Capítulo 3

Aprendizado de Máquina em Plataformas de Processamento Distribuído de Fluxo: Análise e Detecção de Ameaças em Tempo Real

Martin Andreoni Lopez (UFRJ), Igor Jochem Sanz (UFRJ),
Antonio G. P. Lobato (UFRJ), Diogo M. F. Mattos (UFF) e
Otto Carlos M. B. Duarte (UFRJ)

Abstract

In this chapter, we focus on stream processing architectures for real-time threat detection in networks. We present and compare Apache Storm, Apache Spark Streaming, and Apache Flink. The most used feature selection techniques and machine learning algorithms for threat detection are described. We present the common datasets targeted for evaluating the performance of threat detection architectures. Finally, we evaluate proposals based on the lambda architecture, with previous or adaptive training, and zero-day threat detection. The CATRACA real-time threat detection tool, which uses the Apache Spark Streaming platform, is presented.

Resumo

Este capítulo foca em arquiteturas para processamento distribuído de fluxo para detecção de ameaças em redes em tempo real. O capítulo apresenta e compara as plataformas de código aberto para processamento distribuído de fluxo Apache Storm, Apache Spark Streaming e Apache Flink. As principais técnicas de seleção de caraterísticas e algoritmos de aprendizado de máquina usados para classificar os tráfegos em benigno ou malicioso são descritos. Os conjuntos de dados de tráfego usados para a avaliação de desempenho das arquiteturas de detecção de ameaças são mostrados. Por fim, o capítulo avalia propostas baseadas na arquitetura lambda, com treinamento prévio e adaptativo e com detecção de ameaças inéditas (zero-day threats). A ferramenta CATRACA de detecção em tempo real de ameaças usando a plataforma Apache Spark Streaming é apresentada.

Este capítulo foi realizado com recursos do CNPq, CAPES, FAPERJ e FAPESP (2015/24514-9, 2015/24485-9 e 2014/50937-1).

3.1. Introdução

Os ataques a redes de computadores estão cada vez mais comuns e com volumes cada vez maiores. A exploração de novas vulnerabilidades (*zero-day attack*), ataques distribuídos de negação de serviço e uso de *softwares* maliciosos sofisticados têm aumentado. O impacto dos ataques de negação de serviço distribuídos (*Distributed Denial of Service* - DDoS) é cada vez maior, chegando a taxas de ataques da ordem de 1 Tb/s [Chandrasekar et al. 2017]. A popularização dos ataques é tão grande que é possível comprar uma hora de ataque de negação de serviço distribuído por U$S 10 a hora na *deep web* [ARMOR 2018]. Os ataques deixaram de ser apenas motivados por fins econômicos e passaram a ter motivações políticas [Chandrasekar et al. 2017], a exemplo de ataques que visam manipular resultados de eleições, como a suspeita de manipulação russa sobre as eleições dos Estados Unidos em 2016 [ICA 2017].

Detectar ameaças em tempo real e reagir prontamente a ataques são fundamentais para diminuir os impactos de segurança [Andreoni Lopez et al. 2016a]. Atualmente, a detecção de ameaças de segurança pode levar semanas ou meses e espera-se reduzir esse tempo para minutos ou segundos. O cenário para o futuro é ainda mais adverso devido à introdução dos mais de 80 bilhões de dispositivos conectados, até 2025, na Internet das Coisas (IoT - *Internet of Things*) [Gluhak et al. 2011, Mattos et al. 2018]. Esses dispositivos produzem uma grande massa de dados (*Big Data*), que precisa ser gerenciada, processada, transferida e armazenada de forma segura e em tempo real. No entanto, as tecnologias atuais não foram projetadas para essa demanda [Cárdenas et al. 2013].

Os atuais sistemas de segurança, como os sistemas de Gerenciamento e Correlação de Eventos de Segurança (*Security Information and Event Management* - SIEM), não possuem um desempenho satisfatório, pois cerca de 85% das intrusões de redes são detectadas semanas depois de terem ocorrido [Clay 2015]. É essencial que o tempo de detecção seja o mínimo possível para que a prevenção da intrusão possa ser efetiva. A simples análise e a filtragem de pacotes pelos cabeçalhos IP e TCP não são eficientes, já que os atacantes procuram ocultar-se das ferramentas de segurança falsificando o IP de origem e alterando dinamicamente a porta TCP. Além disso, a tecnologia de virtualização, que é empregada em sistemas de computação em nuvem, introduz novas ameaças, gerando novos desafios para sistemas de segurança voltados para a proteção das nuvens [Sanz et al. 2017]. Nesse complexo cenário, uma alternativa promissora para classificar tráfego e detectar ameaças de forma automática baseia-se em técnicas de aprendizado de máquina.

O aprendizado de máquina fornece aos sistemas a capacidade de aprender e melhorar a partir da experiência, sem que sejam explicitamente programados. Baseados em técnicas de aprendizado de máquina, os sistemas são capazes de detectar e reagir automaticamente a ameaças [Lobato et al. 2017, Henke et al. 2011]. As técnicas de aprendizado de máquina se beneficiam da quantidade elevada de amostras para treinar, pois os métodos tendem a ter maior acurácia [Mayhew et al. 2015]. No entanto, para processar grandes quantidades de dados, os métodos tradicionais de aprendizado de máquina apresentam alta latência devido ao alto consumo de recursos computacionais, o que impede respostas em tempo real. A detecção de ameaças é um problema de processamento de uma quantidade massiva de dados (*Big Data*) e, portanto, sistemas que se baseiam no processamento centralizado para o treinamento dos modelos são inviáveis de serem aplicados.

A velocidade e a variabilidade dos dados da Internet são outros elementos complicadores para a detecção de ameaças. A identificação de ameaças em redes deve ser realizada em tempo real e, se possível, enquanto o fluxo da ameaça esteja ativo, para que contramedidas eficazes possam ser aplicadas, diminuindo assim os impactos econômicos e de privacidade. Além disso, devido aos diferentes perfis de uso da rede, as características do tráfego podem variar significativamente. Logo, é importante que as técnicas de aprendizado de máquina usadas sejam capazes de reagir às mudanças no tráfego, através da adaptação dos modelos de aprendizado ou através do retreinamento dos modelos, quando identificada uma mudança no padrão do tráfego [Lobato et al. 2018, Polikar et al. 2001]. Portanto, há a necessidade de se adotar novas ferramentas e plataformas para realizar o aprendizado de máquina sobre a grandes massas de dados coletados na rede, a uma rápida velocidade e levando em conta a grande variabilidade do tráfego da Internet.

A técnica de processamento distribuído de dados em fluxos realizada em aglomerados computacionais (*clusters*) é uma alternativa para processar com rapidez o grande volume de tráfego de redes que é potencialmente ilimitado [Gaber et al. 2005, Stonebraker et al. 2005]. Para enfrentar esse desafio, plataformas de processamento distribuído de fluxo de dados, de código aberto, como a Apache Storm [Zhao et al. 2015], a Apache Flink [Carbone et al. 2015a] e a Apache Spark Streaming [Franklin 2013] vêm sendo propostas. Essas plataformas diferem entre si no modelo de distribuição dos dados, na resiliência a falhas e no desempenho que alcançam ao serem submetidas a uma dada carga de trabalho [Andreoni Lopez et al. 2016c].

Este capítulo apresenta uma avaliação comparativa das três principais plataformas de processamento de fluxo de código aberto: a Apache Storm, a Apache Flink e a Apache Spark Streaming. O foco do capítulo são as arquiteturas de processamento distribuído de fluxo nas plataformas de código aberto mencionadas para a detecção de ameaças em tempo real. São apresentados os conjuntos de dados usados para avaliar o desempenho de algumas propostas apresentadas. Conceitos e técnicas usadas para redução de dimensionalidade e seleção de características são descritas. As principais métricas usadas na avaliação de desempenho de sistemas de aprendizado de máquina em detecção de intrusão são apresentadas. Arquiteturas de processamento distribuído de fluxos que seguem a arquitetura lambda, adaptativas e de processamento incremental são descritas e resultados são mostrados. Direções de pesquisas futuras são apresentadas. Por fim, a ferramenta CATRACA [Andreoni Lopez et al. 2017b] que realiza a detecção de anomalias em fluxos de rede é apresentada como uma parte prática de detecção de anomalias em fluxos de rede usando a plataforma Spark Streaming.

3.2. Plataformas de Processamento Distribuído de Fluxo

O processamento de fluxo (*stream processing*) possibilita extrair valores de dados em movimento, de forma similar à qual o processamento em lotes (*batch processing*) faz para dados estáticos. O objetivo do processamento de fluxo é possibilitar tomadas de decisões em tempo real, ou tempo quase real, ao prover capacidade de inspecionar, correlacionar e analisar os fluxos de dados à medida que os dados atravessam o sistema de processamento. Exemplos de cenários que requerem processamento de fluxo são: aplicações de monitoramento de tráfego para segurança de redes de computadores; aplicações de redes sociais, como o Twitter ou o Facebook; aplicações de análise financeira que mo-

nitoram fluxos de dados de ações reportadas em bolsas de valores; detecção de fraudes de cartão de crédito; controle de estoque; aplicações militares que monitoram leituras de sensores usados por soldados, como pressão arterial, frequência cardíaca e posição; processos de manufatura; gerenciamento energético; entre outras. Muitos cenários requerem capacidades de processamento de milhões ou centenas de milhões de eventos por segundo.

Os sistemas de gerenciamento de base de dados (*Database Management System* - DBMS) convencionais armazenam e indexam registros de dados antes de disponibilizá-los para a atividade de consulta, o que os torna inadequado para aplicações em tempo real ou respostas na ordem dos sub-segundos [Abadi et al. 2005]. As bases de dados estáticas não foram projetadas para o carregamento rápido e contínuo de dados. Logo, não suportam diretamente o processamento contínuo que é típico dos aplicativos de fluxo de dados. Além disso, se o processo não for estritamente estacionário como a maioria das aplicações do mundo real não o são, a saída poderia gradualmente mudar ao longo do tempo. Ameaças à segurança em redes TCP/IP, foco deste capítulo, são um típico exemplo de dados em movimento, no qual a saída muda ao longo do tempo.

Os fluxos de dados diferem do modelo convencional de dados armazenados [Gama and Rodrigues 2007] em: i) os elementos de dados no fluxo chegam em linha (*online*); ii) o sistema não tem controle sobre a ordem em que os elementos de dados chegam para serem processados; iii) os fluxos de dados são potencialmente ilimitados em tamanho; iv) uma vez que um elemento de um fluxo de dados foi processado, ele é descartado ou arquivado e não pode ser recuperado facilmente, a menos que seja armazenado explicitamente na memória, que normalmente é pequena em relação ao tamanho dos fluxos de dados. No processamento de fluxo, deve ser feito a computação antes de armazenar o dado, esquema conhecido como *compute-first*, *store-second*. A Tabela 3.1 resume as principais diferenças entre o processamento em lotes de dados estáticos e o processamento de fluxo de dados em movimento.

Tabela 3.1. Resumo de comparação de características entre o processamento em lotes e o processamento de fluxo.

	Em Lotes	**De Fluxo**
Núm. vezes que pode processar dado	Múltiplas vezes	Uma única vez
Tempo de processamento	Ilimitado	Restrito
Uso de memória	Ilimitado	Restrito
Tipo de resultado	Acurado	Aproximado
Topologia de processamento	Centraliz. / Distrib.	Distribuída
Tolerância a falhas	Alta	Moderada

O processamento de fluxo de dados é modelado através de um grafo acíclico dirigido (GAD) com nós fontes de dados, que continuamente emitem amostras, e nós de processamento interconectados. Um fluxo de dados ϕ é um conjunto infinito de dados, em que $\phi = \{D_t \mid t \geq 0\}$ e o ponto D_t é um dado representado por um conjunto de atributos e uma estampa de tempo. Formalmente, um ponto é descrito por $D_t = (\mathbf{V}, \tau_t)$, em que $\mathbf{V}$ é uma n-tupla na qual cada valor corresponde a um atributo e τ_t é a estampa de tempo para o t-ésimo dado. Os nós fontes emitem tuplas ou mensagens que são recebidas por

nós que realizam processamento, chamados de elementos de processamento (EP). Cada EP recebe dados em suas filas de entrada, executa algum processamento sobre os dados e produz uma saída para suas filas de saída.

Uma série de requisitos devem ser atendidos em plataformas de processamento distribuído de fluxo, Stonebraker *et al.* realçam os mais importantes [Stonebraker et al. 2005]. A capacidade de processar o dado em linha, sem a necessidade de armazená-lo para efetuar as operações, é fundamental para manter a baixa latência, uma vez que operações de armazenamento, como escrita e leitura em disco, acrescentam atrasos inaceitáveis no processamento. Além disso, é ideal que o sistema seja ativo, ou seja, que possua políticas próprias para operar sobre os dados sem depender de instruções externas. Devido ao grande volume, os dados devem ser separados em partições para tratá-los em paralelo. A alta disponibilidade e a recuperação de falhas também são críticas em sistemas de processamento de fluxo. Em aplicações de baixa latência, a recuperação deve ser rápida e eficiente, proporcionando garantias de processamento. Assim, as plataformas de processamento de fluxo devem fornecer mecanismos de resiliência contra imperfeições ou falhas, como atrasos, perda de dados ou amostras fora de ordem, que são comuns no processamento distribuído de fluxo em aglomerados computacionais (*clusters*). Além disso, os sistemas de processamento devem ter um mecanismo de execução altamente otimizado para fornecer resposta em tempo real para aplicações com altas taxas de dados. Portanto, a capacidade de processar milhões de mensagens por segundo com baixa latência, dentro de microssegundos, é essencial. Para alcançar esse desempenho, as plataformas devem minimizar a sobrecarga de comunicação entre os processos distribuídos.

As plataformas de processamento de fluxo vêm sendo pesquisadas desde os anos 90, apresentando uma evolução em três gerações. As plataformas de primeira geração foram baseadas nos sistemas de banco de dados que avaliam as regras expressas como pares condição-ação quando novos eventos chegam. Esses sistemas eram limitados em funcionalidades e também não eram dimensionados para fluxos com grandes volumes de dados. Exemplos de sistemas dessa geração incluem Starburst [Widom 1992], Postgres [Stonebraker and Kemnitz 1991] e NiagaraCQ [Chen et al. 2000]. A empresa Apama[2], fundada em 1999, foi a primeira empresa de aplicações de análise em tempo real orientada a eventos com foco em tomada de decisões de negócios. A tecnologia provida pela plataforma Apama permitia monitorar eventos, analisá-los e realizar ações em milissegundos.

Os sistemas de segunda geração focam em estender a linguagem de consulta estruturada (*Structured Query Language* - SQL) para processar fluxos, explorando as semelhanças entre um fluxo e uma consulta (*query*). Em maio de 2003, na Universidade de Stanford, foi criado o projeto STanford stREam datA Manager (STREAM) [Arasu et al. 2004]. O projeto STREAM é considerado um dos primeiros sistemas de gerenciamento de fluxo de dados de uso geral (*Data Stream Management System* - DSMS). O projeto STREAM impulsionou a fundação, ainda em 2003, da empresa Coral8[3]. Em 2007, Coral8 lança uma plataforma comercial, baseada em tecnologias da Microsoft, capaz de processar e analisar milhares de requisições SQL por segundo. O projeto Aurora [Carney

[2]A primeira empresa de análise de eventos em tempo real, Apama Stream Analytics, foi vendida em 2005 para a Progress Software Coorporation por 25 milhões de dólares. Acessado em abril de 2018. https://www.softwareag.com/corporate/products/apama_webmethods/analytics/default.html

[3]Vendida para a empresa Aleri em 2009.

et al. 2002] foi lançado em 2002 em uma colaboração com a Universidade Brandeis, a Universidade de Brown e o MIT. A principal desvantagem do Aurora é que o projeto foi concebido como um mecanismo único e centralizado de processamento de fluxo. Uma nova versão distribuída foi lançada em 2003, chamada Aurora*. Uma última versão foi lançada oficialmente com o nome de Borealis [Abadi et al. 2005], com algumas outras melhorias, como a tolerância a falhas. O projeto Medusa [Balazinska et al. 2004] usou a distribuição do Borealis para criar um sistema de processamento de fluxo federado. Borealis e Medusa ficaram obsoletos em 2008. Os projetos Aurora/Borealis impulsionaram em 2003 a fundação da empresa StreamBase System[4], que lançou comercialmente a plataforma StreamBase para processamento de eventos complexos (*Complex Event Processing* - CEP) com propósito de tomadas de decisões. A universidade de Berkeley, em 2003, cria uma linguagem para executar continuamente consultas SQL com base no sistema de banco de dados Postgres chamada TelegraphCQ [Chandrasekaran et al. 2003]. Com base no TelegraphCQ, a empresa da Truviso[5] foi criada em 2009, e em 2012 a Truviso foi adquirida pela Cisco. Em 2006, a Universidade de Cornell criou Cayuga [Demers et al. 2007], um sistema de publicação-assinatura (*publish/subscribe*) com estados, que desenvolveu uma linguagem simples de consulta para realizar um processamento que escala tanto com a taxa de chegada de eventos quanto com o número de consultas. Cayuga foi substituído por Cougar[6] e é ainda uma pesquisa ativa.

Os sistemas de terceira geração surgiram para atender a necessidade das empresas associadas à Internet de processar grandes volumes de dados produzidos à alta velocidade. O foco principal passa a ser o processamento distribuído escalável de fluxos de dados em aglomerados computacionais. A Google revoluciona o processamento distribuído propondo o modelo de programação MapReduce [Jiang et al. 2010] para o processamento paralelo escalável de grandes volumes de dados em aglomerados. A ideia chave de espalhar-processar-combinar é usada para realizar de forma escalável diferentes tarefas em paralelo em servidores de um aglomerado computacionais. A plataforma Hadoop [Kala Karun and Chitharanjan 2013] é a implementação de código aberto de MapReduce para realizar analíticas em grandes massas de dados. No entanto, devido à alta latência que o MapReduce produz, alguns projetos foram propostos para realizar analíticas de fluxo de dados em tempo real. O projeto Spark substitui o MapReduce do Hadoop para executar em memória operações que o Hadoop executa sobre dados recuperados do disco. As plataformas de código aberto Storm e Flink são propostas para processamento de fluxo. A plataforma Spark propõe uma extensão para processamento de fluxo em microlotes, o Spark Streaming. Em seguida, são descritas as plataformas escaláveis de terceira geração de código aberto Apache Hadoop, Apache Storm, Apache Flink e Apache Spark Streaming. Hadoop é apresentado por fins didáticos.

3.2.1. A Plataforma Apache Hadoop[7]

O Hadoop permite o processamento distribuído de grandes conjuntos de dados através de um aglomerado de computadores usando um modelo simples de programação.

[4]Vendida para a empresa TIBCO Software em 2013.

[5]Truviso Analytic http://jtonedm.com/2009/03/03/first-look-truviso/ acessado em abril de 2018.

[6]Cougar processing http://www.cs.cornell.edu/database/cougar/ acessado em abril de 2018.

[7]Esta subseção foi baseada em [Costa et al. 2012].

