos valores de entropia podem ser vistos na figura 5.27.

Name	Virtual Address	Virtual Size	Size of Raw Data	Entropy
.text	0x00001000	0x000069b0	0x00007000	6.4042351061
.rdata	0x00008000	0x00005f70	0x00006000	6.66357096841
.data	0x0000e000	0x00001958	0x00002000	4.45574950787
.rsrc	0x00010000	0x00349fa0	0x0034a000	7.9998679751

Figura 5.27: Entropia do *WannaCry*.

No entanto, das partes que não foram cifradas, ainda é possível fazer a análise das *strings* do arquivo. Como por exemplo, pode-se encontrar os endereços das carteiras de *bitcoin*[1] do *WannaCry*, que podem ser vistos na figura 5.28. O endereço "13AM4VW2dhxYgXeQepoHkHSQuy6NgaEb94" é o mesmo mostrado na figura 5.25.

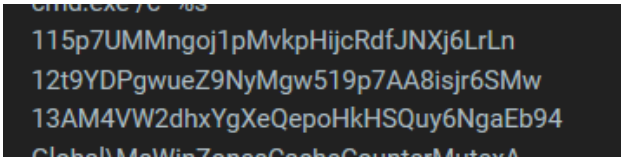


Figura 5.28: *Strings* das carteiras de *bitcoin* do *WannaCry*.

5.2.1.3 Análise de Rede

Como discutido no começo desta seção, o *WannaCry* fez acessos às portas do SMB 137 e 138 pelo protocolo UDP. Isto pode ser um sinal de que a vulnerabilidade *EternalBlue* esteja sendo usada para a propagação do *malware* na rede já que o IP de destino é o de *broadcast* da rede interna 192.168.56.0/24. Na figura a seguir tem-se, em parte, as comunicações UDP do *WannaCry*.

Source		Destination
192.168.56.101:137	→	192.168.56.255:137
192.168.56.101:138	→	192.168.56.255:138

Figura 5.29: Requisições UDP do *WannaCry*

[1] *Bitcoin*: Um tipo de cryptomoeda.

5.2.2 Detectando *notPetya*

Outro *malware* analisado é o *notPetya* ou ExPetr, que começou a se propagar em 27 de Junho de 2017, poucos meses após a ocorrência do *WannaCry*. O alvo principal deste ataque eram empresas e não usuários (PANKOV, 2017), causando danos maiores por indisponibilidades de sistemas como o sistema de monitoramento da usina de *Chernobyl* (MARY ILYUSHINA, 2017). Similar ao *Wannacry*, seu meio de ataque é o a vulnerabilidade *EternalBlue*, no entanto, o *notPetya* também usa técnicas clássicas para se propagar na rede por meio do SMB. Tal método usado para se propagar o classifica como um *worm* além de *ransomware* (SYMANTEC-CORPORATION, 2017b).

A amostra foi submetida ao VMT, retornando resultados das análises estática, dinâmica e de rede, além das predições feitas pelos algoritmos de ML. Estas predições podem ser encontrados na tabela 5.5. É possível notar que em praticamente todos os algoritmos foram identificados padrões que indicam a suspeita de ser um artefato malicioso. O único algoritmo que identificou o artefato como benigno, foi o NB que embora tenha uma precisão, para amostras *worms*, alta, dificilmente identifica o executável como *worm*, ou seja, possui um *recall* baixo. Assim como o *ransomware* anterior, não é possível determinar quais atributos do *malware* foi importante para a classificação errônea do *Naive Bayer*.

Tabela 5.5: Decisão para o *notPetya*

Algoritmo	Chances de ser benigno	Chance de ser *worm*	Decisão
CART	0,00%	100,00%	*Worm*
KNN	15,24%	84,76%	*Worm*
LDA	21,26%	78,74%	*Worm*
LR	26,59%	73,41%	*Worm*
MLP	42,35%	57,65%	*Worm*
NB	100,00%	0,00%	Benigno
SVM	12,12%	87,88%	*Worm*

Uma vez tendo a suspeita de que um dado executável é *worm*, cabe ao especialista averiguar se de fato aquele executável é malicioso.

5.2.2.1 Análise Estática

Assim como o *WannaCry* o *notPetya* também possui os recursos cifrados em sua maioria, embora as mensagens direcionadas ao usuário não possuem evidentemente nenhum tipo de cifragem. A figura 5.30 é uma imagem tirada do VMT da análise estática do artefato, mais especificamente, as strings presentes no arquivo.