O Hadoop é projetado com o objetivo de escalar de um único até milhares de servidores, cada um com processamento e memória local [Costa et al. 2012]. Além da escalabilidade, o Hadoop provê uma robusta tolerância a falhas que é obtida por *software* projetado para detectar e tratar falhas, o que o torna apropriado para aglomerados. O Hadoop é composto por duas partes principais:

- o sistema Hadoop de arquivos distribuídos (*Hadoop Distributed File System* - HDFS), que é um sistema de arquivos para dividir, espalhar, replicar e gerenciar dados ao longo dos nós em um aglomerado;
- o MapReduce, que é um mecanismo computacional para executar aplicações em paralelo. As aplicações são executadas através da divisão em tarefas que manipulam apenas uma parcela dos dados, coletando e redistribuindo resultados intermediários e gerenciando falhas através de todos os nós do aglomerado.

No HDFS um arquivo consiste em blocos com tamanhos iguais e múltiplos dos tamanhos dos blocos de armazenamento. Normalmente os blocos de arquivo são de 64 MB, enquanto os blocos de armazenamento são de 512 kB. O HDFS usa o bloco de arquivos como a unidade a ser empregada para distribuir partes de arquivo entre os discos rígidos dos nós. Como os núcleos de processadores e os discos em um nó e também os nós em um bastidor (*rack*) podem falhar, um bloco de arquivo é armazenado em múltiplos nós do o aglomerado, conforme mostrado na Figura 3.1. O número de cópias pode ser configurado, mas por padrão, é tipicamente igual a três. O sistema de arquivos do Hadoop é classificado como "distribuído" porque ele gerencia o armazenamento para todas as máquinas da rede e os arquivos são distribuídos entre diversos nós. O Hadoop trata todos os nós como nós de dados, o que significa que eles podem armazenar dados. Entretanto, ele elege ao menos um nó para ser o "Name Node". O Name Node decide em qual disco rígido cada uma das cópias de cada um dos blocos de arquivo é armazenada. Além disso, o Name Node mantém todas as informações em tabelas armazenadas localmente em seus discos. Quando um nó falha, o Name Node identifica todos os blocos de arquivo que foram afetados; recupera as cópias desses blocos de arquivo de nós operacionais (sem falha); encontra novos nós para armazenar cópias dos dados afetados; armazena essas cópias no nó escolhido e atualiza a informação em sua tabela. Quando uma aplicação precisa ler um arquivo, ele primeiro se conecta ao Name Node para obter o endereço dos blocos do disco onde os blocos do arquivo estão armazenados. Assim, em seguida, a aplicação pode ler esses blocos diretamente sem outra intervenção do Name Node. Um dos maiores problemas apontados do HDFS é o fato do Name Node poder se tornar um ponto único de falha. Logo, se o nó com o Name Node falhar, todas as informações de mapeamento entre nomes de arquivos e endereços de seus respectivos blocos de arquivo podem ser perdidos. Então, um novo nó precisa ser designado como o Name Node com o mesmo endereço IP do anterior que falhou. Para abordar tal questão, o Hadoop salva cópias das tabelas criadas pelo Name Node em outros nós do aglomerado computacional.

O segundo mecanismo fundamental do Hadoop é o MapReduce. Como o próprio nome sugere, o MapReduce enxerga uma tarefa computacional como consistindo de duas fases, a fase de mapeamento (Map) e a fase de redução (Reduce), que são executadas nessa mesma sequência. Durante a fase de mapeamento, todos os nós desempenham a

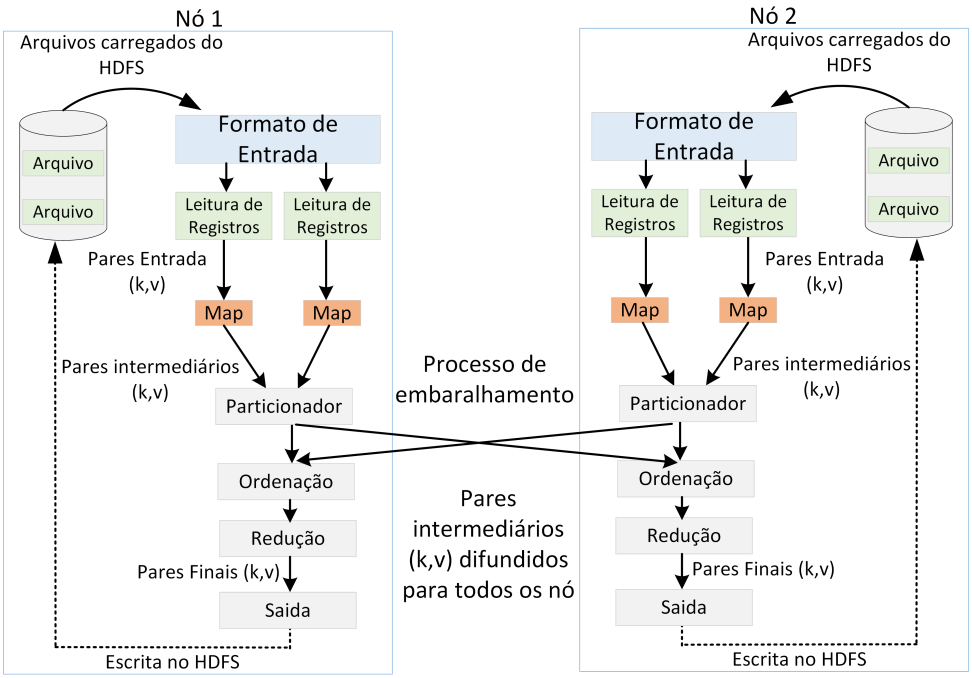


Figura 3.1. Funcionamento do MapReduce. Primeiramente os arquivos são carregados do HDFS. Logo o formato de entrada define como esses arquivos são divididos e lidos. Os arquivos são divididos em blocos e são criados os pares `<chave,valor>` (k,v). Os pares `<chave,valor>` são obtidos localmente, para logo ser intercambiados entre os nos do sistema pela função de embaralhamento. Uma vez que os pares são recebidos, eles são ordenados e reduzidos para obter o par `<chave,valor>` final que será armazenado no HDFS.

mesma tarefa computacional a partir de um subconjunto dos dados que está localizado no próprio nó ou próximo dele. Em outras palavras, o MapReduce usa o princípio da localidade dos dados para aumentar o seu desempenho e para minimizar a movimentação dos dados pelo aglomerado. É importante notar que devido a todos os blocos de arquivos no sistema de arquivos distribuídos do Hadoop terem o mesmo tamanho, a computação na fase de mapeamento pode ser igualmente dividida entre os nós. Se os blocos de arquivo não tivessem o mesmo tamanho, o tempo de processamento seria predominantemente ditado pelo tempo necessário para processar o bloco de arquivo mais longo, enquanto os outros nós permaneceriam ociosos. A saída da fase de redução consiste em uma lista de pares `<chave, valor>`.

Após a execução da função de mapeamento, cada nó produz uma lista de pares chave-valor, na qual cada chave é o identificador da parcela dado a ser processado e o valor é o resultado da computação sobre a parcela de dados identificada pelo valor. Em um exemplo como a execução de uma consulta sobre uma base dados, mostrado na Figura 3.2, a chave é a identificação da consulta e o valor é o resultado da consulta. Pode-se, então, utilizar a fase de redução para consolidação dos resultados obtidos por cada nó, através, por exemplo, da soma de valores identificados pelas mesmas chaves, para "reduzir" a saída das funções de mapeamento a uma única lista de pares chave-valor. No MapReduce, um nó é selecionado para executar a função de redução. Todos os outros nós precisam enviar a lista `<chave, valor>` criada pela própria função de mapeamento ao nó designado. O primeiro procedimento executado agrupa os pares `<chave,`

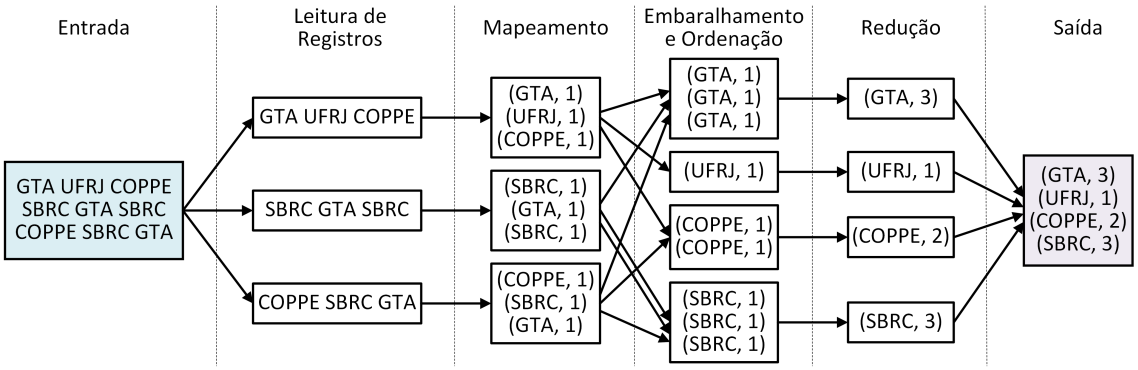


Figura 3.2. Exemplo do MapReduce. As funções principais são a entrada de dados, a leitura de registros e divisão em blocos, o mapeamento em `<chave, valor>`, o embaralhamento e a ordenação dos pares, a redução e finalmente a saída.

`valor>` com a mesma chave de forma ordenada. Em seguida, o procedimento executado é o procedimento de combinação que agrupa os pares `<chave, valor>` com a mesma chave em uma única entrada. O agrupamento é realizado de forma que cada entrada seja composta de uma chave e uma lista com todos os valores relacionadas a mesma chave no procedimento anterior. Finalmente, o procedimento de redução soma os valores associados a cada chave existente do par `<chave, valor>`. No arcabouço MapReduce, ambas operações de mapeamento e redução são consideradas rotinas, que combinadas formam uma tarefa.

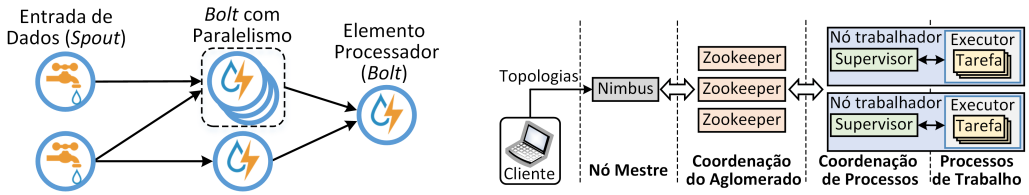
3.2.2. A plataforma Apache Storm

O Apache Storm[8] [Toshniwal et al. 2014] é uma plataforma de código aberto para processamento distribuído de fluxo escrito nas linguagens Java e Clojure. Nesta plataforma, um fluxo de dados consiste em um identificador e um conjunto ilimitado de tuplas n-dimensionais. As aplicações são abstraídas em topologias, análogas a um trabalho (*job*) de MapReduce no Hadoop, que consistem em grafos acíclicos dirigidos nos quais as arestas representam fluxos e os vértices representam os nós de entrada (*spouts*) ou nós de processamento (*bolts*) de dados. Os *spouts* são responsáveis pela abstração dos dados de entrada em tuplas que fluem pelo grafo. Os *bolts* são elementos processadores que executam uma operação atômica definida pelo usuário à medida que o fluxo de dados avança no grafo. Tanto *bolts* como *spouts* são paralalelizáveis e podem ser definidos com um grau de paralelismo que indica a quantidade de tarefas concorrentes presentes em cada nó. Um exemplo de topologia com dois *spouts* e três *bolts* é mostrado na Figura 3.3(a).

Outro conceito importante no Storm é o agrupamento de fluxos (*stream grouping*), que define como o fluxo é distribuído entre as tarefas de um *bolt*. Assim, a decisão sobre o tipo de agrupamento permite que o usuário defina como os dados devem fluir na topologia. O Storm possui oito tipos de agrupamento de fluxos que representam diferentes maneiras de enviar tuplas ao *bolt* seguinte, dos quais se destacam: i) por embaralhamento (*shuffle*),

[8]Nathan Marz, doutor pela Universidade de Stanford, trabalhando na empresa BackType, desenvolve o Storm em 2011, um arcabouço para processamento distribuído de fluxo, para tratar em tempo real a grande quantidade de mensagens (*tweets*) recebidas pela empresa Twitter. A empresa BackType é adquirida pelo Twitter e o Storm torna-se de código aberto, migrando para a Fundação Apache em 2013.

no qual cada tarefa em um *bolt* recebe o mesmo número de tuplas, distribuídas de maneira aleatória e uniforme; ii) por campos (*fields*), no qual as tuplas que possuem valores iguais para um conjunto de campos são enviadas à mesma tarefa de um *bolt*; iii) por difusão (*all*), na qual todas as tuplas são enviadas a todas as tarefas do *bolt* seguinte; e iv) global, na qual todo o fluxo é enviado à tarefa de menor identificador. Cada envio de fluxo, isto é, cada aresta na topologia, deve possuir um tipo de agrupamento decidido pelo usuário, que deve levar em conta as características da sua aplicação.



(a) Exemplo de topologia do Storm. (b) Arquitetura do Apache Storm.

Figura 3.3. Apache Storm. a) uma topologia do Storm com elementos *Spouts*, nós de entrada, e *Bolts*, nós que realizam um processamento nas tuplas; b) a arquitetura do Apache Storm. O Nimbus controla as topologias, se comunicando com os supervisores que coordenam os processos nos trabalhadores. O controle do estado da topologia é feito pelo *Zookeeper*.

A Figura 3.3(b) mostra a coordenação dos processos do Storm em um aglomerado computacional. Um usuário envia suas topologias ao Nimbus, o nó mestre da arquitetura, que coordena a implementação das tarefas de *bolts* e *spouts* em nós trabalhadores através do Apache Zookeeper. O Zookeeper é responsável pelo gerenciamento dos nós trabalhadores e pelo armazenamento estado de todos os elementos do sistema. Em cada nó trabalhador, um supervisor monitora os executores, que são processos responsáveis por executar uma ou mais tarefas. Os supervisores informam o estado e a disponibilidade dos executores através de um mecanismo de sinais periódicos *heartbeat*, permitindo que o Nimbus identifique falhas no sistema. Falhas de executores são tratadas pelos próprios supervisores, que reiniciam os processos correspondentes no nó trabalhador. Uma falha de supervisor é tratada pelo Nimbus, que pode realocar todas as tarefas do supervisor em falha para outro nó trabalhador. Se o Nimbus falha, o sistema ainda é capaz de executar todas as topologias pendentes, porém o usuário não é capaz de submeter novas topologias. Após recuperação, o Nimbus e os supervisores podem retomar o último estado armazenado no Zookeeper.

O Apache Storm usa mecanismos de armazenamento e reconhecimento (ACK) para garantir o processamento das tuplas mesmo após uma falha. Para isto, todas as tuplas são identificadas pelos spouts e seus identificadores são enviadas a um *bolt* especial, denominado *bolt* de reconhecimento (*acker bolt*), que armazena o estado de cada tupla. Um exemplo de topologia com *acker bolt* é mostrado na Figura 3.4. A cada tupla processada, um *bolt* deve enviar um reconhecimento positivo (ACK) para o *acker bolt*. Caso todas as tuplas recebam um ACK para cada *bolt*, o *bolt* de reconhecimento descarta os IDs e informa ao *spout* que o processamento foi realizado com sucesso. Caso contrário, o *bolt* de reconhecimento solicita ao *spout* o reenvio de todas as tuplas e o sistema retrocede até o ponto de falha. O não-recebimento de um ACK é reconhecido pela expiração de um

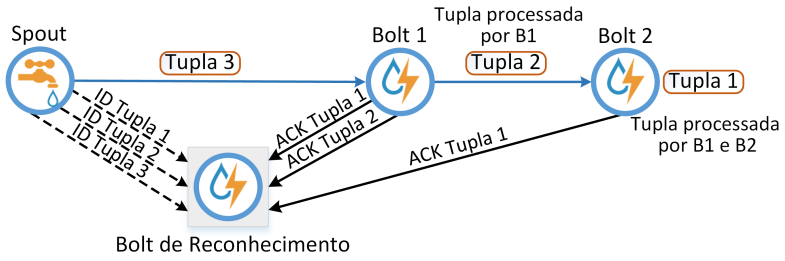


Figura 3.4. Semântica de entrega de mensagens "pelo menos uma vez' (*at least once*) utilizada no Apache Storm. Cada vez que uma tupla é emitida pelo *spout*, um registro é emitido ao *bolt* de reconhecimento. Logo quando a tupla é processada por um *bolt* um ACK é emitido reconhecendo o processamento.

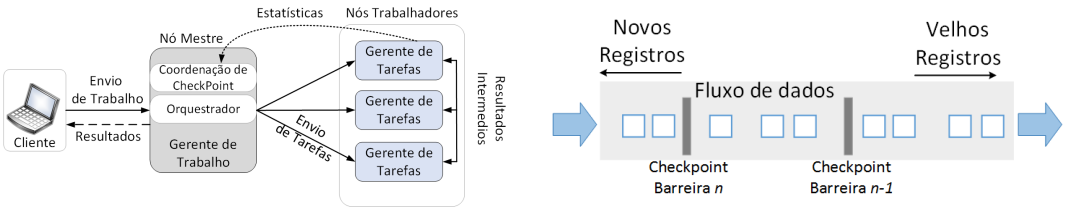
temporizador (*timeout*) definido no *bolt* de reconhecimento. Isto garante a semântica de entrega de mensagens "pelo menos uma vez" (*at least once*), em que cada tupla seja processada uma ou várias vezes, no caso de reenvio. Ainda é possível desativar o *acker bolt* para aplicações que não necessitem de garantias de processamento.

3.2.3. A Plataforma Apache Flink

O Apache Flink[9] [Carbone et al. 2015a] é uma plataforma de processamento híbrido, que suporta processamento de fluxo e em lotes. O núcleo do Flink é o processamento de fluxo, tornando o processamento em lotes um caso especial. As tarefas de analítica do Flink são abstraídas em grafos acíclico dirigido (GAD) formado por quatro componentes: fontes; operadores; torneiras de saída; e registros que percorrem o grafo. A abstração da topologia é realizada através da programação em Java ou Scala. Assim como no Storm, a divisão de trabalho é baseada em um modelo mestre-trabalhador. A Figura 3.5(a) mostra a arquitetura do Apache Flink. O nó mestre do Flink é denominado gerente de trabalho e interage com aplicações de clientes com responsabilidades semelhantes ao nó mestre do Storm (Nimbus). O gerente de trabalho recebe aplicações de clientes, organiza as tarefas e as envia para os nós trabalhadores, que recebem o nome de gerente de tarefas. Além disso, o gerente de trabalho mantém o estado de todas as execuções e de cada trabalhador. Os estados dos trabalhadores são informados através de um mecanismo de sinais periódicos (*heartbeat*). O gerente de tarefas tem uma função semelhante ao nó trabalhador no Storm. Os gerentes de tarefas executam tarefas atribuídas pelo gerente de trabalho e trocam informações com outros gerentes de tarefas quando necessário. Cada gerente de tarefas disponibiliza *slots* de processamento para o aglomerado computacional, que são utilizados para executar tarefas em paralelo.

A abstração do fluxo de dados no Flink é chamada DataStream e é definida como uma sequência de registros parcialmente ordenados. Parcialmente, pois não há garantia de ordem caso um elemento operador receba mais de um fluxo de dados como entrada. DataStreams são semelhantes às tuplas do Storm e recebem dados de fluxo de fontes externas, como filas de mensagens, *sockets* e outros. A programação de DataStream oferece suporte a várias funções nativas para operar fluxos de dados, como *map*, *filtering*, *reduc-*

[9]Flink nasceu em 2010 de um projeto de pesquisa Europeu denominado "Stratosphere: Gestão da Informação na Nuvem" desenvolvido em colaboração da Universidade Técnica de Berlim, Humboldt-Universität zu Berlin e Hasso-Plattner-Institut Potsdam. Em 2014, o Stratosphere muda o nome do projeto para Flink e abre o seu código na Fundação Apache.



(a) Arquitetura do plataforma Flink.

(b) Esquema de pontos de verificação.

Figura 3.5. A arquitetura da plataforma Apache Flink: a) o gerente de trabalho recebe os trabalhos desde o cliente, divide o trabalho em tarefas e envia para os trabalhadores que comunicam as estatísticas e os resultados; b) as barreiras são injetadas nos elementos de origem e passam pelo grafo junto com as amostras. Quando o fluxo passa pelos operadores capturas instantâneas do estado são disparadas. Quando um operador recebe uma barreira de todos os fluxos de entrada, ele verifica seu estado para o armazenamento.

tion, *join* etc., que são aplicadas de forma incremental a cada entrada, gerando um novo DataStream. Cada uma destas operações pode ser paralelizada configurando-se um parâmetro de paralelismo. Assim, as instâncias paralelas das operações são atribuídas aos *slots* de processamento disponíveis dos gerentes de tarefas para tratar partições do DataStream simultaneamente. Este método permite a execução distribuída de operações nos fluxos de dados.

O Flink possui uma semântica de entrega confiável de mensagens do tipo "exatamente uma". Esta semântica se baseia no esquema de tolerância a falha com pontos de verificação, ou barreiras (*checkpoints barriers*), para que o sistema possa retornar em caso de falha. As barreiras funcionam como registros de controle e são injetadas regularmente no fluxo de dados pelos elementos fontes para fluírem pelo grafo juntamente com os registros de dados. A Figura 3.5(b) mostra a implementação do esquema de pontos de verificação por barreiras. Quando uma barreira atravessa um elemento operador, este realiza uma captura momentânea do estado do sistema (*snapshot*). A captura consiste em armazenar o estado do operador, por exemplo, o conteúdo de uma janela deslizante ou uma estrutura customizada de dados, e a sua posição no fluxo de dados. Após uma fase de alinhamento entre operadores para certificar de que a barreira atravessou todos os operadores que recebem aquele fluxo, os operadores escrevem a captura gerada em um sistema de arquivos duráveis, como o HDFS. Em caso de falha de *software*, nó ou rede, o Flink interrompe o DataStream. Em seguida, o sistema reinicia imediatamente os operadores e recomeça a partir do última captura bem-sucedida armazenada. Assim como no Storm, tolerância a falhas do Flink é garantida partindo da premissa que o sistema é precedido por um sistema de mensagens com reenvio persistente, como o Apache Kafka. No caso especial de processamento em lotes, não há esquema de tolerância a falhas, e caso ocorra uma falha, toda a operação deve ser recomeçada do zero.

3.2.4. A Plataforma Apache Spark

Spark[10] [Franklin 2013] é uma plataforma para processamento distribuído de dados, escrita em Java e Scala. O Spark é uma implementação do MapReduce do Hadoop,

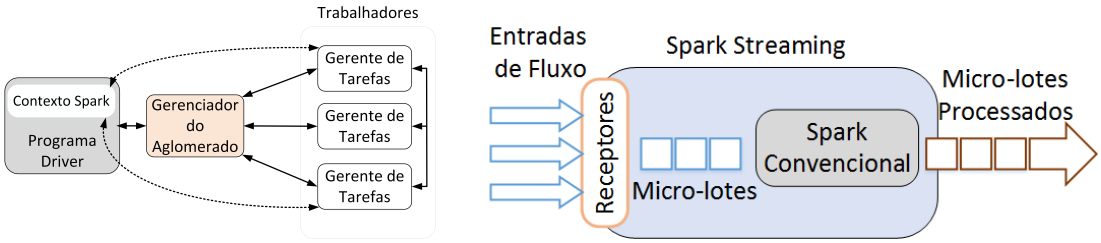
[10]A plataforma Spark foi desenvolvida, em 2012, pelo AmpLab da Universidade de Berkeley. Em 2011, a Universidade de Berkeley recebeu 30 milhões de dólares para para criar o AmpLab e financiar pesquisas análise de dados em tempo real. O projeto Spark foi doado em 2013 para a Fundação Apache.

que processa as tarefas em memória, evitando a escrita e a leitura em disco de resultados intermediários. A principal diferença entre o Spark e o Hadoop é que todo o envio de dados entre os nós do Hadoop é realizado através de escritas e leituras no sistema de arquivo distribuído (HDFS), enquanto no Spark, os envios de dados entre processos são realizados por comunicação entre processos que mantém os dados em memória. Assim, o Spark não introduz a necessidade de realização de repetidas escritas e leituras em disco, operações reconhecidamente lentas. O processamento em memória torna o Spark mais rápido que o MapReduce, obtendo muito sucesso nas aplicações que realizam diversas rodadas de transformações sobre grandes conjuntos de dados, como as de aprendizado de máquina. Vale ressaltar que o Spark, assim como o Hadoop, realiza processamento em lotes e, portanto, não é apropriado para analítica em fluxos em tempo real. Por sua vez, a extensão Spark Streaming [Zaharia et al. 2013], baseada em microlotes, foi desenvolvida em 2014 como uma biblioteca executando no topo do Spark Engine para realizar analítica em tempo real.

A biblioteca Spark Streaming discretiza um fluxo de dados em lotes pequenos de dados, chamados microlotes. Cada microlote é um conjunto de tuplas de dados armazenado em memória. O lote é a unidade de dados tratada pelo Spark Engine. A abstração do fluxo é chamada de fluxo discreto (*Discrete Stream* - D-Stream) e é definida como um conjunto de tarefas curtas, sem estado e discretizadas. No Spark, como mostra a Figura 3.6(b), o processamento de fluxos é tratado como uma série de processamento em lotes discretos em intervalos de tempo menores, os microlotes. Semelhante ao MapReduce, um trabalho no Spark é definido como uma computação paralela que consiste em várias tarefas. A tarefa é uma unidade de trabalho que é enviada para o Gerenciador de Tarefas. Quando um fluxo chega ao Spark, os dados são discretizados em microlotes que são distribuídos de forma a garantir a resiliência e, portanto, os microlotes são implementados por uma estrutura de dados chamada de Conjunto de Dados Distribuídos Resilientes (*Resilient Distributed Dataset* - RDD). O RDD é a principal estrutura de dados do Spark Engine que armazena os dados em memória e os distribui garantindo a resiliência. É importante ressaltar que há a necessidade de se garantir a resiliência na distribuição dos dados em fluxo, pois os dados são sempre armazenados em memória volátil, ao contrário do que ocorre com o Hadoop em que os dados são sempre armazenados em disco, memória não volátil. Um D-Stream é uma sequência potencialmente infinita de conjuntos de dados distribuídos resilientes (RDD). Em seguida, o Spark Engine é executado gerando trabalhos para processar os microlotes representados pela estrutura de dados RDD.

As operações realizadas com Spark Streaming são realizadas sobre D-Streams. As operações do Spark Engine ocorrem sobre os RDDs que estão contidos em um D-Stream. Assim, ao operar com Spark Streaming são realizados dois níveis de distribuição de dados. O primeiro nível itera e distribui os RDDs contidos em uma fração do D-Stream entre diversos nós, semelhante a uma operação de mapeamento (*map*), em que cada nó recebe uma partição do conjunto de RDDs. No segundo nível, opera-se sobre os RDDs, em que os nós realizam o mapeamento e a redução das tuplas de dados contidas em cada RDD. As operações de ações e transformações sobre os dados ocorrem através do mapeamento e redução sobre os RDDs.

A Figura 3.6(a) mostra a configuração de um aglomerado computacional Spark. Os trabalhos no Spark são executados como processos independentes no aglomerado,



(a) Arquitetura do sistema Spark. (b) Utilização dos microlotes.

Figura 3.6. Distribuição de dados e tarefas no Spark. a) arquitetura do aglomerado do sistema Spark Streaming: o programa de driver, o gerenciador do aglomerado e os nós de trabalho, que executam os gerenciadores de tarefas; b) processamento de microlote usado no Spark Streaming, no qual os fluxos de entrada são recebidos pelos receptores e são transformados em microlotes que são executados em um mecanismo tradicional Map-Reduce.

que é coordenado pelo mestre ou o *Driver Program*, responsável pelo agendamento de tarefas e pela criação do *Spark Context*. O *Spark Context* se conecta a vários tipos de gerenciadores de aglomerado, como o Spark StandAlone, Mesos ou Hadoop YARN (*Yet Another Resource Negotiator*). Esses gerenciadores de aglomerado são responsáveis pela alocação de recursos entre aplicativos. Uma vez conectado, o Spark executa a tarefa dentro dos gerenciadores de tarefas, que executam o processamento e o armazenamento de dados, equivalentes aos trabalhadores Storm, e os resultados são então comunicados ao *Spark Context*. O mecanismo descrito para o Storm, no qual cada processo de trabalho é executado dentro de uma topologia, pode ser aplicado ao Spark, no qual aplicativos ou trabalhos são equivalentes as topologias. A desvantagem deste conceito no Spark é a troca de mensagens entre diferentes programas, que é feito indiretamente, como escrever dados em um arquivo, piora a latência que pode ser da ordem de segundos em aplicações de várias operações.

Como o Spark opera sobre dados armazenados na memória volátil, há a necessidade de se prover a tolerância a falhas para os dados enquanto estão sendo processados, e não somente após a gravação em disco como feito no Hadoop. O Spark tem semântica de entrega de mensagens “exatamente uma vez”. A ideia é processar uma tarefa em vários nós de trabalho distintos e, no evento de uma falha, o processamento do microlote pode ser redistribuído e recalculado. O estado dos RDDs é periodicamente replicado para outros nós de trabalho. As tarefas são então discretizadas em tarefas menores executadas em qualquer nó, sem afetar a execução. Assim, as tarefas com falhas podem ser lançadas em paralelo, distribuindo uniformemente a tarefa, sem afetar o desempenho. Este procedimento é chamado de recuperação paralela. A semântica de “exatamente uma vez” reduz a sobrecarga em comparação ao *upstream backup*, em que todas as tuplas devem ser positivamente reconhecidas, como no Storm. No entanto, o processamento de microlote tem desvantagens. O processamento de microlotes leva mais tempo nas operações para baixo (*downstream*). A configuração e distribuição de cada microlote pode demorar mais do que a taxa de chegada do fluxo nativo. Consequentemente, microlotes podem ficar enfileirados na fila de processamento.

Os criadores da Spark fundaram a DataBricks[11] em 2013, uma empresa para dar suporte na ferramenta Spark. O Twitter ainda gerencia o projeto Storm. Da mesma forma que a DataBricks está para o Spark, os proprietários do Apache Flink criam a Data Artisan[12]. Em 2011, o Yahoo desenvolveu o S4 [Neumeyer et al. 2010] e doou para a fundação Apache, mas o projeto nunca evoluiu para incubação e foi interrompido em 2015. As grandes empresas desenvolvem seus próprios produtos. Google Millwheel [Akidau et al. 2013], IBM InfoSphere Streams [Ballard et al. 2014], Microsoft Azure Analytics[13] entre outros são exemplos de soluções proprietárias.

Esquemas de Tolerância a Falhas nas Plataformas de Processamento Distribuído

Um esquema robusto de tolerância a falha é essencial para as plataformas de processamento distribuído que rodam em aglomerados, que são sensíveis a falhas de nó, de rede e de *software*. Deve ser observado que um centro de dados possui uma estrutura em aglomerados computacionais, nos quais os nós são servidores de baixo custo (*commercial off-the-shelf* - COTS). Nos sistemas de processamento em lote, a latência é aceitável e, consequentemente, o sistema não precisa se recuperar rapidamente de uma falha. No entanto, em sistemas em tempo real, as falhas podem significar perda de dados, uma vez que os dados não estão armazenados. Portanto, a recuperação rápida e eficiente é importante para evitar a perda de informações [Kamburugamuve et al. 2013].

A forma mais comum de recuperação de falhas é o armazenamento para reenvio (*upstream backup*). Ao considerar a topologia de processamento, o algoritmo usa os nós pais para atuar como *backups*, armazenando e preservando temporariamente as tuplas em suas filas de saída até que seus vizinhos, descendentes, as processem e enviem um reconhecimento positivo (*acknowledgement* - ACK). Toda tupla deve ser individualmente reconhecida positivamente com um ACK. Se algum desses vizinhos descendentes falhar, um ACK não será enviado e, por estouro de temporizador, o nó pai reproduz as tuplas em um outro nó. Uma outra forma de reconhecimento positivo é por grupo de tuplas. Basta identificar uma tupla que falta que o grupo inteiro de tuplas é reproduzido.

Uma desvantagem desta abordagem é o tempo de recuperação longo, já que o sistema deve aguardar até o nó protegido assumir o controle. Para abordar este problema, em [Zaharia et al. 2013] é proposto o algoritmo de recuperação paralela. Neste algoritmo, o sistema verifica periodicamente os estados replicando de maneira assíncrona para outros nós. Quando um nó falha, o sistema detecta as partições em falta e lança tarefas para recuperá-las do último ponto de verificação. Muitas tarefas podem ser lançadas ao mesmo tempo para calcular diferentes partições em diferentes nós. Assim, a recuperação paralela é mais rápida do que o *backup* a montante.

Outra solução é proposta em [Carbone et al. 2015b] com base no algoritmo de instantâneos de barreira assíncrona (*Asynchronous Barrier Snapshotting*- ABS). A ideia central é fazer uma marcação do estado global de um sistema distribuído. Na proposta, um *snapshot*, ou um instantâneo, é o estado global do grafo de processamento, capturando todas as informações necessárias para reiniciar o cálculo desse estado de execução específico. Uma barreira separa o conjunto de registros que acompanham o instantâneo

[11] DataBricks analytics https://databricks.com/

[12] Data Artisan Analytics http://data-artisans.com/)

[13] Análise Microsoft Azure https://azure.microsoft.com/en-us/services/stream-analytics/

atual dos registros que se inserem no próximo instantâneo. As barreiras não interrompem o fluxo de dados. Várias barreiras de instantâneos diferentes podem estar no fluxo ao mesmo tempo, o que significa que vários instantâneos podem ocorrer simultaneamente. Quando uma fonte recebe uma barreira, a fonte tira um *snapshot* do seu estado atual e, então, transmite a barreira para todas as saídas. Quando uma tarefa não-fonte recebe uma barreira de uma de suas entradas, ela bloqueia essa entrada até receber uma barreira de todas as entradas. Quando as barreiras foram recebidas de todas as entradas, a tarefa faz um *snapshot* de seu estado atual e transmite a barreira para suas saídas. Então, a tarefa desbloqueia seus canais de entrada para continuar sua computação. Assim, a recuperação de falhas reverte todos os estados do operador para seus respetivos estados retirados do último instantâneo bem-sucedido e reinicia os fluxos de entrada a partir da última barreira para a qual há um instantâneo.

A semântica de garantia de entrega que um sistema oferece para processar ou não uma amostra pode ser divididas em três tipos: "exatamente uma vez" (*exactly once*), "pelo menos uma vez" (*at least once*), e "no máximo uma vez" (*at most once*). A semântica mais simples a "no máximo uma vez", em que não há recuperação de erro, ou seja, as amostras são processadas ou perdidas. Quando ocorre uma falha, os dados podem ser encaminhados para outro elemento de processamento sem perder informações. Na semântica "exatamente uma vez" os reconhecimentos positivos são individuais por tupla. Na semântica "pelo menos uma vez", a correção de erro é feita em conjunto para um grupo de amostras, dessa forma, se ocorrer um erro com qualquer uma das amostras, o grupo inteiro é reprocessado e, portanto, é possível que algumas amostras sejam processadas mais de uma vez. A semântica "pelo menos uma vez" é menos custosa do que "exatamente uma vez", que requer a confirmação individual para cada tupla processada.

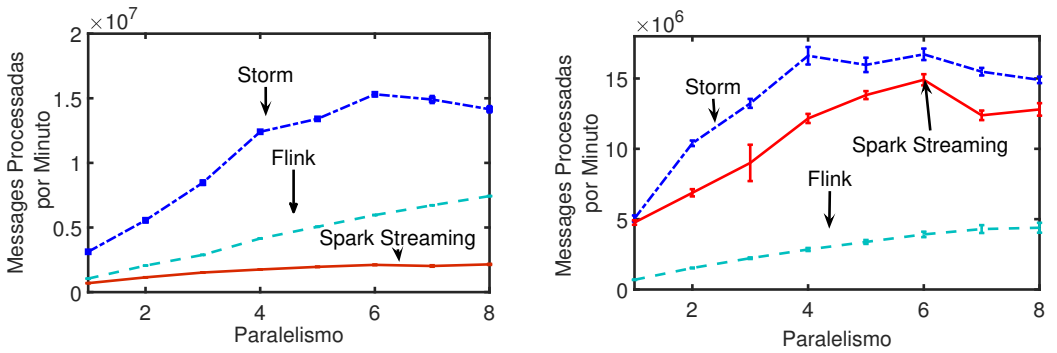
A Tabela 3.2 apresenta um resumo das características de comparação dos sistemas de processamento de fluxo. O modelo de programação pode ser classificado como composicional e declarativo. A abordagem composicional fornece blocos de construção básicos, como *Spouts* e *Bolts* em Storm e devem ser conectados entre si para criar uma topologia. Por outro lado, os operadores do modelo declarativo são definidos como funções de ordem superior, que permitem escrever código funcional com tipos abstratos e o sistema cria automaticamente a topologia.

Tabela 3.2. Comparação entre plataformas de processamento distribuído de fluxo.

	Storm	**Flink**	**Spark Streaming**
Abstração do Fluxo	Tuplas	DataStream	D-Stream
Ling. Programação	Java/Clojure	Java/Scala	Java/Scala
Tolerância Falhas	Mínimo 1 vez	Exatamente 1 vez	Exatamente 1 vez
API	Composicional	Declarativa	Declarativa
Mecanismo Falha	*Backup*	Barreiras	Recuperação
Subsist. de Falhas	Nimbus, Zookeeper	Não há	Não há

3.3. Avaliação de Desempenho das Plataformas para a Classificação com Aprendizado de Máquina

Esta seção avalia a taxa de processamento e o comportamento durante a falha do nó das três plataformas de processamento de fluxo apresentadas: Apache Storm versão 0.9.4, Apache Flink versão 0.10.2 e Apache Spark Streaming versão 1.6.1, com tamanho de microlotes estabelecido em 0,5 segundos. A aplicação avaliada é um sistema de detecção de ameaças com um classificador de rede neural programado em Java. As experiências foram realizadas em um ambiente com oito máquinas virtuais executadas em um servidor com o processador Intel Xeon E5-2650 a 2,00 GHz e 64 GB de RAM. A configuração de topologia de experiência é um mestre e sete nós de trabalho para os três sistemas avaliados. Os resultados são apresentados com um intervalo de confiança de 95%. O Apache Kafka na versão 0.8.2.1, que opera como um serviço de publicação/inscrição, foi usado para inserir dados a taxas elevadas nos sistemas de processamento de fluxo. No Kafka, as amostras, ou eventos, são chamadas de mensagens, nome usado a partir de agora. O Kafka abstrai o fluxo de mensagens em tópicos que atuam como *buffers* ou filas, ajustando diferentes taxas de produção e consumo. Portanto, os produtores registram os dados em tópicos e os consumidores leem os dados desses tópicos. O conjunto de dados usado foi criado pelos autores [Andreoni Lopez et al. 2017a], que foi replicado para obter dados suficientes para avaliar o máximo de processamento no qual o sistema pode operar.