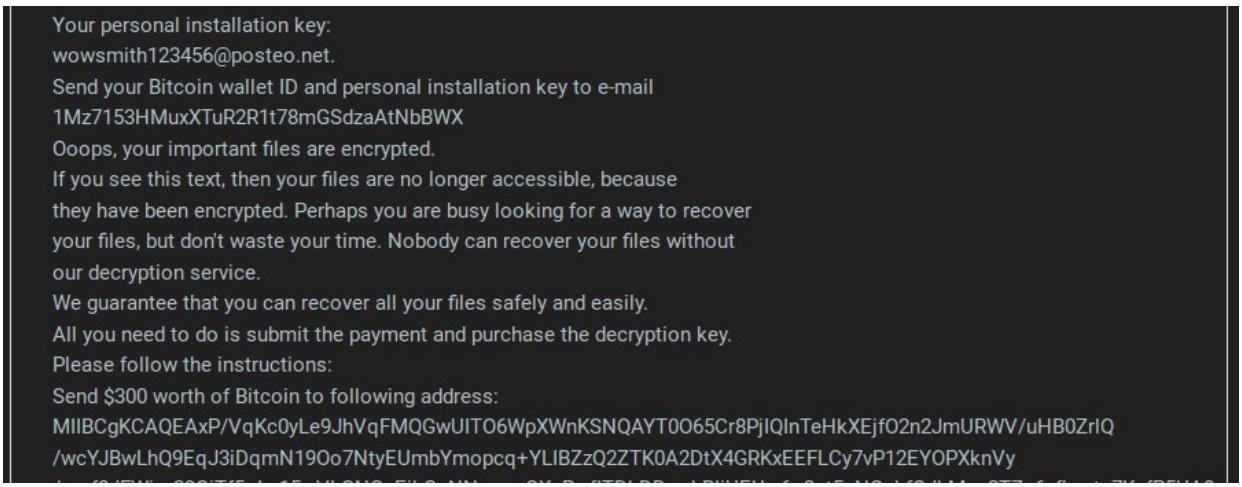


Figura 5.30: *Strings* do artefato.

Nesta figura é possível observar a mensagem utilizada para reportar à vítima que seus dados foram cifrados. Há também, nas *strings*, informações como o preço para obter a chave que decifra os dados roubados e o endereço da carteira *bitcoin* para a transferência monetária.

5.2.2.2 Análise de Rede

Na figura 5.31 é possível notar que assim como no experimento com o *WannaCry* também foram feitas varreduras nas portas 137 e 138, utilizadas para espalhamento na rede.

Source		Destination
192.168.56.101:137	→	192.168.56.255:137
192.168.56.101:138	→	192.168.56.255:138

Figura 5.31: Requisições UDP do *notPetya*.

5.2.3 Submissão de um Arquivo Benigno

Para validar que a sistema pode efetivamente auxiliar um analista na seleção dos artefatos importantes, foi submetido um arquivo benigno, no caso um instalador do Sublime, programa comum para edição de texto. Na tabela 5.6 é possível verificar que a maioria dos modelos de ML obtiveram êxito na predição do instalador do programa *Sublime* como benigno.

Tabela 5.6: Decisão para o Instalador Sublime

Algoritmo	Chances de ser benigno	Chance de ser *worm*	Decisão
CART	38,84%	61,16%	*Worm*
KNN	64,41%	35,59%	Benigno
LDA	67,85%	32,15%	Benigno
LR	71,15%	28,85%	Benigno
MLP	99,40%	0,60%	Benigno
NB	100,00%	0,00%	Benigno
SVM	32,83%	67,17%	*Worm*

A predição erroneamente dada pelo SVM deve-se ao fato de estar *overfitting* e não discerne este tipo de amostra benigna, o CART, por outro lado, embora seja o melhor classificador, não pôde diferenciar o instalador do *Sublime* de um *worm*, mostrando que ainda é necessária a presença de um analista para decidir a procedência de um dado arquivo. Por fim, como a grande maioria dos modelos indicaram o artefato como benigno, o analista poderia inferir que este é um artefato que não necessita ser analisado prioritariamente.

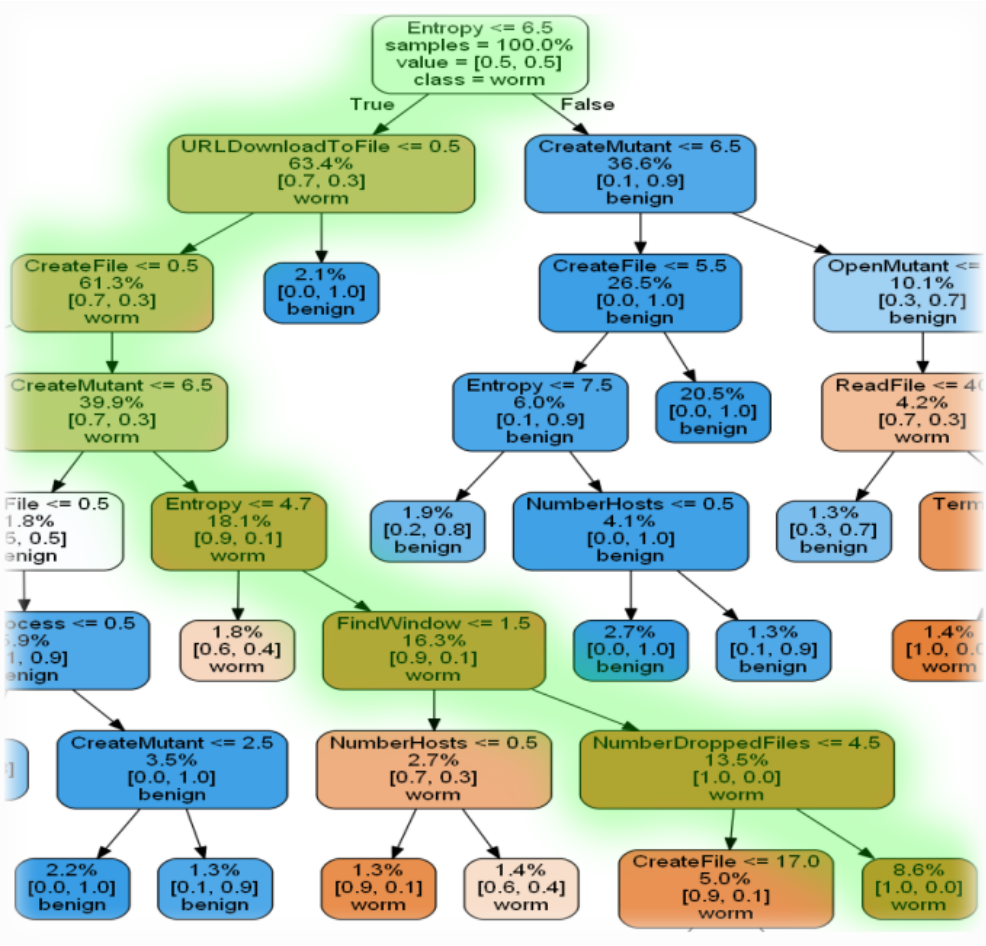


Figura 5.32: Caminho da Decisão do CART para o Sublime Text 2.