(a) Mensagens processadas por minuto aplicação de detecção de intrusão.

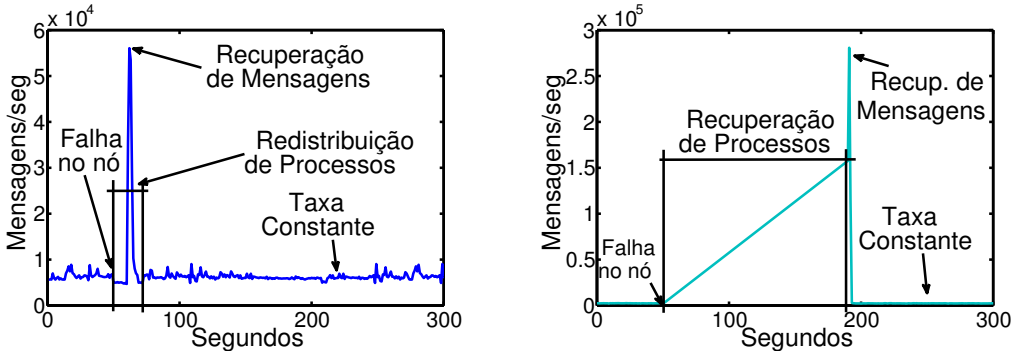
(b) Mensagens processadas por minuto aplicação de contagem de palavras

Figura 3.7. Comparação do paralelismo em plataformas de processamento de fluxo. a) Número de mensagens processadas por minuto em função do paralelismo de tarefas para a aplicação de detecção de intrusão no conjunto de dados de segurança. b) Número de mensagens processadas por minuto em função do paralelismo de tarefas para a aplicação de contagem de palavras no conjunto de dados do Twitter.

O primeiro experimento avalia o desempenho das plataformas em termos de processamento [Andreoni Lopez et al. 2018]. O conjunto de dados é injetado no sistema em sua totalidade e replicado quantas vezes for necessário para criar um grande volume de dados. O experimento calcula a taxa de consumo e processamento de cada plataforma. Também foi variado o parâmetro de paralelismo, que representa o número total de núcleos disponíveis para o *cluster* processar amostras em paralelo. A Figura 3.7(a) mostra os resultados da experiência. O Apache Storm apresenta a maior taxa de transferência.

Para um único núcleo, sem paralelismo, o Storm já mostra um melhor desempenho, com uma vazão pelo menos 50% maior quando comparado a Flink e Spark Streaming. Flink tem um crescimento linear, mas com valores sempre inferiores aos de Apache Storm. A taxa de processamento de Apache Spark Streaming, quando comparada a Storm e a Flink, é muito inferior e isso se deve ao uso de microlote. Cada microlote é agrupado antes do processamento, gerando um atraso no processamento por amostras. O comportamento de Apache Storm é linear até o paralelismo de quatro núcleos. Então, a taxa de processamento cresce até o paralelismo de seis, em que o sistema satura. Esse comportamento também foi observado em Apache Spark Streaming com o mesmo paralelismo de seis núcleos.

Para avaliar o desempenho outro experimento foi criado. Esse experimento conta o número de vezes que cada palavra aparece em um texto, usando um conjunto de dados que contém mais de 5.000.000 tweets [Cheng et al. 2010]. As três plataformas oferecem o aplicativo de contagem de palavras como exemplos de tutoriais, portanto, mostramos esse resultado para obter uma comparação imparcial que não é afetada pela implementação do código em cada plataforma. Figura 3.7(b) mostra o desempenho dos três sistemas no programa `wordcount`. Esse experimento mostra um resultado semelhante ao mostrado anteriormente.

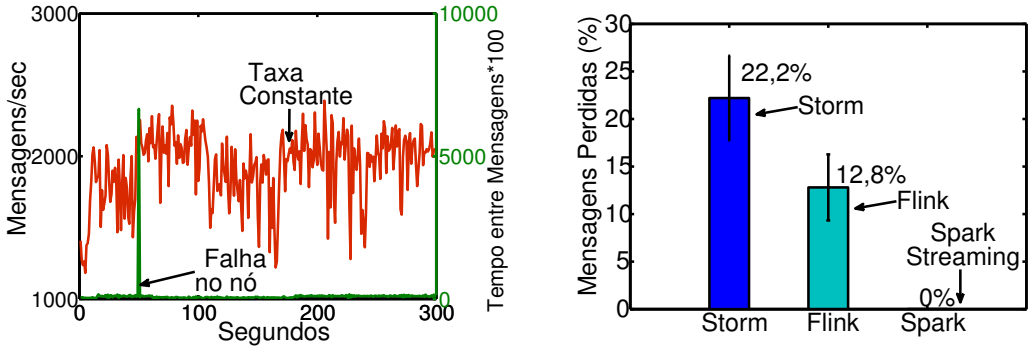


(a) Comportamento da ferramenta Storm ante uma falha de nó.

(b) Comportamento da ferramenta Flink ante uma falha de um nó.

Figura 3.8. Comportamento de Storm e Flink durante uma falha de nó. Uma falha é produzida em 50 segundos. a) Storm e b) Flink. Comportamento dos sistemas após detectar a falha consiste em procedimentos de redistribuição de processos e recuperação de mensagens.

O próximo experimento mostra o comportamento do sistema quando um nó falha. As mensagens são enviadas a uma taxa constante para analisar o comportamento do sistema durante a falha. A falha do nó é simulada desligando uma máquina virtual. As figuras 3.8(a), 3.8(b) e 3.9(a) mostram o comportamento dos três sistemas antes e depois de uma falha do nó do trabalhador em 50 segundos. O Apache Storm leva algum tempo nos processos de redistribuição após a detecção da falha. Esse tempo é devido à comunicação com o *Zookeeper*. O *Zookeeper* tem uma visão geral do *cluster* e reporta o estado ao Nimbus no Storm, que redistribui os processos em outros nós. Logo após esta redistribuição, o sistema recupera mensagens do Kafka em aproximadamente 75 segundos. Embora o sistema possa recuperar rapidamente da falha do nó, durante o processo



(a) Comportamento da ferramenta Spark ante uma falha de um nó.

(b) Comparação da perda de mensagens nos três sistemas durante a falha do nó.

Figura 3.9. Comportamento do Spark Streaming ante uma falha de nó e comparação de perdas de mensagens. a) O comportamento do sistema Spark sob falha, indicando que ele se mantém estável e não perde mensagens. b) Porcentagem de perdas de mensagens.

há uma perda de mensagem significativa. Um comportamento semelhante é observado no Apache Flink. Depois de detectar a falha em aproximadamente 50 segundos, o sistema redistribui os processos para nós ativos. O Flink faz esse processo internamente sem a ajuda de nenhum subsistema, ao contrário do Apache Storm que usa o *Zookeeper*.

A Figura 3.8(b) mostra o período de tempo em que o Flink redistribui processos é muito maior do que o tempo gasto no Apache Storm. No entanto, a recuperação de mensagens também é maior, perdendo algumas mensagens durante a redistribuição do processo. A Figura 3.9(a) mostra o comportamento de Spark Streaming durante uma falha. Quando ocorre uma falha em aproximadamente 50 segundos, o comportamento do sistema é basicamente o mesmo que antes. Isto é devido ao uso de tarefas em micro-lote que são distribuídas rapidamente, sem afetar o desempenho. O Spark Streaming não mostra perda de mensagem durante a falha. Assim, apesar do baixo desempenho do Spark Streaming, esse pode ser uma boa escolha em aplicações em que a resiliência e o processamento de todas as mensagens são necessários. Figura 3.9(b) mostra a comparação de mensagens perdidas entre o Storm, o Flink e o Spark. O Spark não teve perda durante a falha. O resultado mostra o percentual de mensagens perdidas por sistemas, calculada pela diferença de mensagens enviadas pelo Apache Kafka e por mensagens analisadas pelos sistemas. Assim, Apache Flink tem uma menor perda de mensagens durante uma falha com cerca de 12,8% em comparação com 22,2% em Storm. Os resultados foram obtidos com intervalo de confiança de 95%. Pode-se concluir que o Apache Storm apresentou os melhores resultados em relação a vazão de dados. Por outro lado, se a tolerância a falhas e o processamento de mensagens "exatamente uma vez" for necessário, o Apache Spark e o Apache Flink são boas opções.

3.4. O Aprendizado de Máquina e o Problema de Detecção de Ameaças em Tempo Real

Em Segurança da Informação, define-se como **ameaça** um evento externo que explora uma **vulnerabilidade**, interna a um sistema, causando um **incidente** de segurança que tem como efeito algum **impacto**. Nesse contexto, a vulnerabilidade é intrínseca do

sistema e pode nunca vir a ser descoberta. Ao ser descoberta uma nova vulnerabilidade, essa passa a ser possível de ser explorada por um agente externo, configurando uma ameaça. Ao longo deste capítulo, são definidas como ameaças a exploração, a tentativa de exploração ou até mesmo a busca por vulnerabilidades em redes de computadores. Tais ações apresentam comportamentos característicos e identificá-los entre o tráfego legítimo é um desafio em segurança. Outro desafio é detectar as ameaças de forma rápida, e se possível, em tempo real. A detecção tardia de ameaças, pela análise de conjuntos de dados armazenados previamente, impede o acionamento de contramedidas eficazes. Além disso, na era das Grandes Massas de Dados (*big data*), a alta capacidade de sistemas de defesa de processar grandes volumes de dados para detectar ameaças é fundamental. Quando a ameaça explora uma vulnerabilidade recém descoberta, chama-se de ameaça inédita (*zero-day threat*). Ameaças inéditas configuram outro grave problema para a segurança, pois passam despercebidas por sistemas de defesas, como sistemas de detecção de intrusão e antivírus, podendo permanecer indetectáveis por semanas e causar impactos catastróficos. Nesse sentido, as técnicas de aprendizado de máquina permitem aprender com o comportamento da rede, se adaptar a mudanças no tráfego de forma automática e detectar ameaças, até então, desconhecidas. Aplicar estas técnicas em linha (*online*) é fundamental para detectar rapidamente e até impedir a ocorrência de incidentes de segurança, limitando os impactos causados por uma ameaça [Chandrasekar et al. 2017]. Portanto, o aprendizado de máquina integrado à tecnologia de processamento distribuído de fluxo é uma poderosa estratégia para solucionar o problema de detecção de ameaças em tempo real.

O problema da detecção de ameaças em tempo real refere-se à classificação dos dados, representados como amostras, assim que chegam ao sistema de classificação. Em geral, as amostras são classificadas em normal ou ameaça. Normal refere-se ao tráfego benigno, enquanto ameaça refere-se ao tráfego suspeito de ser malicioso. O problema de detecção de ameaças em tempo real é um problema de classificação de dados de fluxo potencialmente ilimitados, em que a tomada de decisão deve ser limitada a tempos próximo ao tempo real [Gaber et al. 2005]. Portanto, a classificação de dados em fluxo em tempo real apresenta os desafios da escolha do conjunto de dados adequado, do algoritmo de classificação, das características de representação dos dados, da arquitetura de detecção em fluxo, e, por fim, da avaliação e comparação de classificadores.

Algoritmos de aprendizado de máquina têm a capacidade de aprender de forma automática com os dados, ou seja, sem serem explicitamente programados para este fim. O algoritmo procura aprender e descobrir como realizar tarefas importantes a partir de generalizações dos exemplos. Como cada vez mais dados estão sendo disponibilizados, a tomada de decisão com a intervenção humana torna-se cada vez mais difícil e, então, a proposta de usar aprendizado de máquina para automatizar tarefas fica cada vez mais atrativa. No entanto, o sucesso no desenvolvimento de aplicações de aprendizado de máquina requer diferentes aspectos que não são fáceis de serem seguidos. Um primeiro desafio é escolher qual algoritmo de aprendizado de máquina que deve ser usado, pois existem milhares de algoritmos e todo ano centenas de novos algoritmos são propostos [Domingos 2012]. Esta seção foca em algoritmos de aprendizado de máquina aplicados à arquitetura de detecção de ameaças em tempo real em redes de computadores. Para isso, são introduzidos conceitos e exemplos de conjunto de dados, as técnicas de seleção e de redução

de características de tráfego e as métricas de avaliação de desempenho de classificadores mais utilizadas na detecção de ameaças. As arquiteturas de detecção em tempo real, tanto com treinamento prévio quanto com treinamento adaptativo para a detecção de ataques inéditos (*zero-day attacks*), são apresentadas e a implementação dos algoritmos nas arquiteturas de detecção é avaliada.

Para avaliar os mecanismos de defesa contra ataques de rede, o primeiro desafio é obter um conjunto de dados adequado para as avaliações. A disponibilidade de conjuntos de dados na literatura é limitada, pois há a preocupação com a privacidade e o receio de vazamento de informações confidenciais contidas na carga útil de pacotes [Heidemann and Papdopoulos 2009]. Um dos principais conjuntos de dados disponíveis é o **DARPA** [Lippmann et al. 2000a], composto por tráfego TCP/IP em formato cru (*raw data*) e dados do sistema operacional UNIX de uma rede simulada, obtidos ao longo de sete semanas de coleta totalizando 9,87 GB de dados. Como o DARPA 98 consiste em arquivos brutos, é preciso extrair as características desses arquivos para usá-los em algoritmos de aprendizado de máquina. Uma maior quantidade de tráfego de fundo e diferentes tipos de ataques foram adicionados para construir o DARPA 99. As primeiras duas semanas foram livres de ataque, portanto, é adequado para treinar algoritmos de detecção de anomalia. Nas cinco semanas seguintes vários ataques simulados foram usados contra a base.Novos ataques foram inseridos no DARPA 99 [Haines et al. 2001], principalmente ataques em diferentes sistemas operacionais como SunOS, Solaris, Linux, e Windows NT. A maioria das pesquisas utiliza uma mistura dos os dois conjuntos de dados referindo-se ao conjunto de dados do DARPA. O conjunto de dados **KDD99**, por sua vez, foi criado a partir dos arquivos do conjunto DARPA 98 para uma competição de detecção de intrusão no ano de 1999 [Lee et al. 1999], e é composto de amostras definidas por 41 características e uma classe de saída. O conjunto de dados possui duas semanas de ataques. As classes são divididas em cinco categorias que contêm 24 tipos de ataques para o treinamento e mais 14 tipos de ataques no conjunto de treino, totalizando 38 ataques. O conjunto de treinamento possui 494.021 fluxos e o de teste 311.029 fluxos. As classes incluem Negação de Serviço (*Denial of Service* - DoS), sondagem (*Probe*), *Root2Local* (R2L), *User2Root* (U2R) e normal. Um dos problemas do KDD99 é o desbalanceamento. Aproximadamente 80% das amostras são consideradas ataques, o que difere largamente da realidade. O conjunto contém poucos tipos de ataques U2R e R2L e muitos destes ataques são duplicados. Ao duplicar amostras, os classificadores se tornam tendenciosos a ataques de negação de serviço (DoS) e ao tráfego normal, os quais são os mais abundantes no KDD99. O **NSL-KDD** é uma modificação do conjunto original KDD-99 e possui as mesmas 41 características e as mesmas cinco categorias que o KDD 99 [Tavallaee et al. 2009b]. As melhoras do NSL-KDD em relação ao KDD 99 são a eliminação de amostras redundantes e duplicadas, para evitar uma classificação tendenciosa e o sobreajuste (*overfitting*), e um melhor balanceamento entre classes para evitar a escolha aleatória. Apesar da redução em tamanho, o NSL-KDD mantém as proporções entre as classes como no KDD 99, sendo 125.973 amostras de treino e 22.544 de teste. No entanto, o DARPA, o KDD, assim como o NLS-KDD são criticados, pois os seus tráfegos são sintéticos e, portanto, não representam fielmente cenários reais de rede de computadores [Tavallaee et al. 2009a]. Existem dados redundantes, que afetam o desempenho dos algoritmos de classificação de dados. Outra crítica importante é que esses conjuntos de dados estão

desatualizados, uma vez que foram simulados há mais de 15 anos [Sommer and Paxson 2010] e muitas aplicações, bem como ataques, surgiram desde então. Desde a criação do KDD99, outros conjuntos de dados foram publicados e disponibilizados introduzindo vantagens e desvantagens. Dessa forma, não há um conjunto de dados que atenda a todos os casos e a escolha de qual usar depende do cenário e da aplicação. Alguns exemplos de conjunto de dados frequentemente encontrados na literatura são o **UNB ISCX IDS 2012** [Shiravi et al. 2012], tráfego da simulação de um cenário real, o **CTU-13** [Garcia et al. 2014], tráfego simulado de uma *botnet*, o **CAIDA DDoS**[14], tráfego de um ataque DDoS real, o **MAWI** [Fontugne et al. 2010], tráfego real de um *backbone* entre EUA e Japão, o **Kyoto**, tráfego real de *honeypots* e o **LBNL**[15] tráfego real de roteadores de borda da LBNL.

O Grupo de Teleinformática e Automação da Universidade Federal do Rio de Janeiro (**GTA/UFRJ**) elaborou um conjunto de dados para testes e avaliações de desempenho [Lobato et al. 2016]. O conjunto de dados corresponde à captura de pacotes de tráfego reais do laboratório e contém comportamentos legítimos de usuários e ameaças de rede reais. Os ataques reais foram realizados de forma controlada pela ferramenta de teste de penetração *Kali Linux*, com a base de ataques mais recente até sua data de criação, contendo um total de sete tipos de ataques de negação de serviço e nove tipos de varreduras de porta. A principal vantagem desse conjunto de dados é que os ataques são mais atuais em relação a outros conjuntos de dados disponíveis na literatura. Após a captura, os pacotes são agrupados no formato de fluxos, em que o fluxo é definido como uma sequência de pacotes com os mesmos endereços IP de origem e destino dentro de uma janela de tempo. Então, foram extraídos dados de cabeçalhos TCP/IP de cada pacote e abstraídos em 26 características de fluxo, como as taxas de pacotes TCP, UDP e ICMP, a quantidade de pacotes com portas de origem e de destino, a quantidade de pacotes com cada sinal (*flag*) TCP, entre outras. Uma análise detalhada das características e ataques pode ser encontrada em [Andreoni Lopez et al. 2017a]. O conjunto de dados tem mais de 95 GB em pacotes capturados, resultando em 214.200 fluxos de tráfego malicioso e normal. A classe normal representa aproximadamente 70% do conjunto de dados, enquanto a classe de negação de serviço 10% do conjunto de dados e a classe de varredura da porta 20% das amostras.

O **NetOp** é um conjunto de dados reais de uma operadora do Brasil [Andreoni Lopez et al. 2017c, Andreoni Lopez et al. 2017d], usado na avaliação de desempenho de arquiteturas propostas por Andreoni *et al.*. O conjunto de dados possui tráfego anonimizado de acesso de 373 usuários de banda larga da Zona Sul da cidade do Rio de Janeiro. Os dados foram obtidos durante 1 semana de coleta de dados ininterrupta, de 24 de fevereiro a 4 de março de 2017. O conjunto de dados contém 5 TB de dados reais em formato bruto (*raw data*) que representam um total de 5,3 milhões de fluxos, definidos pela quíntupla IPs e portas de origem e de destino e protocolo de transporte. Os dados foram previamente rotulados pelo sistema de detecção de intrusão (*Intrusion Detection System* - IDS) Suricata para validar os fluxos entre tráfego legítimo ou malicioso. Os pacotes são resumidos em um conjunto de dados de 46 características de fluxos associadas

[14]The Cooperative Analysis for Internet Data Analysis, http://www.caida.org

[15]Lawrence Berkeley National Laboratory - LBNL/ICSI Enterprise Tracing Project, http://www.icir.org/enterprise-tracing/download.html

à uma classe de alarme do IDS ou à classe de tráfego legítimo [Andreoni Lopez et al. 2017d].

A Tabela 3.3 resume as principais características dos conjuntos de dados utilizar na literatura de detecção de intrusão.

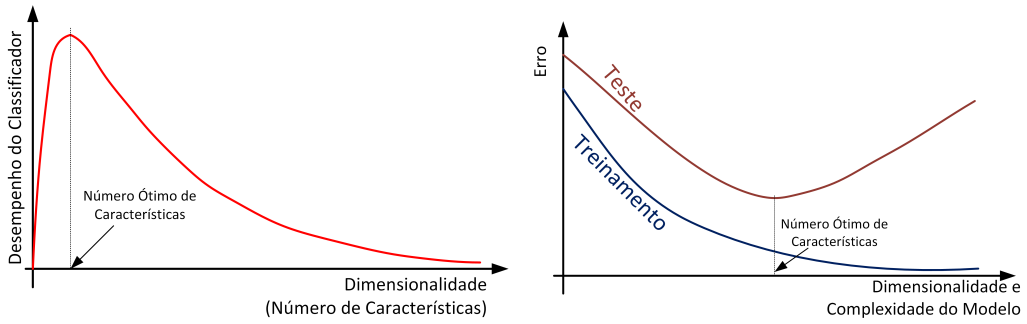
Tabela 3.3. Resumo dos principais conjuntos de dados disponíveis na literatura (Ca: características; Fl: fluxos).

Conjunto	Formato	Tamanho	Ataques	Tipo	Ano
DARPA 98/99	pcap	9.87 GB	80.1 %	Sintético	1998/99
KDD99	csv (41 Ca)	805.050 Fl	80.1 %	Sintético	1998
NLS-KDD	csv (41 Ca)	148.517 Fl	80.5 %	Sintético	1998
LBNL	pcap	11 GB	-	Real	2005
Kyoto	txt (24 Ca)	14 GB	100%	Real	2006
CAIDA DDoS	pcap	21 GB	100%	Real	2007
ISCX IDS	pcap	84.42 GB	2.8 %	Sintético	2012
CTU-13	pcap	697 GB	11.69 %	Sintético	2014
MAWI	pcap	$\sim$ 2 GB/ano	-	Real	2001–18
GTA/UFRJ	csv (26 Ca)	95 GB	30 %	Real/Controlado	2016
NetOp	csv (46 Ca)	5.320.955 Fl	-	Real	2017

3.4.1. Seleção de Características e Redução da Dimensionalidade

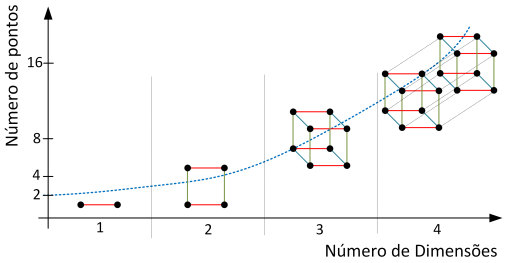
Uma informação pode ser representada ou classificada por suas características ou atributos. A quantidade de características ou atributos usados para representar uma informação varia bastante. Um problema relevante é que o aumento do número de atributos nem sempre melhora a acurácia da classificação da informação. Este problema é conhecido como "a maldição da dimensionalidade" [Zhai et al. 2014] que afirma existir um número ótimo de características que podem ser selecionados em relação ao tamanho da amostra para maximizar o desempenho do classificador. A Figura 3.10(a) mostra que a medida que a dimensionalidade aumenta, o desempenho do classificador aumenta até que o número ótimo de características seja atingido e, a partir desse valor ótimo, o desempenho cai. Logo, a partir desse valor ótimo do número de características, aumentar a dimensionalidade sem aumentar o número de amostras de treinamento resulta em uma diminuição no desempenho do classificador. A Figura 3.10(b) mostra a degradação do desempenho dos algoritmos. O incremento na dimensionalidade é diretamente proporcional à complexidade do modelo, obtendo uma taxa de erro baixa durante a fase de treinamento. No entanto, esses modelos apresentam o problema de sobreajuste (*overfitting*) ao treinamento, fazendo com que o modelo apresente um desempenho insatisfatório na etapa de teste. A Figura 3.10(c) mostra que o número de pontos aumenta exponencialmente com o aumento da dimensionalidade. Nos espaços com muitas dimensões, os pontos tornam-se esparsos e pouco similares, com pontos muito distantes uns dos outros e aproximadamente equidistantes, o que leva um classificador a cometer mais erros. Existem também outros problemas ao utilizar algoritmos de aprendizado de máquina com um número alto de dimensões, pois algumas métricas tradicionais de distância, como a distância Euclidiana, deixam de ter sentido em dimensões altas, sendo necessária a utilização de outros tipos de métricas, como a distância cosseno, que possuem maior custo computacional.

Ainda, a maioria das técnicas de aprendizado de máquina são ineficazes no tratamento de dados dimensionais elevados, incorrendo em problemas de sobreajuste (*overfitting*) durante a fase de classificação. Portanto, é comum reduzir o número de características ou a dimensionalidade antes da aplicação de um algoritmo de aprendizado de máquina.



(a) Desempenho do classificador com o aumento da dimensionalidade.

(b) Efeito de sobreajuste aos dados do treinamento, o que reduz o desempenho no teste.



(c) Crescimento exponencial do número de pontos com o aumento da dimensionalidade.

Figura 3.10. O problema da "maldição da dimensionalidade" afirma existir um número ótimo de características que podem ser selecionadas em relação ao tamanho da amostra para maximizar o desempenho do classificador.

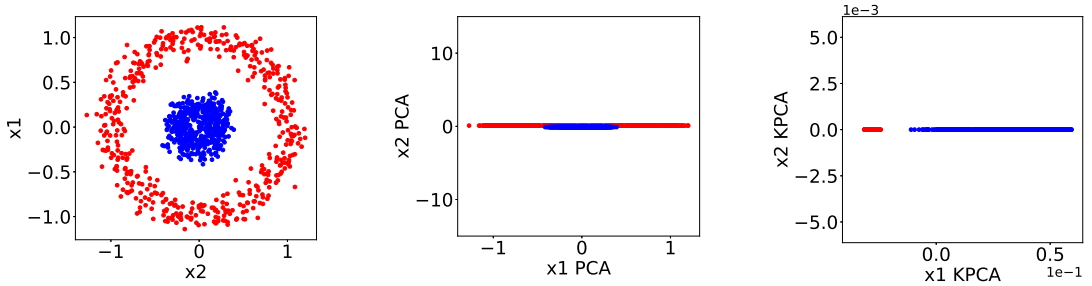
A seleção de características ou a redução de dimensionalidade são duas técnicas bastante usadas para se obter mais desempenho no aprendizado. A seleção de características mantém os atributos mais relevantes do conjunto de dados original, criando um subconjunto das características. Por outro lado, a redução da dimensionalidade aproveita a redundância dos dados de entrada, calculando um conjunto menor de novas dimensões, ou atributos sintéticos. Os novos atributos sintéticos são uma combinação linear ou não linear dos atributos de entrada. A ideia principal dos métodos é remover toda informação redundante, mantendo unicamente a informação necessária e suficiente para representar a informação original. O objetivo tanto da seleção de características como da redução da dimensionalidade é produzir um conjunto mínimo de características para que o novo subconjunto mantenha um desempenho o mais semelhante possível ou superior ao do conjunto gerador. Portanto, a seleção de características e a redução da dimensionalidade melhoram o desempenho da classificação, provendo classificadores mais rápidos e economicamente viáveis. A seleção de características possui uma propriedade adicional, pois permite um melhor entendimento do processo que gera o dado. Na redução de dimensionalidade as características selecionadas são sintéticas e compostas pela combinação das características originais, o que dificulta o entendimento do processo.

A **redução da dimensionalidade** também pode ser vista como o processo de derivar um conjunto de graus de liberdade, que pode ser usado para reproduzir a maior parte da variabilidade de um conjunto de dados [Van Der Maaten et al. 2009]. Idealmente, a representação reduzida deve ter uma dimensionalidade que corresponde à dimensionalidade intrínseca dos dados. A dimensionalidade intrínseca dos dados é o número mínimo de parâmetros necessários para atender as propriedades observadas dos dados. Geralmente, na redução de dimensionalidade é criado um novo espaço de características através de algum tipo de transformação do espaço de características original. Na redução da dimensionalidade, dada a variável aleatória de p-dimensões x = $(x_1, x_2, \ldots, x_p)$, é possível encontrar uma representação dimensional inferior, s = $(s_1, s_2, \ldots, s_k)$ com $k \leq p$.

Muitos algoritmos com abordagens diferentes foram desenvolvidos para reduzir a dimensionalidade que podem ser classificadas em dois grupos: os lineares e os não-lineares. A redução linear de dimensionalidade é uma projeção linear, na qual os dados p-dimensionais são reduzidos em dados k-dimensionais usando k combinações lineares das p dimensões. Dois importantes exemplos de algoritmos de redução de dimensão lineares são a análise de componentes principais (*Principal Component Analysis* - PCA) e a análise de componentes independentes (*Independent Component Analysis* - ICA). O objetivo do PCA é encontrar uma transformação linear ortogonal que maximize a variância das características. O primeiro vetor da base PCA, a componente principal, é o que melhor descreve a variabilidade de seus dados. O segundo vetor é a segunda melhor descrição e deve ser ortogonal ao primeiro e assim por diante em ordem de importância. Por outro lado, o objetivo da ICA é encontrar uma transformação linear na qual os vetores base são estatisticamente independentes e não gaussianos, ou seja, a informação mútua entre duas variáveis do novo espaço vetorial é igual a zero. Ao contrário do PCA, os vetores de base na ICA não são nem ortogonais nem classificados em ordem, todos os vetores são igualmente importantes. O PCA é utilizado quando se quer encontrar uma representação reduzida dos dados. Por outro lado, o ICA é utilizado para se obter a extração de características, selecionando as características que melhor se adequam à aplicação.

Nos casos em que os dados de alta dimensionalidade apresentam uma natureza não-linear, os algoritmos lineares não apresentam bom desempenho. Isto significa que a relação entre as classes não é descrita em um subespaço linear, como mostrado na Figura 3.11(a). Para estes casos, é possível utilizar a técnica do PCA com funções de *kernels*. A função *kernel* transforma os vetores de entrada do espaço original a um espaço dimensional maior, no qual o problema se torna linearmente separável [Schölkopf et al. 1999]. A Figura 3.11 mostra uma comparação dos métodos de redução linear e não linear. O conjunto de dados original, mostrado na Figura 3.11(a), é um caso de dois círculos concêntricos, cada círculo é uma classe. O objetivo é reduzir a 2 dimensões ($\mathbb{R}^2$) para espaço de 1 dimensão ($\mathbb{R}^1$). Depois de aplicar uma redução linear, Figura 3.11(b), as componentes principais não conseguem um subespaço em que as classes estejam separadas linearmente no espaço ($\mathbb{R}^1$). Isso ocorre porque os dois círculos concêntricos são duas classes não-linearmente separáveis. Depois de aplicar um método não-linear, como um *Gaussian Kernel* PCA, mostrado na Figura 3.11(c), o método consegue criar um subespaço onde as classes se separaram corretamente.

Existem duas abordagens para a separação de classes em dados que não podem ser separados de forma linear, como mostra a Figura 3.12(a). A primeira consiste em



(a) Conjunto de dados original de duas classes.

(b) Resultado da análise da PCA linear.

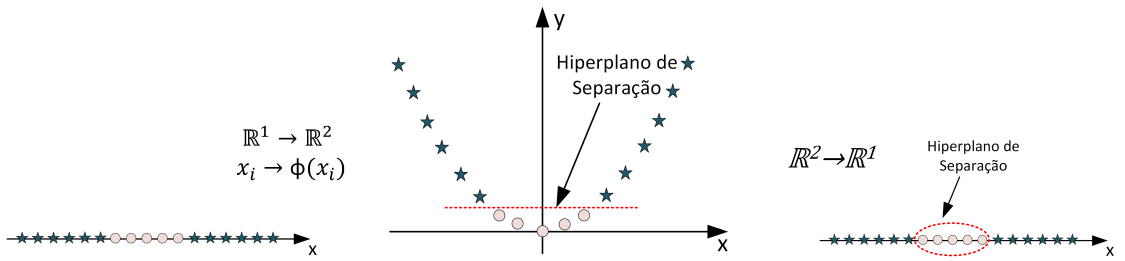
(c) Resultado da PCA com função de *kernel*, não-linear.

Figura 3.11. Exemplo do "artifício do *kernel*" para o PCA: a) duas classes com separação não-linear, b) dificuldade da separação de classes aplicando PCA linear e c) resultado da separação de classes ao aplicar PCA com função de *kernel* não-linear.

mapear os dados em um espaço de maiores dimensões, no qual as classes possam ser separadas linearmente, ou seja, por retas, como mostra a Figura 3.12(b). No exemplo da Figura 3.12 é um exemplo binário, no qual no plano de $\mathbb{R}^1$ não existe nenhum hiperplano linear que seja capaz de separar as duas classes. Se o problema é levado a um plano com dimensionalidade superior, $\mathbb{R}^1 \rightarrow \mathbb{R}^2$, é possível encontrar um hiperplano, traços na Figura 3.12(b), que separe as classes. O conjunto de dados pode ser mapeado em um espaço de dimensão superior ao original, $\mathbb{R}^1 \rightarrow \mathbb{R}^2$, e com isso possibilitar um hiperplano que separe as classes, como mostra a Figura 3.12(b). Esse hiperplano, quando é trazido para uma dimensão menor, $\mathbb{R}^2 \rightarrow \mathbb{R}^1$, corresponde a uma superfície não-linear como mostrado na Figura 3.12(c). No entanto, ao realizar o mapeamento para dimensões maiores se incorre na "maldição da dimensionalidade" explicada anteriormente, o que gera um alto custo computacional. Para resolver esse problema é aplicado o "artifício do *kernel*". Uma função *Kernel* é uma função de similaridade que corresponde ao produto escalar em um espaço vetorial expandido. A ideia é encontrar uma função não-linear na qual não seja necessário fazer o mapeamento de dimensões e que a computação seja independentemente do número de características. Se existe uma transformação não-linear $\Phi(x)$ do espaço de características D-dimensional original para um espaço de características M-dimensional, onde geralmente $M \gg D$. Então, cada ponto de dados x_i é projetado para um ponto $\Phi(x_i)$. Se a matriz $K()$ que contém os produtos escalares entre todos os pares de pontos de dados, agora é calculada como $K(x_i, x_j) = \Phi(x_i)^T \Phi(x_j)$. Os *kernels* mais utilizados são o núcleo polinômico, gaussiano e tangente (tangente hiperbólica). Se um *kernel* usado é linear, podemos ver o PCA padrão como um caso do *Kernel* PCA.

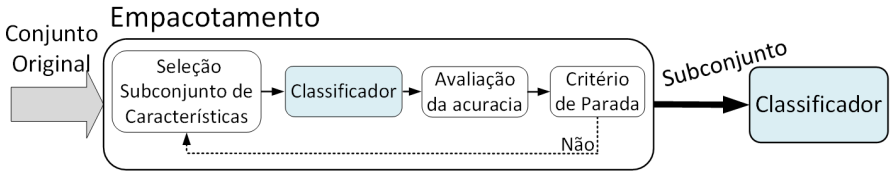
A **seleção de características** produz um subconjunto das características originais, que são as melhores representantes dos dados, portanto não há perda de significado, ao contrário da redução de dimensionalidade. As técnicas de seleção de características podem ser divididas em três tipos de algoritmos [Mladenić 2006]: empacotamento (*wrappers*), filtragem e embarcados.

Os métodos de empacotamento, como mostra a Figura 3.13(a), usam algoritmos de aprendizado de máquina, como máquina de vetores de suporte (*Support Vector Machine* - SVM), árvore de decisão, entre outros, para medir a qualidade dos subconjuntos

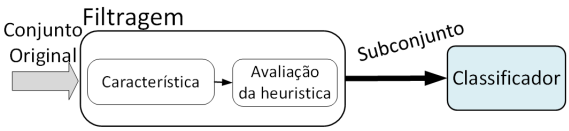


(a) Exemplo de classes binárias. (b) Mapeamento para uma dimensão maior. (c) Duas classes separadas através de um hiperplano não-linear.

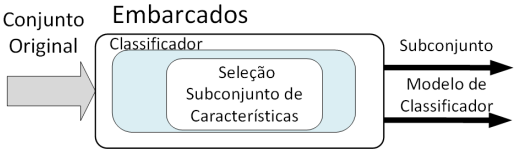
Figura 3.12. Estratégias para separação de classes de distribuição de dados não-linear: a) conjunto de dados original de duas classes com distribuição não-linear; b) mapeamento dos dados com acréscimo de dimensão que possibilite a separação com algoritmos lineares, como o PCA e c) uso de hiperplano não linear, como no algoritmo *Kernel* não-linear do PCA.



(a) Métodos de Empacotamento.



(b) Métodos de Filtragem.



(c) Métodos Embarcados.

Figura 3.13. Algoritmos de Seleção de Características: a) métodos de empacotamento que utilizam um classificador para avaliar os subgrupos de características, b) métodos de filtragem que utilizam heurísticas para avaliar uma característica ou um subconjunto delas e c) métodos embarcados que utilizam um algoritmo especifico de classificação para fazer a seleção naturalmente.

de características sem incorporar conhecimento sobre a estrutura específica da função de classificação. Assim, o método avaliará subconjuntos com base na acurácia do classificador. A avaliação é repetida para cada subconjunto até encontrar um conjunto que apresente um bom desempenho. Os métodos de empacotamento tendem ser mais acurados que os métodos de filtragem, porém apresentam um maior custo computacional.