Na figura anterior, pode-se ver o caminho tomado pelo algoritmo CART para chegar na decisão de que o Sublime Text 2 é considerado *worm*. Isto é, o editor de texto tem entropia maior que 4,7 e menor que 6,5, a quantidade de chamadas para *URLDownloadToFile* é menor ou igual a 0,5, as quantidades de chamadas para *CreateFile* e *FindWindow* foram maiores que 0,5 e 1,5, respectivamente e, por último, a quantidade de arquivos baixados foi maior que 4,5. Para os demais algoritmos, não é possível fazer este processo intuitivo que demonstra quais atributos foram mais ou menos relevantes para a decisão, ou seja, são algoritmos *black box*.

Para entender o diferencial dessa abordagem, basta imaginar o seguinte senário, um especialista recebe o artefato (e.g *notPetya*), em uma *honeynet*[2] por exemplo, submete no VMT, tem o retorno com as predições positivas para *worm* por grande parte dos algoritmos de aprendizado de máquina. No caso de um *malware* recente que não possui assinaturas em nenhum AV, terá uma grande chance do especialista descobrir as vulnerabilidades usadas e como são feitos os novos ataques e, consequentemente, poderá aumentar a segurança do ambiente antes que ataques como o *WannaCry* e *notPetya* se disseminem.

Pode-se ver que o sistema VMT implementado neste projeto conseguiu identificar dois *ransomwares*, com características de *worms*, com sucesso. Além disto, conseguiu-se medir o desempenho dos sete diferentes algorítimos de aprendizado de máquina usando técnicas de *holdout* e de validação cruzada n-dobras. Contudo, com exceção do CART, não é possível determinar os motivos das predições dos algoritmos por eles serem *black box*. Vale notar que, os algoritmos LDA e LR são *white box* quando a quantidade de atributos utilizadas em um *dataset* é pequena, o que não é o caso deste projeto que utiliza mais de 25 atributos diferentes.

[2]Honeynet: ferramenta que consiste de uma rede projetada especificamente para ser comprometida, e que contém mecanismos de controle para prevenir que seja utilizada como base de ataques contra outras redes (CRISTINE et al., 2007).

Capítulo 6

Conclusão

Neste projeto foi desenvolvido um sistema de detecção de *worms* que pode auxiliar na análise de arquivos desconhecidos por um especialista. Tanto o *Venus Malware Trap* quanto o *Cuckoo Learning* podem ser adaptados e melhoradas para outras soluções, sendo levado em consideração sempre a simplicidade de instalação e o reuso do código implementado. Neste capítulo são tratadas as conclusões obtidas da análise e também as propostas de trabalhos futuros.

Pode-se concluir que mesmo com um *dataset* com menos de 2000 amostras ainda é possível que algorítimos diferentes de aprendizado de máquinas possam fazer a distinção entre um *worm* e um programa benigno com até 87,8% de acurácia, usando apenas 26 atributos diferentes.

Dado o projeto, observou-se que o uso da ferramenta de *sandbox* é de extrema importância na análise de arquivos maliciosos, ajudando ver funções de comportamento que um determinado programa tem e que são ofuscadas em análises de antivírus. Ferramentas como o *VirusTotal* vem recentemente aplicando esta técnica como pode ser visto nas suas postagens recentes no blog oficial da página (VIRUSTOTAL, 2017).

Por fim, verificou-se que a ferramenta conseguiu discernir um *worm* de um programa benigno, por exemplo, os *malwares WannaCry* e *notPetya*, pertencentes à classe *worm*, foram classificados com sucesso enquanto o Sublime foi predito como pertencente a classe benigna.

6.1 Trabalhos Futuros

Em trabalhos futuros espera-se a utilização de outros tipos de aprendizado de máquina. Considerando a maneira em que a aplicação foi desenvolvida, é possível a utilização do aprendizado não supervisionado por exemplo. Além disto, para que se possa executar uma maior variedade de *malwares*, pode-se utilizar máquinas virtuais com sistema operacionais em 64bits e mais recentes como os Windows 7, 8 e 10, que serão uteis para a análise de novos *malwares* específicos para esses SO.

Também é possível nos próximos trabalhos implementar uma arquitetura mais robusta utilizando outras ferramentas de virtualização em um ambiente mais complexo e flexível, como por

exemplo VMware, podendo melhorar o desempenho da aplicação uma vez que com um maior número de máquinas virtuais é possível executar uma maior quantidade de *malwares* simultaneamente.

Outro aspecto que pode ser explorado futuramente é a implementação do sistema VMT voltado para *smartphones* Android, isto é, uma análise automatizada com uso de aprendizado de máquina explorando as APIs do sistema operacional Android em ambiente virtualizado. Existem trabalhos (PEIRAVIAN; ZHU, 2013) que envolvem já a análise do SO Android com o uso de aprendizado de máquina, mas não fazem proveito da análise dinâmica.