Para diminuir o alto custo computacional da avaliação os diversos subconjuntos de características baseada nos classificadores foram implementados os métodos de filtragem. Os métodos de filtragem são chamados de métodos de ciclo aberto porque não possuem interação com o classificador. Ao invés de classificadores, o método usa heurísticas para avaliar a relevância da característica no conjunto de dados [Chandrashekar and Sahin 2014, Andreoni Lopez et al. 2017a]. Como seu nome indica, a característica que não supera o critério da heurística é filtrada. O ganho de informação, a distância, a consistência e a semelhança entre características, assim como medidas estatísticas, são algumas

das heurísticas mais utilizadas na avaliação da filtragem. Já que o processo de seleção é feito em uma etapa precedente à classificação, só depois que as melhores características são encontradas, os algoritmos de classificação podem usá-las. Esse método é rápido para selecionar característica, mas, por não ter interação com o classificador, o subconjunto de características selecionado pode não ter boa acurácia. Um dos algoritmos de filtragem mais populares é o Relief, no qual a pontuação das características é calculada como a diferença entre a distância do exemplo mais próximo da mesma classe e o exemplo mais próximo da classe diferente. A principal desvantagem deste método é que as classes dos dados devem ser rotuladas antecipadamente. O Relief é limitado a problemas com apenas duas classes, mas o ReliefF [Robnik-Šikonja and Kononenko 2003] é uma melhoria do método de Relief que trata de classes múltiplas usando a técnica dos k-vizinhos mais próximos. O ReliefF é um método supervisionado, no qual as rotulagens das classes devem ser conhecidas antes da aplicação do método. Em aplicações como o monitoramento de rede e a detecção de ameaças os fluxos de rede chegam aos classificadores sem rótulo. Logo, devem ser aplicados algoritmos não supervisionados. Andreoni *et al.* resolvem esse problema apresentando um algoritmo que executa uma seleção de características não supervisionada utilizando um método de filtragem [Andreoni Lopez et al. 2017a]. A correlação e a variança entre as características é usada para medir a quantidade de informação que cada características representa em relação às outras independentemente da classe. Assim, o algoritmo rapidamente seleciona as características que apresentam maior informação.

Finalmente, os métodos embarcados são uma sub-classe dos métodos de empacotamento. Neste caso, é obtido o subconjunto de características assim como o modelo no qual foram selecionadas. Os métodos embarcados executam o processo de seleção de características durante a fase de aprendizagem de um classificador. Esses métodos selecionam características com base em critérios gerados durante o processo de aprendizagem de um classificador específico. Em contraste com os métodos de empacotamento, não separam a aprendizagem da seleção de características. Os métodos embarcados possuem um comportamento semelhante aos métodos de empacotamento, usando a acurácia de um classificador para avaliar a relevância da característica. No entanto, os métodos embarcados precisam modificar o algoritmo de classificação no processo de aprendizado.

Ao aplicar a redução de dimensionalidade e a redução de características a dados em fluxo, uma das abordagens é aplicar transformações nos dados quando chegam ao sistema de classificação. Assim, calcula-se um modelo *a priori*, com base em dados armazenados, e aplica-se aos dados que chegam em fluxo. O modelo pode ser tanto calculado para redução de dimensionalidade [Lobato et al. 2018], como para seleção de características [Andreoni Lopez et al. 2017a]. A aplicação da seleção de características sobre dados em fluxo usa técnicas como filtragem e rascunho (*sketching*) dos dados [Gaber et al. 2005, Gama and Rodrigues 2007].

3.4.2. Métricas de Avaliação de Desempenho de Classificadores

As principais métricas utilizadas para avaliar a qualidade de um classificador de detecção de intrusão são:

- **matriz de confusão** é forma tradicional de apresentar o desempenho de um mé-

todo de classificação, que informa a quantidade de Verdadeiros Positivos (VP), Falsos Positivos (FP), Verdadeiros Negativos (VN) e Falsos Negativos (FN) para cada classe. A partir destas informações pode-se derivar outras métricas que facilitam a comparação de diferentes métodos;

- **acurácia** de um método é a razão do total de amostras classificadas corretamente (VP + VN) dividido pelo número total de amostras (P+N). Para conjunto de dados não-balanceados, em que a proporção entre classes não é a mesma, a acurácia não é um indicador confiável de bom desempenho;

- **sensibilidade**, também conhecida como **revocação** (*recall*) ou **taxa de verdadeiros positivos**, é a razão da quantidade de amostras classificadas corretamente para a classe positiva (VP), dividido pelo total de amostras que pertencem a esta classe (VP + FN). O análogo para a classe negativa é chamado de **especificidade** ou **taxa de verdadeiros negativos**;

- **precisão** é a razão entre o número de amostras classificadas corretamente para a classe positiva (VP), dividido pelo total de amostras classificadas para esta classe (VP + FP). Há um compromisso natural entre a sensibilidade e a precisão de um algoritmo e equilibrar uma boa precisão com uma boa sensibilidade é uma tarefa difícil;

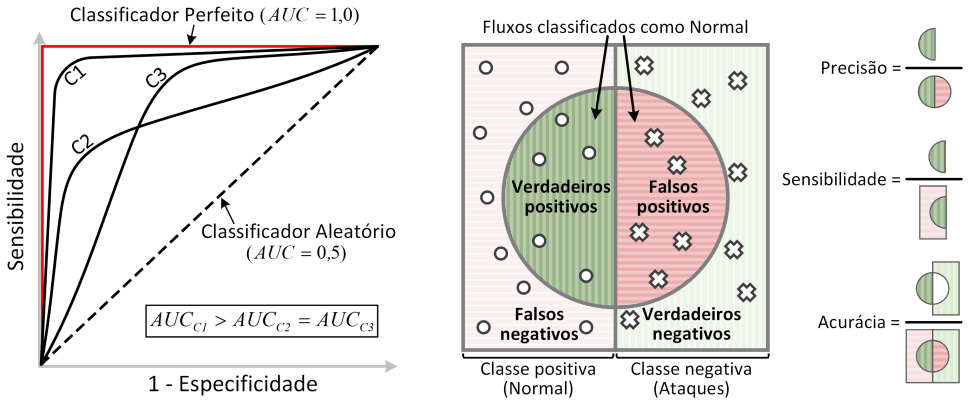
- **F1 score** é a média harmônica da precisão com a sensibilidade, expressa por

$$F1\ score = \frac{2}{\frac{1}{precisão} + \frac{1}{sensibilidade}}. \tag{1}$$

  Esta métrica busca o compromisso da precisão e com a sensibilidade. Geralmente, quanto maior o F1 score, melhor a classificação.

- **área abaixo da curva ROC** (*Area Under the ROC Curve* - AUC) é uma métrica eficaz para quantificar o desempenho da classificação binária em conjuntos de dados não-balanceados. A curva de Característica de Operação do Receptor (*Receiver Operating Characteristic* - ROC) é um gráfico que representa o compromisso entre a taxa de verdadeiros positivos (sensibilidade) e a taxa de falsos positivos ($1-$ especificidade). A Figura 3.14(a) exemplifica a comparação das áreas abaixo da curva ROC de diferentes classificadores. Quanto maior a área AUC, melhor é o classificador.

A Figura 3.14(b) ilustra a relação das métricas de precisão, sensibilidade e acurácia com as estatísticas fundamentais da classificação binária. A classe positiva representa as amostras de tráfego normal e a classe negativa as ameaças. Além das métricas tradicionais, outras métricas podem ser utilizadas para complementar a avaliação e comparação dos métodos de classificação, como as medidas de tempo de treinamento e de tempo de predição, *Log Loss*, Coeficiente de Correlação de Matthews, Coeficiente de Cohen's Kappa entre outros [Lippmann et al. 2000b, Gaffney and Ulvila 2001, Gu et al. 2006, Cárdenas and Baras 2006]. É importante notar que o desempenho de um método de



(a) Curvas ROC de classificadores. (b) Representação das métricas de desempenho.

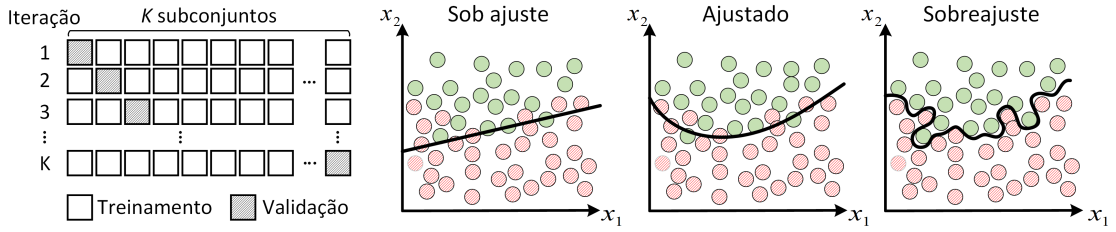
Figura 3.14. a) Curvas ROC de classificadores e comparação da área abaixo da curva ROC (AUC). b) Representação da precisão, sensibilidade e acurácia em um problema de classificação binária.

aprendizado de máquina está condicionado ao conjunto de características selecionadas, às técnicas de redução, seleção ou enriquecimento de características utilizada, à qualidade do conjunto de dados a ser classificado e à aplicabilidade do modelo ao problema.

O desempenho de um modelo representa o quão boa está a aproximação de sua função objetivo para generalizar a realidade. Dessa forma, o sobreajuste em aprendizado de máquina refere-se a uma função objetivo que modela demasiadamente bem os dados de treinamento, tão bem que prejudica a capacidade de generalização do modelo. Assim, o sobreajuste ocorre quando o modelo aprende com as flutuações e o ruído do conjunto de treinamento, o que piora o desempenho do modelo para novos conjuntos de dados. As principais causas do sobreajuste são conjuntos insuficientes de dados, alta dimensionalidade ou modelos com muitos graus de liberdade. O sobreajuste dos dados é mais comum em algoritmos não-paramétricos e não-lineares, em que a função objetivo tem mais flexibilidade de aprendizado. Por isso, esses algoritmos costumam ter mecanismos que procuram limitar e restringir o nível de detalhes do aprendizado.

Duas técnicas bastante empregadas para evitar o sobreajuste são a validação cruzada e o balanceamento. A validação cruzada *K-fold*, representada na Figura 3.15(a), realiza K iterações de treinamento nas frações dos dados e, a cada iteração, nas $K-1$ frações dos dados restantes, realiza o teste de forma mutuamente exclusiva, sendo $K = 10$ o valor comumente usado. O balanceamento também evita o sobreajuste e é muito empregado em conjuntos de dados em que as quantidades de amostras de cada classe são desproporcionais. Nesse caso, as classes minoritárias não apresentam amostras suficientes para que o classificador aprenda adequadamente sobre elas de forma que o treinamento e, consequentemente, as métricas de avaliação de desempenho, se tornam tendenciosos para a classe dominante. No geral, as técnicas de balanceamento se resumem em replicar amostras da classe minoritária ou eliminar amostras da classe majoritária. Essas duas ações podem ser feitas por algoritmos específicos [Batista et al. 2004] ou de forma aleatória, que possui o risco de eliminar amostras importantes ou gerar mais sobreajuste nos dados. Por outro lado, o sob ajuste em aprendizado de máquina refere-se a um modelo que nem

modela bem o conjunto de treino nem generaliza bem para novos conjuntos de dados. Isso ocorre quando o classificador não é adequado para o problema e apresenta baixo desempenho na classificação. A solução nesse caso é a escolha de um algoritmo diferente de aprendizado. A Figura 3.10(b) ilustra o problema do sob ajuste e sobreajuste de um modelo de classificador.



(a) Técnica *K-fold*. (b) Sob ajuste e sobreajuste do modelo de classificação.

Figura 3.15. a) Técnica de validação cruzada *K-fold*, que particiona o conjunto de dados em K partições mutuamente exclusivas para obter uma medida mais confiável do desempenho do classificador. b) Ilustração do sob ajuste e sobreajuste de um modelo de classificação.

3.4.3. Detecção de Ameaças em Tempo Real a partir de Treinamento Prévio

A arquitetura kappa[16], mostrada na Figura 3.16, possui um caminho direto da coleta, passando pelo processamento do fluxo e finalizando na visualização dos dados em tempo real. A kappa é voltada para aplicativos que demandem baixa latência sem a necessidade de armazenamento dos dados.

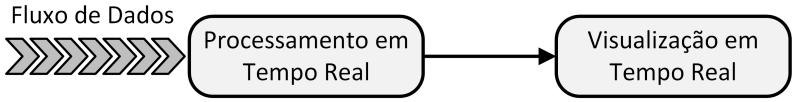


Figura 3.16. A arquitetura kappa provê um caminho direto com a coleta, o processamento do fluxo e a visualização dos dados em tempo real.

A arquitetura lambda[17] combina o processamento de fluxo em tempo real com o processamento em lotes, para analisar grandes massas de dados (*Big Data*) [Marz and Warren 2013]. O processamento em lotes permite o uso de métodos acurados de classificação que necessitam de parâmetros calculados em tempo diferenciado. Assim, esta arquitetura possui um caminho em linha (*online*) de processamento de dados em movimento para obter resultados aproximados a baixa latência e um caminho em tempo diferenciado (*offline*) para processamento em lotes com resultados tardios, porém precisos.

Conforme mostrado na Figura 3.17, a arquitetura lambda possui três camadas: a camada de processamento de fluxo; a camada de processamento em lotes e a camada de serviço. A camada de processamento de fluxo lida com os dados recebidos em tempo real, objetivando a baixa latência no processamento. A camada de processamento em lotes possui uma visão global de uma grande quantidade de dados históricos e os analisa de

[16]Arquitetura kappa foi proposta pelo Jay Kreps co-criador do apache Kafka https://www.oreilly.com/ideas/questioning-the-lambda-architecture.

[17]A arquitetura lambda foi proposta por Nathan Marz criador da plataforma Storm.

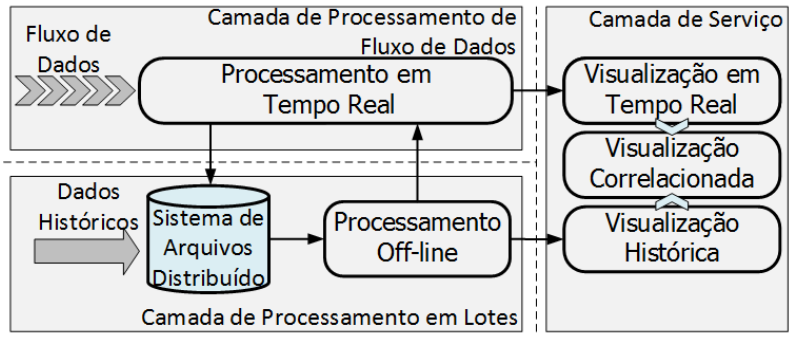


Figura 3.17. A arquitetura lambda que combina as três camadas de processamento de fluxo, de processamento em lotes e de serviço.

maneira distribuída através de técnicas como MapReduce. Finalmente, a camada de serviço combina a informação obtida das duas camadas anteriores para fornecer uma saída composta por dados analíticos para o usuário. Portanto, o objetivo da arquitetura lambda é fornecer análises precisas com base em dados históricos e, ao mesmo tempo, obter resultados em tempo real. Dessa forma, a arquitetura lambda combina o processamento em lotes tradicional em um banco de dados históricos com a análise de processamento de fluxo em tempo real. Isso permite implementar uma gama de aplicações em que uma resposta imediata seja necessária no menor tempo possível assim que novos dados chegam. Uma destas aplicações é a detecção de ameaças em tempo real a partir de modelos pré-treinados dos algoritmos de aprendizado de máquina implantados na camada de processamento de fluxo. Há um período de treinamento do algoritmo, antes de começar a classificar, para ele "aprender" o que é tráfego normal e tráfego malicioso. Esse treinamento, em outras palavras, significa encontrar os melhores parâmetros, no caso dos algoritmos paramétricos, ou encontrar o melhor modelo de classificação no caso de algoritmos não-paramétricos, a partir de dados de exemplo. Na arquitetura lambda, este treinamento é feito em tempo diferenciado na camada de processamento em lotes e o modelo ou parâmetros calculados são atualizados no processamento em fluxo eventualmente.

A Figura 3.18 ilustra como a arquitetura lambda pode ser aplicada no contexto de detecção de ameaças em linha (*online*). A captura em linha (*online*) de dados consiste em analisadores de tráfego e arquivos de registros (*logs*) que geram dados em linha a partir de múltiplas fontes e os enviam para um *buffer* de dados do processamento. Para poupar processamento, a informação gerada pode ser transformada no vetor de características da instância ainda na fonte. A camada de processamento de fluxo consiste em um *buffer* de dados e o módulo de detecção em linha. O *buffer* de dados é um mecanismo de segurança que garante que as instâncias não serão perdidas caso a taxa de chegada de informação seja maior que a capacidade de processamento. Já a detecção online consiste em um arcabouço de processamento de fluxo, que carrega um modelo de classificação previamente estabelecido e o aplica para cada amostra recebida. A camada de processamento em lotes, por sua vez, contém um banco de dados distribuído que armazena os conjunto de dados rotulados que serão usados para treinar o modelo de classificação. O treinamento neste caso é feito em tempo diferenciado (*offline*), e uma vez definido o modelo, seus parâmetros são atualizados. Como os dados que chegam em tempo real não possuem rótulos confiáveis, a atualização do modelo de classificação depende da adição de novos dados

rotulados à base histórica, ou de melhorias nos métodos de treinamento. Portanto, essa forma de detectar ameaças em linha (*online*) não é eficaz contra novos ataques, que ainda não constam da base histórica.

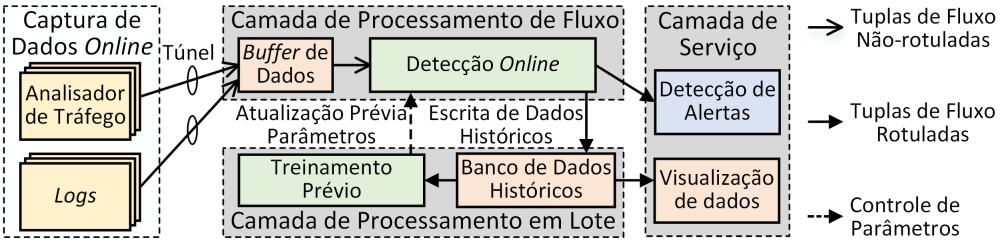


Figura 3.18. Aplicação da arquitetura lambda para a detecção em linha (*online*) de ameaças. O treinamento do classificador é feito em tempo diferenciado (*offline*), enquanto a classificação é em linha (*online*).

Uma grande vantagem desse tipo de arquitetura de classificação é a aplicação de modelos complexos aos dados em fluxo sem a adição do atraso da fase de treinamento. Essa arquitetura é particularmente eficiente para algoritmos que possuem um longo tempo de treinamento e que, uma vez construído o modelo, a predição da classe de saída é rápida. No contexto de detecção de intrusão na Internet, a velocidade em que os dados evoluem é alta e novas classes de ataques surgem a todo momento com a descoberta de vulnerabilidades, e ainda, o comportamento de tráfego benigno está em constante evolução com o surgimento de novas aplicações. Portanto, a eficácia desse tipo de arquitetura na detecção contra novas ameaças depende da frequente atualização dos parâmetros do modelo, o que não é feito de forma automática. A atualização pode ser feita de duas formas básicas, através do aperfeiçoamento dos algoritmos de aprendizado utilizados, como a alteração no métodos ou nas características utilizadas, ou através da adição de novos dados rotulados à base histórica para reforçar o treinamento.