Bibliografia

AYCOCK, John. **Computer Viruses and Malware**. [S.l.]: Springer, 2006. ISBN 978-0-387-30236-2.

ELISAN, Cristopher C. Advanced Malware Analysis. In: [s.l.]: McGraw-Hill Education, 2015. ISBN 978-0-07-181975-6.

ANDRADE, C. A. B. d.; MELLO, C. G. d.; DUARTE, J. C. Malware Automatic Analysis. In: 2013 BRICS Congress on Computational Intelligence and 11th Brazilian Congress on Computational Intelligence. [S.l.: s.n.], set. 2013. p. 681–686. DOI: 10.1109/BRICS-CCI-CBIC.2013.119.

AVTEST. **Statistics Malware**. 2017. Disponível em: <https://www.av-test.org/en/statistics/malware/>. Acesso em: 10 jun. 2017.

KASPERSKY-LAB. **Heuristic analysis in Kaspersky Internet Security 2012**. 2013. Disponível em: <https://support.kaspersky.com/6324>. Acesso em: 11 jun. 2017.

SIKORSKI, Michael; HONIG, Andrew. Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software. In: [s.l.]: No Starch Press, 2012. ISBN 978-1-59327-290-6.

VIRUSTOTAL. **Sobre - VirusTotal**. 2017. Disponível em: <https://www.virustotal.com/pt/about/>. Acesso em: 15 jun. 2017.

CERT-BR. **Cartilha de Segurança – Códigos Maliciosos (Malware)**. 2017. Disponível em: <https://cartilha.cert.br/malware/>. Acesso em: 11 jun. 2017.

KNUTSON, Austin; LIU, Menglu; SCHLENKER, Derek. **The Digital Asset of the Future: Bitcoin & Ether**. 2016. Disponível em: <http://www.economist.com/sites/default/files/the_digital_asset_of_the_future__team_byu.pdf>. Acesso em: 1 dez. 2017.

MALWAREBYTES-LABS. **2017 - State of Malware Report**. 2017. Disponível em: <https://www.malwarebytes.com/pdf/white-papers/stateofmalware.pdf>. Acesso em: 15 jun. 2017.

KAREN, Scarfone; PETER, Mell. **Guide to Intrusion Detection and Prevention Systems (IDPS)**. 2007. Disponível em: <http://ws680.nist.gov/publication/get_pdf.cfm?pub_id=50951>. Acesso em: 11 jun. 2017.

TOM M., Mitchell. Machine Learning. In: [s.l.]: McGraw-Hill, 1997. cap.1, p. 1–5. ISBN 0070428077.

MOHSSEN, Mohammed; MUHAMMAD B., Khan; EIHAB B. M., Bashier. **Machine Learning: Algorithms and Applications**. [S.l.]: CRC Press, 2017. ISBN 978-1-498-70538-7.

MOHRI, Mehryar; ROSTAMIZADEH, Afshin; TALWALKAR, Ameet. Foundations of Machine Learning. In: [s.l.]: The MIT Press, 2012. cap.1, p. 7–10. ISBN 978-0-262-01825-8.

ROKACH, Lior; MAIMON, Oded. Data Mining With Decision Trees: Theory and Applications. In: [s.l.]: World Scientific Publishing Company, 2015. cap.1, p. 7–10. ISBN 978-9-814-59007-5.

AURÉLIEN, Géron. **Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems**. [S.l.]: O'Reilly, 2017. ISBN 978-1-491-96229-9.

JAMES, Gareth et al. An Introduction to Statistical Learning with Applications in R. In: 1. ed. [S.l.]: Springer, 2013. p. 130–149. ISBN 978-1461471370.

CONWAY, Drew; JOHN MILES, White. Machine Learning for Hackers: Case Studies and Algorithms to Get You Started. In: 1. ed. [S.l.]: O'Reilly, 2012. p. 178–182. ISBN 978-1449303716.

BISHOP, Christopher M. Pattern Recognition and Machine Learning. In: 1. ed. [S.l.]: Springer, 2011. p. 188. ISBN 978-0387-31073-2.

THEODORIDIS, Sergios; KOUTROUMBAS, Konstantinos. Pattern Recognition. In: [s.l.]: Elsevier Academic Press, 2003. cap.2, p. 44–54. ISBN 0-12-685875-6.

COURNAPEAU, David. **scikit-learn: machine learning in Python — scikit-learn 0.19.1 documentation**. 2017. Disponível em: <`http://scikit-learn.org/stable/index.html`>. Acesso em: 1 nov. 2017.

CUCKOO-FOUNDATION. **What is Cuckoo? — Cuckoo Sandbox v2.0.0 Book**. 2017. Disponível em: <`http://docs.cuckoosandbox.org/en/latest/`>. Acesso em: 15 jun. 2017.

PYTHON-SOFTWARE-FOUNDATION. **Python 2.7.13 Documentation**. 2017. Disponível em: <`https://docs.python.org/2/tutorial/index.html`>. Acesso em: 28 jun. 2017.

DIAKOPOULOS, Nick; CASS, Stephen. **The 2017 Top Programming Languages**. 2016. Disponível em: <`https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages`>. Acesso em: 17 nov. 2017.

MCKINNEY, Wes. **pandas: powerful Python data analysis toolkit**. 2017. Disponível em: <`https://pandas.pydata.org/pandas-docs/stable/`>. Acesso em: 1 nov. 2017.

BICKING, Ian. **Virtualenv**. 2017. Disponível em: <`https://virtualenv.pypa.io/en/stable/`>. Acesso em: 1 nov. 2017.

RONACHER, Armin. **Welcome to Jinja2 — Jinja2 Documentation (2.9)**. 2017. Disponível em: <`http://jinja.pocoo.org/docs/2.9/`>. Acesso em: 2 nov. 2017.

ORACLE. **Virtualbox Manual**. 2017. Disponível em: <`https://www.virtualbox.org/manual/`>. Acesso em: 1 nov. 2017.

GOOGLE. **Documentation - Materialize**. 2017. Disponível em: <`http://materializecss.com/`>. Acesso em: 2 nov. 2017.

ANDRADE, César Augusto Borges de. **Análise Automática de Malwares Utilizando as Técnicas de Sandbox e Aprendizado de Máquina**. 2013. Diss. (Mestrado) – Instituto Militar de Engenharia, Rio de Janeiro.

PICARD, Robert. **Documentation - Flask**. 2017. Disponível em: <`http://exploreflask.com/en/latest/blueprints.html`>. Acesso em: 2 nov. 2017.

BAYER, Michael. **SQLAlchemy Documentation**. 2017. Disponível em: <http://docs.sqlalchemy.org/en/latest/>. Acesso em: 2 nov. 2017.

KUBAT, Miroslav; HOLTE, Robert C.; MATWIN, Stan. Machine Learning for the Detection of Oil Spills in Satellite Radar Images. **Machine Learning**, v. 30, n. 2, p. 195–215, fev. 1998. ISSN 1573-0565. DOI: 10.1023/A:1007452223027. Disponível em: <https://doi.org/10.1023/A:1007452223027>.

FREITAG, Dayne. Machine Learning for Information Extraction in Informal Domains. **Machine Learning**, v. 39, n. 2, p. 169–202, maio 2000. ISSN 1573-0565. DOI: 10.1023/A:1007601113994. Disponível em: <https://doi.org/10.1023/A:1007601113994>.

SYMANTEC-CORPORATION. **What you need to know about the WannaCry Ransomware**. 2017. Disponível em: <https://www.symantec.com/blogs/threat-intelligence/wannacry-ransomware-attack>. Acesso em: 25 nov. 2017.

FORCEPOINT-SECURITY-LABS. **WannaCry Ransomware-Worm Targets Unpatched Systems**. 2017. Disponível em: <https://blogs.forcepoint.com/security-labs/wannacry-ransomware-worm-targets-unpatched-systems>. Acesso em: 2 dez. 2017.

PANKOV, Nikolay. **ExPetr targets serious business**. 2017. Disponível em: <https://www.kaspersky.com/blog/expetr-for-b2b/17343/>. Acesso em: 28 nov. 2017.

MARY ILYUSHINA, Eric Levenson. **Chernobyl monitoring system hit by global cyber attack**. 2017. Disponível em: <http://edition.cnn.com/2017/06/27/europe/chernobyl-cyber-attack/index.html?iid=EL>. Acesso em: 28 nov. 2017.

SYMANTEC-CORPORATION. **Petya ransomware outbreak: Here's what you need to know | Symantec Blogs**. 2017. Disponível em: <https://www.symantec.com/blogs/threat-intelligence/petya-ransomware-wiper>. Acesso em: 1 dez. 2017.

CRISTINE, Hoepers; KLAUS, Steding-Jessen; MARCELO H. P. C., Chaves. **Honeypots e Honeynets: Definições e Aplicações**. 2007. Disponível em: <https://www.cert.br/docs/whitepapers/honeypots-honeynets/>. Acesso em: 11 jun. 2017.

PEIRAVIAN, N.; ZHU, X. Machine Learning for Android Malware Detection Using Permission and API Calls. In: 2013 IEEE 25th International Conference on Tools with Artificial Intelligence. [S.l.: s.n.], nov. 2013. p. 300–305. DOI: 10.1109/ICTAI.2013.53.

ANEXOS

I. INSTALAÇÃO DAS FERRAMENTAS *OPENSOURCE*

I.1 Pré-requisitos

O *Cuckoo Sandbox*, principal ferramenta utilizada nesse trabalho, por ser desenvolvida completamente em *Python*. É necessário instalar os seguintes pacotes no *Host Cuckoo*:

```
$ sudo apt-get install python python-pip python-dev libffi-dev
$ sudo apt-get install libssl-dev virtualenv python-virtualenv
$ sudo apt-get install python-setuptools libjpeg-dev zlib1g-dev swig
```

A interface Web do *Cuckoo* é baseada em *Django*(Framework Python). É necessária a instalação do banco de dados MongoDB utilizado no caso pelo *framework*.

```
$ sudo apt-get install mongodb
```

Já para o banco de dados utilizado pelo *Cuckoo* se faz necessário a instalação do PostgreSQL

```
$ sudo apt-get install postgresql libpq-dev
```

Para analisar as atividades na rede foi utilizado o tcpdump um *sniffer* padrão da ferramenta.

```
$ sudo apt-get install tcpdump apparmor-utils
$ sudo aa-disable /usr/sbin/tcpdump
```

Obs.: Em alguns sistemas o AppArmor não permite a criação de pacotes PCAP Para que seja possível rodar o *tcpdump* sem (escalar?) de nível de permissão root, pois não gostaríamos de executar o cuckoo como root é necessário o seguinte comando:

```
$ sudo setcap cap_net_raw,cap_net_admin=eip /usr/sbin/tcpdump
```

I.1.1 Criar Usuário

A ferramenta, por padrão, necessita de um usuário próprio para funcionar, portanto é preciso criar um novo usuário no sistema.

```
$ sudo adduser cuckoo
```