O treinamento prévio do classificador é realizado em tempo diferenciado (*offline*) usando os algoritmos de aprendizado de máquina convencionais. Neste capítulo, o critério de seleção de quais algoritmos de aprendizado de máquina escolher para as aplicações de detecção de intrusão aqui descritas se apoia na lista dos algoritmos mais usados realizada por Buczak e Guven [Buczak and Guven 2015] e nos mais comumente encontrados na literatura de detecção de intrusão.

Os algoritmos de **árvore de decisão** (*Decision Tree* - DT) constroem uma árvore em que cada nó é responsável pelo teste de uma característica do conjunto de dados. Os valores de probabilidade de cada classe são armazenados naquele nó e servem como parâmetros para a tomada de decisão. Assim, os nós ramos representam condições baseadas no valor de uma das características de entrada e os nós folhas representam a classe final. A cada nova amostra desconhecida de entrada, o algoritmo percorre os ramos da árvore e avalia as respectivas características para estimar a probabilidade daquela amostra pertencer a uma determinada classe. Dessa forma, não é necessário percorrer todas as características da amostra para realizar a classificação, o que reduz o tempo de processamento na classificação.

As heurísticas de construção da árvore diferem quanto às diversas implementações do algoritmo existentes na literatura. Dentre as mais conhecidas encontram-se o

Iterative Dichotomiser 3 (ID3), o C4.5 e o *Classification And Regression Tree* (CART). A diferença principal entre as implementações são as métricas definidas para fazer a divisão entre os ramos e as técnicas para evitar sobreajuste. O algoritmo ID3 maximiza o ganho de informação e minimiza a entropia para escolher a ordem de divisão entre os ramos. O CART realiza divisões binárias utilizando a impureza de Gini e, nas suas primeiras versões, usava a redução da variância. O C4.5, por sua vez, é uma evolução do ID3 que permite tanto características categóricas quanto numéricas, além de resolver o problema de sobreajuste usando técnicas de poda.

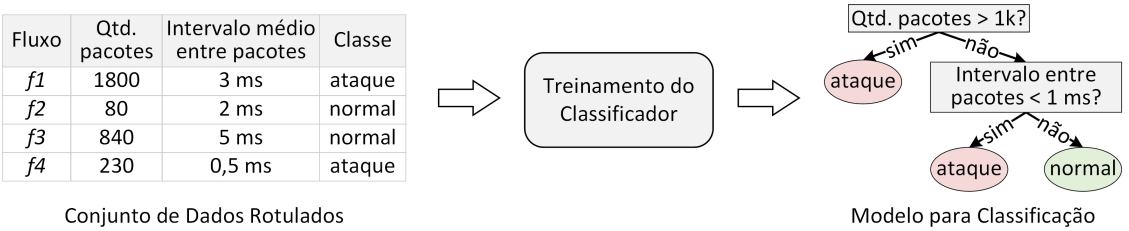


Figura 3.19. Exemplo do treinamento em tempo diferenciado (*offline*) de um classificador do tipo árvore de decisão para negação serviço por inundação. Neste classificador, a saída é um modelo de árvore de probabilidades do tipo *if-then-else*.

As árvores de decisão têm como vantagem a interpretação. Por serem um conjunto de regras do tipo "*if-then-else*", é possível interpretá-las e até exibi-las graficamente para facilitar o entendimento das regras de decisão. A Figura 3.19 ilustra um treinamento de um classificador de árvore de decisão que, por ter um tamanho reduzido, pode ser facilmente interpretado. Além disso, as árvores são capazes de lidar com os dados numéricos e categóricos e apresentam um bom desempenho com conjunto de dados de grande tamanho. Entretanto, é comum a ocorrência de sobreajuste (*overfitting*) no conjunto de dados, exigindo a utilização de técnicas externas ao algoritmo como a poda (*pruning*).

As **redes neurais artificiais** (*Artificial Neural Networks* - ANN) são famílias de algoritmos inspiradas no funcionamento do cérebro humano, em que cada neurônio realiza uma pequena parte do processamento e transmite o resultado para o próximo neurônio. A transmissão, ou disparo, para o próximo neurônio, só ocorre caso o resultado de uma função de ativação sobre suas entradas seja maior que determinado limiar. As ANN são divididas em camadas, em que cada camada transfere sua saída para a seguinte até que a última camada produza o resultado de saída. A saída de uma ANN representa um grau associado à cada classe, em que o maior grau entre as saídas define a classe da amostra. O processo de treinamento calcula os vetores de peso w_n, que determinam o peso de cada conexão de neurônio. No treinamento, existem espaços de amostra de entrada e saída e dos erros causados por cada parâmetro, que são minimizados através do algoritmo de propagação direta (*forward-propagation*). Para determinar a classe de uma amostra de entrada, cada camada da rede neural calcula as equações

$$z_{(i)} = w_{(i)} \cdot y_{(i-1)} + b_{(i)} \quad (2) \qquad y_{(i)} = \varphi(z_{(i)}) \quad (3) \qquad \varphi(z) = \frac{1}{1+e^{-z}}, \quad (4)$$

em que $y_{(i)}$ é o vetor que determina a saída da camada i, $w_{(i)}$ é o vetor de peso que leva a camada $i-1$ para a camada i, $b_{(i)}$ é o vetor de parâmetros de polarização (*bias*) da

camada i e $z_{(i)}$ a função de transferência da camada i. A função $\varphi(z)$ é uma função de ativação, sendo geralmente utilizada a função *Sigmoid*, representada na Equação 4. A função de ativação tem como objetivo capturar uma relação não-linear entre as entradas e convertê-las em uma saída mais útil. Para valores elevados de z, $\varphi(z)$ retorna 1 e para valores baixos $\varphi(z)$ retorna 0. Portanto, a camada de saída fornece o grau de associação de cada classe, entre 0 e 1.

O **perceptron**, também chamado de neurônio artificial, é a forma mais simples de uma rede neural do tipo direta (*feed-forward*), sem realimentação, em que a informação sempre se move no mesmo sentido e não há laços (*loops*). Já o **perceptron multicamadas** (*Multi-Layer Perceptron* - MLP) é uma rede neural também do tipo sem realimentação (*feed-forward*) composta de múltiplas camadas de neurônios. Na maioria das aplicações, a função de ativação do MLP é a função *sigmoid* e o método de aprendizado é o de retropropagação (*backpropagation*). O método de retropropagação consiste em ajustar os pesos de cada neurônio da rede neural de acordo com o gradiente de uma função de perda (*loss function*) arbitrária. O objetivo é minimizar o erro de predição dos neurônios em relação à saída real para aumentar a acurácia da classificação. Esse processo é repetido inúmeras vezes até que o erro seja suficientemente pequeno. A função de perda mais comumente utilizada na literatura é a média da distância Euclidiana entre a saída real e a predita expressa por

$$E = \frac{1}{2n}\sum_{x} ||y(x) - y'(x)||^2, \tag{5}$$

em que n é o número de entradas no conjunto de treinamento, x é a entrada avaliada e $y(x)$ e $y'(x)$ são, respectivamente, os valores de saída real e preditos. O ajuste dos pesos de um neurônio j da camada i é feito pela minimização da função perda, geralmente através do método do gradiente descendente

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}}, \tag{6}$$

em que η é um parâmetro de ajuste do modelo que controla a velocidade de atualização dos pesos w_{ij}. Os erros E_i de cada camada são retropropagados e os pesos ajustados de acordo com as derivadas parciais em relação a cada neurônio.

A Figura 3.20 mostra um exemplo de uma rede neural artificial MLP, composta por uma camada de entrada de três neurônios, que representam as características, duas camadas ocultas de quatro neurônios cada e uma camada de saída de dois neurônios, que representam as classes normal e ameaça. As redes neurais artificiais (ANN) têm a vantagem de aprender e modelar relacionamentos não-lineares e complexos. Embora um único neurônio não pareça vantajoso, conectar muitos neurônios de maneira adequada pode criar um modelo de aprendizado altamente eficaz. O poder das ANN é baseado no Teorema da Aproximação Universal, que afirma que a partir de funções de ativação, como *Sigmoid*, Tangente Hiperbólica, Linear Retificada etc., uma ANN pode modelar qualquer função contínua dados os requisitos de tamanho e de estrutura.

A **máquina de vetores de suporte** (*Support Vector Machine* - SVM) é um classificador binário, baseado no conceito de um plano de decisão que define os limites de

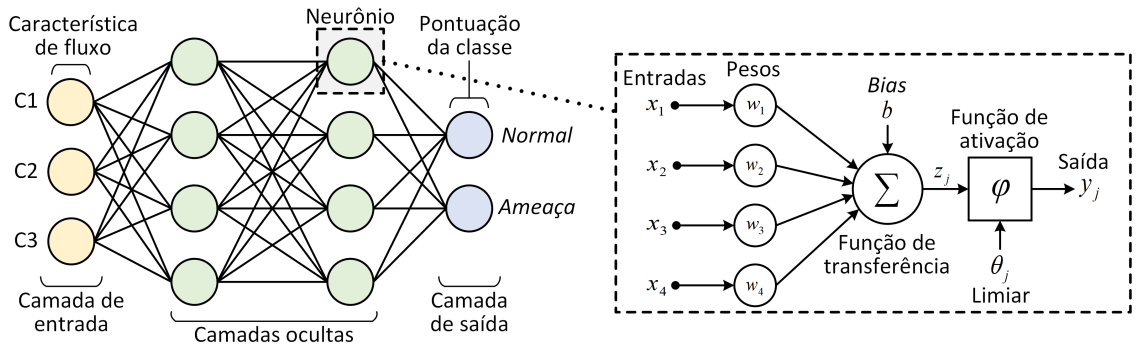
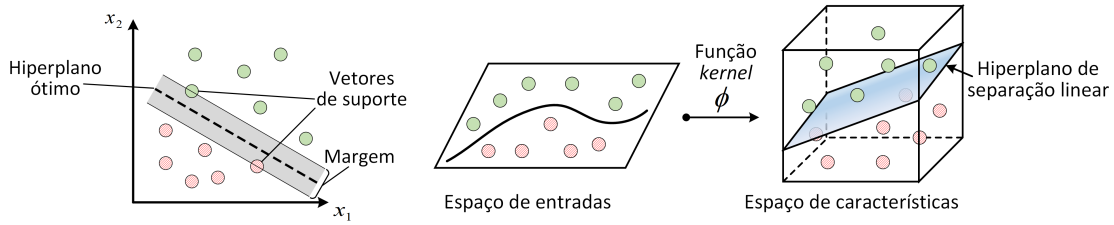


Figura 3.20. Rede neural artificial (ANN) com dois neurônios na camada de saída que representam as classes normal e ataque. Cada neurônio da rede neural realiza uma função, denominada função de ativação, sobre a soma ponderada de suas entradas. A saída desta função é propagada para os neurônios da camada seguinte.

decisão. Basicamente, a SVM classifica através da construção de um hiperplano em um espaço multidimensional que divide diferentes classes. Um algoritmo iterativo definido por uma função *kernel* minimiza uma função de erro, encontrando a melhor separação do hiperplano. Dessa forma, a SVM encontra o hiperplano de separação máxima, ou seja, o hiperplano que maximiza a distância entre as classes ou a margem. Uma ilustração desta separação é apresentada na Figura 3.21(a).



(a) Hiperplano SVM. (b) "Artifício do *kernel*" para o SVM.

Figura 3.21. a) Definição do hiperplano de separação ótima entre classes binárias no algoritmo SVM. b) Um artifício para obter uma separação linear a partir de um conjunto não-linear consiste em usar uma função *kernel* que mapeia o espaço de entrada no espaço de características.

Para uma classificação multiclasse, divide-se o problema em subproblemas de classificação binária. Assumindo como exemplo um problema com três classes, normal, negação de serviço (DoS) e varredura, isto significa, encontrar os hiperplanos de separação ótima entre: normal e não normal; DoS e não-DoS; e varredura e não-varredura. Uma vez obtido o resultado de cada classe, escolhe-se aquela com maior pontuação. A pontuação do classificador de uma amostra x é a distância de x para os limites de decisão, que vai de $-\infty$ a $+\infty$. A classificação do classificador é dada por

$$f(x) = \sum_{j=1}^{n} \alpha_j y_j G(x_j, x) + b, \tag{7}$$

em que $(\alpha_1, \ldots, \alpha_n)$ são os parâmetros estimados do algoritmo SVM e $G(x_j, x)$ é a função

kernel utilizada. Se não for possível separar os valores de entrada através de um hiperplano linear, uma estratégia é utilizar o "artifício de *kernel*" (*kernel-trick*), como mostra a Figura 3.21(b). O artifício de *kernel* utiliza uma função que leva o espaço de entrada dimensional n para um de dimensão maior m, tal que $m > n$, em que seja possível separar os valores usando um hiperplano. Os *kernels* mais utilizados são Polinomial, *Radial Basis Function* (RBF) e Sigmoid. A SVM é capaz de evitar o sobreajuste (*overfitting*). Uma vez que o hiperplano é encontrado, a maioria dos dados que não sejam os vetores de suporte, que são os pontos mais próximos do limite, tornam-se redundantes. Isso significa que mudanças pequenas nos dados não podem afetar o hiperplano.

A Tabela 3.4 mostra a acurácia de todos os métodos de classificação de ameaças com treinamento prévio apresentados. Além disso, essa tabela detalha a precisão e a sensibilidade do tráfego legítimo e malicioso. Como o principal objetivo da detecção de ameaças é acionar contramedidas, essas métricas mostram o número de alertas verdadeiros e falsos positivos que acionariam as medidas de segurança. Como mostra a Tabela 3.4, bons resultados para a classe normal foram obtidos, geralmente acima de 90%, tanto na precisão quanto na sensibilidade. Consequentemente, isso garante um baixo nível de falsos positivos que resultariam em bloqueio de tráfego legítimo. Em relação à classe de ameaça, para o conjunto de dados NetOp, os resultados de precisão são menores, porque o conjunto de dados é composto na maior parte de amostras legítimas. Portanto, ao avaliar o número absoluto de amostras normais classificadas como ameaças, elas têm um impacto negativo na precisão. No entanto, conforme mostrado nos resultados da classe normal, eles não resultam em muitos falsos positivos. Levando isso em consideração, a medida mais interessante para avaliar a classe de ameaça é a sensibilidade que mede a taxa de detecção de ameaças, ou seja, o número de ameaças classificadas corretamente entre todas as ameaças reais nos conjuntos de dados. Vale destacar que o SVM obteve a melhor tava de detecção, obtendo um sensibilidade na classe de ameaça acima de 87% nos dois conjuntos de dados. O SVM é um classificador robusto, que maximiza a margem para o hiperplano de decisão.

Tabela 3.4. Resumo da classificação de ameaças para os algoritmos Árvore de Decisão, Rede Neurais e SVM.

Algoritmo	Conj. Dados	Acurácia	Normal		Ameaça	
			Precisão	Sens.	Precisão	Sens.
Árvore de Decisão	GTA	80.6%	80.7%	94.6%	80.1%	48.8%
	NetOp	92.8%	97.3%	94.8%	51.5%	67.9%
Rede Neural	GTA	96.0%	94.8%	98.8%	97.9%	91.5%
	NetOp	95.1%	97.5%	97.2%	66.7%	69.2%
SVM	GTA	96.3%	95.4%	98.8%	98.0%	92.4%
	NetOp	95.8%	96.5%	98.9%	66.9%	87.2%

3.4.4. Detecção de Ameaças Inéditas em Tempo Real a partir de Treinamento Adaptativo

Os atacantes criam novos ataques (*zero-days attacks*) e alteram os ataques convencionais de forma a ludibriarem e passarem despercebidos por sistemas de detecção

de intrusão baseados em assinatura ou métodos de classificação em tempo diferenciado (*offline*). Para aumentar a robustez da detecção de ameaças, este capítulo apresenta um esquema adaptativo de coleta de dados capaz de aprender ataques novos e detectar ameaças que se modificam com o tempo [Lobato et al. 2018]. Este esquema de coleta de dados proposto garante adaptabilidade tanto para o comportamento legítimo como para o malicioso. Os métodos se adaptam a mudanças aceitáveis no uso da rede e aprendem novos ataques que são realizados nos *honeypots*. Assim, são propostos três métodos que adaptam seus parâmetros em tempo real à medida que os fluxos de dados chegam. Os dois primeiros algoritmos de detecção são algoritmos de classificação em linha (*online*) que visam classificar os ataques com base na captura de comportamento de ataques conhecidos pelo *honeypots*, enquanto o outro é um algoritmo de detecção de anomalia. Os três algoritmos implementados possuem a capacidade de detectar novas ameaças.

Os métodos de classificação em linha (*online*) requerem dados rotulados. Assim, no método de detecção proposto, todos os dados provenientes dos potes de mel (*honeypots*) são rotulados como ameaças, uma vez que não há nenhum serviço real sendo oferecido nos *honeypots* e todo o acesso é destinado a explorar uma vulnerabilidade intencionalmente instalada ou desconhecida *a priori*. Portanto, todos os fluxos que chegam aos *honeypots* serão usados pelos algoritmos para atualizar os parâmetros rotulados como ameaças. Essa realimentação de dados em tempo real garante comportamento adaptativo para a detecção de ameaças. Então, sempre que um atacante executa um novo ataque ou muda seu comportamento, os modelos de classificação são atualizados. Os métodos de classificação propostos assumem como tráfego benigno os fluxos recebidos nos sensores de tráfego na rede durante o tempo de configuração, que é um período em que o uso da rede é monitorado para garantir que todos os fluxos sejam legítimos. Após esse tempo, caso um fluxo proveniente dos sensores de rede seja classificado como ameaça, os parâmetros no algoritmo não são atualizados. Se o fluxo for considerado normal os parâmetros do algoritmo são atualizados adaptando-se às mudanças de comportamento normal da rede.

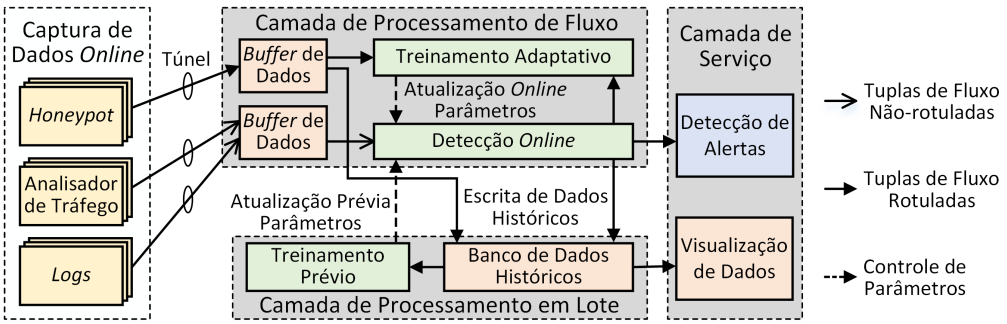


Figura 3.22. Arquitetura para a detecção em tempo real com treinamento adaptativo para proteção contra ataques desconhecidos.