Uma vez criado o usuário basta colocar o novo usuário cuckoo no grupo "vboxusers", caso esteja utilizando o VirtualBox, para que o usuário tenha controle sobre as máquinas *guests*, como é mostrado a seguir.

```
$ sudo usermod -a -G vboxusers cuckoo
```

I.2 Instalação do Cuckoo

Primeiramente, é necessário clonar o repositório que será utilizada da versão adaptada do Cuckoo para que este possa ser instalado.

```
$ git clone https://github.com/relisfleuris/cuckoo-learning.git
```

É também fortemente recomendado que se utilize o virtualenv para a instalação do cuckoo. Dentro do repositório clonado do cuckoo faça os seguintes comandos.

```
$ virtualenv venv
$ source venv/bin/activate
(venv)$ pip install -U pip setuptools
(venv)$ pip install -e .
```

Uma vez instalado o cuckoo, é nesserário gerar seu diretório de trabalho (CWD) para isso basta simplesmente escrever a palavra *cuckoo* no terminal, desse modo o software já terá criado o CWD, agora basta configurar de acordo com a documentação do cuckoo (CUCKOO-FOUNDATION, 2017).

I.3 Configurações do Cuckoo

Os arquivos de configurações do Cuckoo são encontrados dentro do diretório CWD dele, mais especificamente, em "/home/cuckoo/.cuckoo/conf/" caso o usuário tenha usado o diretório padrão no usuário cuckoo. Na tabela a seguir pode-se ver onde foram feitas as mudanças dos valores padrões de configurações do Cuckoo.

Tabela I.1: Modificações nas configurações do Cuckoo.

Arquivo	Modulo	Atributo	Antigo valor	Novo valor
cuckoo.conf	timeouts	default	120	300
cuckoo.conf	database	connection	*Vazio*	mysql://user:password@localhost/dbname
processing.conf	virustotal	scan	no	yes
virtualbox.conf	cuckoo1	label	cuckoo1	WINXP SP3 32bit-1

Quando se usa mais de uma máquina virtual, o *Cuckoo* gera uma mensagem de *warning* falando que o banco de dados padrão dele não se comporta corretamente com paralelismo de VMs e que deve-se usar outro banco de dados, por isto usa-se o MySQL, como descrito na tabela I.1, onde

user é o usuário do banco, *password* é sua respectiva senha e *dbname* o nome do banco de dados criado.

Caso o usuário decida usar mais de uma máquina virtual, basta ele adicionar o seguinte trecho de código antes do modulo "honeyd", substituindo o "X" pelo número da máquina virtual (2,3, e assim por diante):

```
[cuckooX]
label = WINXP SP3 32bit-X
platform = windows
ip = 192.168.56.10X
snapshot =
interface =
resultserver_ip =
resultserver_port =
tags =
options =
osprofile =
```

Além disso, deve-se colocar a nova máquina virtual na lista de "machines" no modulo "virtualbox" deste mesmo arquivo como exemplificado no trecho a seguir:

```
machines = cuckoo1, cuckoo2
```

I.4 Configuração de Redes na Máquinas *Host*

Para que a máquina *guest* tenha acesso a internet, é preciso configurar o iptables da máquina *host* com um usuário que tenha acesso ao comando "sudo". Assumindo que a interface com acesso à rede da máquina *host* é "eth0" e que o endereço da rede virtual é 192.168.56.0 com máscara de rede 255.255.255.0. Além do acesso a internet, a máquina *guest* também terá acesso ao *host*. Os comandos a serem usados são os seguintes:

```
$ sudo iptables -t nat -A POSTROUTING -o eth0 -s 192.168.56.0/24 -j
    MASQUERADE

# Drop Padrao.
$ sudo iptables -P FORWARD DROP

# Conexoes existentes.
$ sudo iptables -A FORWARD -m state --state RELATED,ESTABLISHED -j
    ACCEPT

# Aceita conexao da vboxnet para toda a internet.
$ sudo iptables -A FORWARD -s 192.168.56.0/24 -j ACCEPT
```

```
# Trafico interno.
$ sudo iptables -A FORWARD -s 192.168.56.0/24 -d 192.168.56.0/24 -j
    ACCEPT

# Cria Logs de tudo que chegou a este ponto.
$ sudo iptables -A FORWARD -j LOG
```

No entanto, estas regras não iram fazer qualquer forma de encaminhamento de pacotes a não ser que o encaminhamento IP esteja explicitamente permitido. Para fazer isto tem-se um comando que deve ser executado toda vez que o sistema operacional é inicializado (CUCKOO-FOUNDATION, 2017). Os comandos são os dois a seguir e devem ser rodados com usuário que tenha permissão para usar "sudo":

```
$ echo 1 | sudo tee -a /proc/sys/net/ipv4/ip_forward
$ sudo sysctl -w net.ipv4.ip_forward=1
```

Existem outras configurações de redes para o Cuckoo que possam ser uteis, como o roteamento que da *drop* em todos os pacotes que não são para o *host* e o roteamento por redes Tor. Para mais informações destes roteamentos, recomenda-se o acesso à página de documentação do Cuckoo.

I.5 Criação do Adaptador vboxnet0 no VirtualBox

Na seção anterior pode-se ver que há uma sub-rede chamada de vboxnet0 no VirtualBox configurada no modo *Host-Only* como recomendado na documentação do próprio Cuckoo. Para a criação desta rede é necessário que o administrador do sistema vá nas configurações de preferencias do VirtualBox e na aba Rede adicione uma nova rede exclusiva de hospedeiro ao clicar no pequeno ícone à direita que se assemelha a uma placa de rede com um sinal de mais por cima. Na imagem a seguir tem-se este menu de preferências com o botão a ser clicado circulado no canto superior direito.

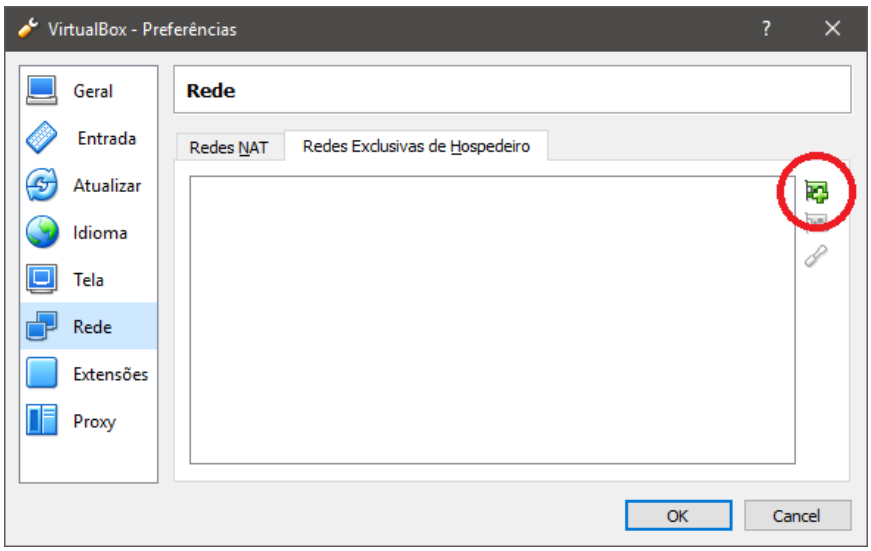


Figura I.1: Menu de Preferências do VirtualBox.

Com a rede criada, basta então clicar duas vezes sobre a rede criada neste menu. Isto irá acionar uma nova janela como mostra na figura I.2. O Endereço IPv4 deve ser 192.168.56.1 e a Máscara de Rede IPv4 255.255.255.0. Ao final dessas configurações, o administrador do sistema pode então clicar nos botões "OK" para finalizar com sucesso a configuração da rede vboxnet0 no *host*.

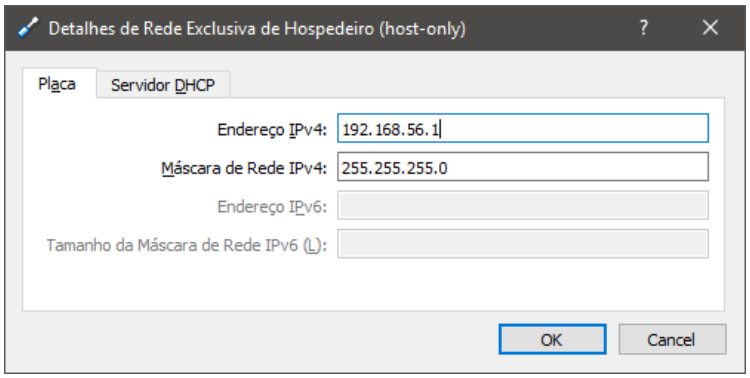


Figura I.2: Configurações de IP e máscara de rede.

Caso outro programa de virtualização de máquinas seja usado, deve-se visitar a documentação do Cuckoo para mais informações de como configurá-los.

I.6 Configurações na Máquina *Guest*

As máquina *guest* deve ter o IP estático configurado na rede 192.168.56.0/24, o Cuckoo recomenda que os IPs de máquinas virtuais usadas para análises comecem a partir do endereço 192.168.56.101. Além disto, deve-se configurar no programa utilizado para virtualização de máquinas (no caso o VirtualBox) a rede do tipo *host-only* de nome vboxnet0.

Além desta configuração de rede, deve-se colocar o *agent* Cuckoo citado na seção 3.2.5 no *startup* da máquinas virtual para que ela possa receber os comandos do *host* Cuckoo. Para isto, basta colocar o arquivo agent.py na pasta de inicialização do Windows XP (CUCKOO-FOUNDATION, 2017).

Por fim, deve-se fazer um *snapshot*, que preserva o estado atual da máquina virtual, pois o Cuckoo irá usá-lo toda vez que uma nova análise é iniciada.

I.7 Instalação do Venus Malware Trap

Primeiramente é necessário clonar o repositório que será utilizado do VMT.

```
$ git clone https://gitlab.com/relisfleuris/venus_malwares_trap.git
```

É também fortemente recomendado que se utilize o virtualenv para a instalação do VMT. Semelhante ao Cuckoo, no repositório clonado do VMT faça os seguintes comandos.

```
$ virtualenv venv
$ source venv/bin/activate
(venv)$ pip install -U pip setuptools
(venv)$ pip install -e .
```

Apos a execução desses comandos basta escrever a palavra *venus* no terminal e o sistema web estará pronto para ser acessado. Por padrão o sistema utiliza a porta 8080.

II. NOVAS FUNCIONALIDADES NO *CUCKOO*

Neste anexo serão abordados as novas funcionalidades feitas no *Cuckoo* e anexado o código das principais modificações, embora o código completo de toda a aplicação se encontra no `https://gitlab.com`.

II.1 Criando um Novo Módulo de *Learning*

Como tratado no capítulo 4 a etapa de *Learning* segue o mesmo padrão das etapas de *Reporting* e *Processing*. Basicamente é necessário alterar 3 partes do sistema para acrescentar um novo módulo.

- Alterando o arquivo de configuração :

  Primeiramente é necessário alterar o arquivo de configuração *learning.conf* (que se encontra no CWD/conf/learning.conf) :

```
[cart]
enabled = yes

[NOVO-MODULO]
enabled = yes
```

  É possível criar outros parâmetros além de "*enabled*", de acordo com a necessidade de configurar o novo modelo de ML.

- Alterando "config.py": É necessário alterar o arquivo "config.py" ,localizado no repositorio do *Cuckoo* (cuckoo/common/config.py), para que a aplicação reconheça as novas configurações:

```
781         },
782         "learning":{
783             "learning":{
784                 "dataset_name": String("worms.data"),
785                 "parameters": List(String, "CreateFile,CreateMutant,Create
786             },
787             "cart":{
788                 "enabled": Boolean(False),
789             },
790             "knn":{
791                 "enabled": Boolean(False),
792             },
793             "NOVO-MODULO":{
794                 "enabled": Boolean(False),
795             },
```

Figura II.1: Acrescentando um Novo Módulo *Learning*.

Com o novo módulo acrescentado no arquivo é possível de fato criar o código que gerará o novo modelo de aprendizado de máquina.

- Novo módulo: Agora que as configurações estão prontas é possível implementar um novo modelo para predição de *malware*, o código a seguir é um exemplo de módulo de *Learning* criado neste trabalho.

```
from sklearn.tree import DecisionTreeClassifier
from cuckoo.common.abstracts import Learn

class CART(Learn):

  def run(self, results):
    self.preparate_dataset()
    cart = DecisionTreeClassifier(
      min_samples_split=50,
      class_weight='balanced'
    )
    cart.fit(self.X, self.Y)
    data = self.get_data(results)
    if data is None:
      return
    prediction = cart.predict(data)
    score = cart.predict_proba(data)
    self.set_predict(results,prediction[0])
    self.set_score(results, score[0])
  def get_data(self, results):
    #first get all apis used in this artfact (20)
    api_list = self.parameters[:20]
    if 'behavior' in results:
      apistats = results["behavior"]["apistats"]
    else:
      return None
    report = []
    count = 0
    for a in api_list:
      for process, values in apistats.iteritems():
        for api, freq in values.iteritems():
          if a in api:
            count +=freq
      report.append(count)
      count = 0
    behavior = results["behavior"]
```

```
    try:
      dropped = results["dropped"]
    except:
      dropped = []
    report.append(len(behavior))
    report.append(len(dropped))
    try:
      hosts = results["network"]["hosts"]
    except:
      hosts = []
    try:
      avgentropy = round(self.entropy(
      results["static"]["pe_sections"]), 4)
    except:
      avgentropy = 5 #standard avg entropy
    report.append(len(hosts))
    report.append(avgentropy)
    answer = [] #needs a 2D array, to predict something
    print report
    answer.append(report)
    print answer
    return answer
```

II.2 Código de Geração do Dataset

O código utilizado na etapa de *Reporting* para gerar o dataset, descrito a seguir.

```
import os
import json
import codecs

from cuckoo.common.abstracts import Report
from cuckoo.common.exceptions import CuckooReportError
from cuckoo.misc import cwd, version, decide_cwd

class Dataset(Report):
  """Saves analysis results in CSV format."""

  def run(self, results):
    """Writes report.
    @param results: Cuckoo results dict.
    @raise CuckooReportError: if fails to write report.
    """
    api_list = ["CreateFile","CreateMutant", "CreateProcess", "CreateRemoteThread",
     "CreateService", "DeleteFile", "FindWindow", "OpenMutant",
     "OpenSCManager", "ReadFile", "ReadProcessMemory", "RegDeleteKey",
     "RegEnumKey", "RegEnumValue", "RegOpenKey", "ShellExecute",
     "TerminateProcess", "URLDownloadToFile", "WriteFile", "WriteProcessMemory"]
    api_results = []
    result = 0
    try:
      apistats = results["behavior"]["apistats"]
      classification = results["info"]["custom"]
      duration = results["info"]["duration"]
      pe_id = results["info"]["id"]
```

```
      for a in api_list:
        for process, values in apistats.iteritems():
          for api, freq in values.iteritems():
            if a in api:
              result +=freq
              #print(api, freq)
        api_results.append(result)
        result = 0
      behavior = results["behavior"]
      try:
        dropped = results["dropped"]
      except:
        #vazio = length 0
        dropped = []
      try:
        hosts = results["network"]["hosts"]
      except:
        #vazio = length 0
        hosts = []
      csv_results = "".join(str(res)+',' for res in api_results)
      avgentropy = round(self.entropy(results["static"]["pe_sections"]), 4)
      csv_results += str(len(behavior)) + "," + str(len(dropped)) + "," + str(len(hosts)) + "," +
    str(avgentropy) + "," \
     + classification+","+results["target"]["file"]["name"]
      if not os.path.exists(self.learning_path):
        os.mkdir(self.learning_path)
      report = codecs.open(os.path.join(self.learning_path, "worms.data"), "a", "utf-8")
      report.write(csv_results + "\n")
      report.close()
    except (UnicodeError, TypeError, IOError) as e:
      raise CuckooReportError("Failed to generate CSV file: %s" % e)

  def entropy(self, pe_sections):
    sectionslen = len(str(pe_sections))
    entropies = []
    totalsize = sum(int(section["size_of_data"], 16) for section in pe_sections)
    return sum(section["entropy"]*int(section["size_of_data"],16)/totalsize for section in
     pe_sections)
```