A arquitetura mostrada na Figura 3.22 exemplifica um mecanismo de detecção de intrusão que processa dados em fluxo para adaptar os modelos treinados e aumentar a acurácia da detecção. Os dados da rede são coletados e enviados para um *buffer*. O algoritmo de aprendizado de máquina classifica o dado. Caso a amostra seja considerada como tráfego benigno é, então, usada como entrada na rotina de treinamento do algoritmo de aprendizado de máquina rotulada como benigna. Já as amostras de ameaças são coleta-

das em *honeypots* espalhados pela rede e são enviadas diretamente à rotina de treinamento rotuladas como ameaças. A cada amostra recebida, os algoritmos de aprendizado de máquina incrementais executam um passo de treinamento.

3.4.4.1. O Treinamento Adaptativo para Algoritmos de Detecção de Ameaças

O **algoritmo gradiente estocástico descendente com momento** é uma aproximação do algoritmo do gradiente estocástico descendente (*Stochastic Gradient Descent* - SGD), na qual o gradiente é aproximado por uma única amostra. Na aplicação de detecção de ameaças são consideradas duas classes: normal e ameaça. Portanto, a função Sigmoid, expressa por

$$h_\theta(\mathbf{x}) = \frac{1}{1+e^{-\theta^\intercal \mathbf{x}}} \tag{8}$$

é utilizada para executar a regressão logística. Na função Sigmoid, baixos valores de produto dos parâmetros $\theta^\intercal$ vezes o vetor da característica da amostra $\mathbf{x}$ retornam 0, enquanto valores altos retornam 1. Quando uma nova amostra $\mathbf{x}_{(i)}$ chega, o método avalia a função Sigmoid e retorna um para $h_\theta(\mathbf{x}_{(i)})$ maiores que 0.5 e zero, caso contrário. Essa decisão apresenta um custo associado, com base na classe real da amostra $y_{(i)}$. A função de custo é definida na Equação 9. Essa função é convexa e o objetivo do algoritmo SGD é encontrar o seu mínimo, expresso por

$$J_{(i)}(\theta) = y_{(i)} log(h_\theta(\mathbf{x}_{(i)})) + (1-y_{(i)}) log(1-h_\theta(\mathbf{x}_{(i)})). \tag{9}$$

Quando uma nova amostra chega, o algoritmo dá um passo em direção ao mínimo custo com base no gradiente desta função.

Algoritmo 1: Gradiente Estocástico Descendente com Momento.

Entrada: Características de fluxo de entrada x, Classe y
Saida : Classe Prevista *predict*, Parâmetros do treinamento θ

Inicializar θ, $\Delta\theta$, α, β;
for $i \leftarrow 1$ **to** m **do**
  $h_\theta(x_{(i)}) = \frac{1}{1+e^{-\theta^\intercal x_{(i)}}}$;
  $predict = round(h_\theta(x_{(i)}))$;
  if $predict == 1$ *and* $y_{(i)} == 0$ **then**
    Envia Alerta;
  else
    $\theta = \theta - \alpha \nabla J_{(i)}(\theta) + \beta \Delta\theta$;
    $\Delta\theta = \alpha \nabla J_{(i)}(\theta)$;
  end
end

O Algoritmo 1 mostra a implementação do algoritmo do SGD. A cada amostra recebida do vetor $\mathbf{x}_{(i)}$ o método determina a classe $y_{(i)}$ baseado no tipo da origem. Se

a amostra vier de um *honeypot* o rótulo é 1, enquanto se vier de uma sonda de trafego o rótulo é 0. Se a amostra vem de um analisador de tráfego e o SGD prevê como uma ameaça, o algoritmo envia um alerta. Caso contrário, atualiza os parâmetros com base no gradiente da função custo. O termo $\Delta\theta$ é o momento e tem o valor da atualização anterior do parâmetro. No contexto do SGD, este termo considera o movimento passado ao atualizar os parâmetros θ. Os parâmetros α e β são periodicamente atualizados em tempo diferenciado, *offline*, com base no custo histórico de cada amostra.

A **máquina de vetores de suporte incremental** (*Incremental Support Vector Machine* - ISVM) é um classificador binário com base no conceito de plano de decisão que define os limites das classes. Um hiperplano construído em um espaço multidimensional divide os dados. O algoritmo SVM *online* usa uma aproximação de margem suave com a função convexa de perda de articulação (*hinge loss*), dada por

$$\max\{0, 1 - y\theta^{\intercal}x\}. \tag{10}$$

O objetivo deste algoritmo é minimizar a função de perda. Assim como o algoritmo anterior, o método determina a classe $y_{(i)}$ com base na origem. Se vier de um *honeypot*, o rótulo é 1, enquanto que se ele vem de um sensor de rede, o rótulo é -1. Novamente, quando um tráfego de uma sonda é classificado como ameaça, o método envia um alerta e não atualiza o modelo. O Algoritmo 2 mostra a implementação do SVM *online*. Os parâmetros α, λ são periodicamente atualizados com base na avaliação da função de perda sobre as amostras históricas.

Algoritmo 2: Máquina de Vetores de Suporte Incremental.

Entrada: Características de fluxo de entrada x, Classe y
Saída : Classe prevista *predict*, parâmetros do treinamento θ

Inicializar θ, α, λ;
for $i \leftarrow 1$ **to** m **do**
  $predict = sign(\theta^{\intercal}x_{(i)})$;
  if $predict == 1$ *and* $y_{(i)} == -1$ **then**
    Envia Alerta;
  else
    if $y_{(i)}\theta^{\intercal}x_{(i)} > 1$ **then**
      $\nabla_{(i)} = \theta$;
    else
      $\nabla_{(i)} = -\lambda y_{(i)}x_{(i)}$;
    end
    $\theta = \theta - \alpha\nabla_{(i)}$
  end
end

O algoritmo de **detecção de anomalia pela distribuição normal** utiliza a distância da característica da amostra para a média de uma distribuição normal para detectar anomalias. Dessa forma, as anomalias são detectadas quando a distância de uma amostra normal do conjunto de dados de treinamento para a média é maior que um limiar vezes

a variância para pelo menos uma das características. A implementação em tempo real realiza a detecção de anomalia à medida que os dados de transmissão estão chegando. Portanto, uma anomalia é detectada se pelo menos uma das seguintes condições

$$X_j > \mu_j + limiar * \sigma_j^2 \qquad (11) \qquad\qquad X_j < \mu_j - limiar * \sigma_j^2 \qquad (12)$$

é verdadeira para pelo menos uma característica j, levando em consideração os μ_j e as variâncias σ_j^2 calculadas no treinamento. Além disso, quando uma nova amostra chega e não é detectada como uma anomalia pelas condições (11) e (12), a média μ_j e a variância Σ_j^2 de cada característica definida por

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} X_j \qquad (13) \qquad\qquad \sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(X_j - \mu_j)^2 \qquad (14)$$

são atualizados, considerando a nova amostra. Os valores atuais do total de amostras N e sua soma são armazenados na plataforma e incrementados quando uma nova amostra chega, considerando cada recurso X_j. Consequentemente, os parâmetros de distribuição normais são sempre atualizados por amostras consideradas legítimas, garantindo adaptabilidade.

O algoritmo de **detecção de anomalia pela série temporal de entropia** identifica uma anomalia pela análise do valor de entropia de uma janela deslizante de fluxos. A entropia de amostra, expressa por

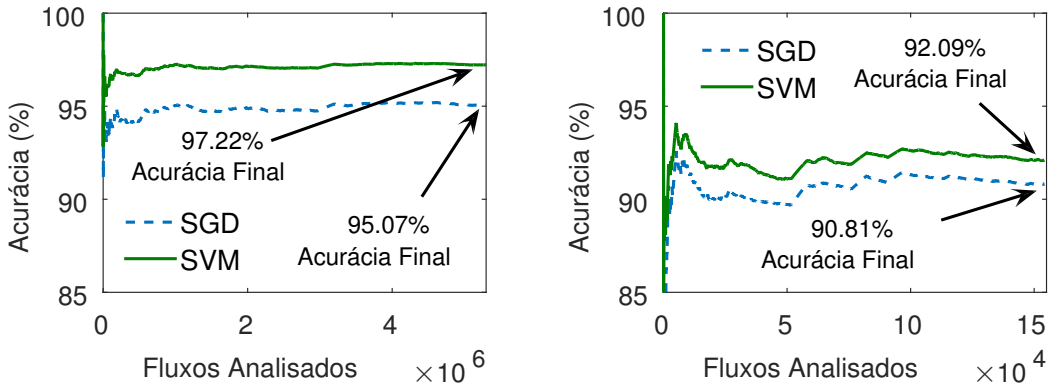
$$H(X) = -\sum_{i=1}^{N}(\frac{n_i}{S})\log_2(\frac{n_i}{S}), \qquad (15)$$

indica o grau de concentração ou dispersão de uma característica, em que S é o número total de observações, n_i é o número de observações dentro do intervalo i de valores e N é o número de intervalos. Quando todos os valores estão concentrados em um intervalo, $H(X)$ é igual a zero e quando cada valor está em uma faixa diferente i, o valor de $H(X)$ é $\log_2(N)$. Então, dada uma série de observações X, a entropia de amostra resume o nível de concentração em um único valor. Foi definida uma janela deslizante de 40 fluxos e calculado o valor de $H(X)$ para cada uma dessas janelas, gerando as séries temporais. Outro parâmetro determinado na fase de configuração é o intervalo contendo a maioria das amostras. Estes parâmetros são atualizados conforme novas amostras chegam. Os valores normais de entropia do tráfego tendem a ser concentrados e o valor mais frequente tende a estar no centro. Assim, o esquema de detecção de ameaça por anomalia proposto define um limiar que determina a distância aceita da entropia $H(X)$ para o intervalo mais frequente [Lobato et al. 2018].

3.4.4.2. Resultados de Detecção em Tempo Real com Treinamento Adaptativo

Os algoritmos de treinamento adaptativo e os algoritmos de detecção de anomalias foram avaliados para dois conjuntos de dados: o conjunto de dados desenvolvido no GTA/UFRJ e o conjunto de dados da operadora de telecomunicações (NetOp). A Figura 3.23 mostra o comportamento da precisão ao longo do tempo para cada amostra

recebida para a aplicação dos algoritmos de aprendizado de máquina com treinamento adaptativo, SGD e SVM. As Figuras 3.23(a) e 3.23(b) mostram os resultados de cada conjunto de dados. As precisões para os algoritmos implementados estabilizam em valores altos, sempre acima de 90%, mesmo quando ameaças desconhecidas aparecem nos conjuntos de dados. No início da análise, os métodos não conhecem ameaças. No entanto, à medida que os dados do pote de mel (*honeypot*) chegam, os métodos aprendem as ameaças e alcançam altas taxas de detecção de ameaças. Como os métodos de detecção não conhecem nenhuma das 16 ameaças no conjunto de dados do GTA/UFRJ e as 13 ameaças no conjunto de dados NetOp, essas ameaças são consideradas ameaças inéditas (*zero-day threats*) que são aprendidas e detectadas em linha (*online*) à medida que novas amostras chegam.



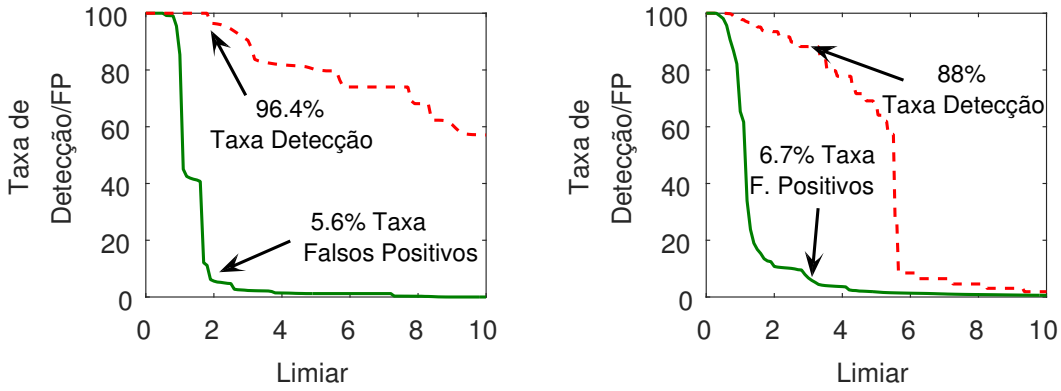
(a) Conjunto de dados GTA/UFRJ (b) Conjunto de dados NetOp

Figura 3.23. Acurácia para ambos os esquemas propostos conforme as amostras chegam. Em ambos os conjuntos de dados, a acurácia permanece estável mesmo com novos ataques e mudanças de comportamento de uso legítimo.

Como os dois esquemas de detecção de anomalia não são supervisionados e modelam o comportamento normal do usuário, são utilizados apenas os dados do *honeypot* para avaliar o desempenho do algoritmo. Os parâmetros são atualizados durante o tempo de configuração com base apenas nos dados capturados pelos analisadores de tráfego. Após esse período, se uma amostra for considerada normal, os parâmetros serão atualizados. Por outro lado, se for considerado malicioso, um alerta é enviado e os parâmetros dos algoritmos não são atualizados. Para avaliar os algoritmos, foi utilizado 70 % dos dados normais no tempo de *setup* e 30 % para avaliar os falsos positivos. A taxa de detecção de ataques é obtida com base em todas as ameaças nos conjuntos de dados.

A Figura 3.24 mostra os resultados do algoritmo de detecção de anomalia pela distribuição normal para diferentes valores limiares. Para o conjunto de dados do GTA/UFRJ, observa-se que selecionando um limiar de 2, o algoritmo apresenta um compromisso muito bom entre 96,4 % da taxa de detecção e 5,6 % da taxa de falsos positivos. O mesmo comportamento é apresentado para o conjunto de dados NetOp, com um bom compromisso com um limiar de 3, apresentando 88 % de taxa de detecção de ameaças e 6,7 % de taxa de falsos positivos. O parâmetro do limiar pode ser ajustado de acordo com a relação entre verdadeiros positivos e falso positivos desejada. Para redes com requisitos de segurança altos, esse limiar deve ser pequeno, resultando em alta taxa de detecção ao

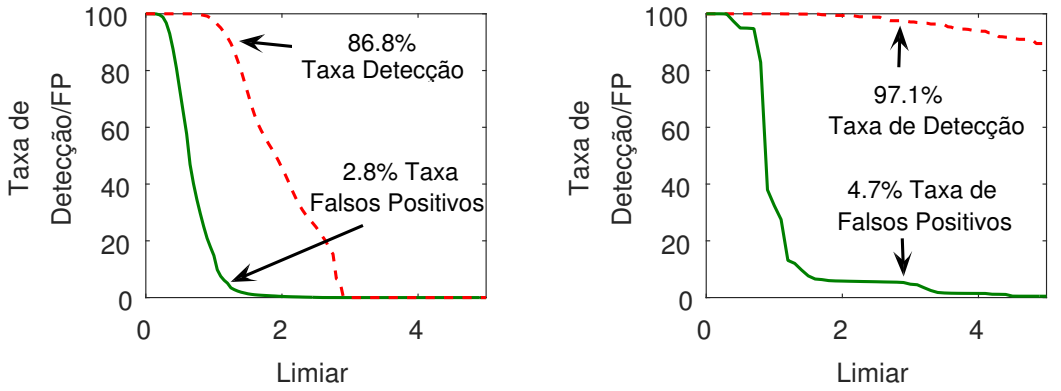
custo de uma taxa de falsos positivos mais alta [Lobato et al. 2018].



(a) Conjunto de dados GTA/UFRJ. (b) Conjunto de dados NetOp.

Figura 3.24. Detecção de anomalia pela distribuição normal: análise do compromisso entre a taxa de detecção e a taxa de falsos positivos ao variar o parâmetro de limiar.

A Figura 3.25 mostra os resultados da detecção de anomalia por série temporal de entropia para diferentes limiares. Para limiares pequenos, a taxa de detecção é muito alta, resultando também em uma alta taxa de falsos positivos. Para valores baixos, ocorre o inverso, uma baixa taxa de falso positivo a um custo também de baixa taxa de detecção. No entanto, existem valores limiares que apresentam um bom equilíbrio entre essas taxas. Para o conjunto de dados GTA/UFRJ, o limiar de 1,3 resulta em 86,8 % de taxa de detecção de ataques com apenas 2,8 % de taxa de falsos positivos, enquanto para o conjunto de dados NetOp, o limiar 3.0 obtém uma baixa taxa de falsos positivos de 4,7 % com um taxa de detecção de ataque notavelmente alta de 97,1 %.



(a) Conjunto de dados GTA/UFRJ. (b) Conjunto de dados NetOp.

Figura 3.25. Detecção de anomalia por série temporal de entropia: análise do compromisso entre a taxa de detecção e a taxa de falsos positivos ao variar o parâmetro de limiar.

3.5. Considerações Finais e Problemas em Aberto

O uso de plataformas de processamento distribuídos de fluxo representa uma nova direção na pesquisa de detecção de ameaças. O paralelismo e a possibilidade de montar grafos de processamento permitem que métodos de detecção complexos rodem simultaneamente e de forma escalável no volume de grandes massas de dados e em tempo real. Este capítulo apresentou o problema de detecção de ameaça em tempo real baseada em aprendizado de máquina usando plataformas de processamento de fluxo de código aberto. As plataforma de processamento distribuído Apache Storm, Apache Flink e Apache Spark são apresentadas e o seu desempenho foi avaliado. Foram introduzidos conceitos de conjunto de dados e mostrados os conjuntos de dados usados nas avaliações de desempenho. Técnicas usadas para redução de dimensionalidade e seleção de características foram apresentadas assim como as principais métricas sistemas de aprendizado de máquina em detecção de intrusão. O foco do capítulo foram as arquiteturas de processamento distribuído de fluxo que seguem a arquitetura lambda, adaptativas e de processamento incremental.

A detecção de anomalias através do processamento distribuído de fluxos de dados é um desafio, pois os dados são gerados nas redes de forma contínua e com alta velocidade. Algoritmos que processam fluxos de dados proveem soluções aproximadas e uma resposta rápida usando poucos recursos de memória. A resposta aproximada pode ser bastante útil para uma primeira análise da anomalia, mas a medida em que se exige uma margem de erro menor, os recursos computacionais exigidos aumentam, assim como as dimensões das entradas de dados também aumentam. As tendências das pesquisas atuais em processamento de fluxos de dados envolvem a busca por algoritmos e estruturas de dados com limites estreitos de erro. Outro problema importante ainda em aberto envolve o processamento de fluxo sem invadir a privacidade dos usuários. Cada vez mais as aplicações usam protocolos seguros nos quais os dados são criptografados. Como fazer análises em dados criptografados ainda é um desafio. Há, ainda, questões semânticas em relação aos operadores aplicados sobre os dados.

A taxa de geração dos fluxos varia, podendo atingir elevados valores que viriam por estourar a fila de processamento dos sistemas de detecção de ameaças. Caso isso ocorra, ataques passariam despercebidos, o que representa uma ameaça real à rede e um problema em aberto. O uso de técnicas de mineração de dados em fluxo visa diminuir a quantidade de recursos necessários para o processamento de dados. Em cenários de sobrecarga, técnicas como amostragem, descarte de carga (*Load Shedding*), estruturas de dados de Sinopse (*Synopsis*), entre outras, podem melhorar a segurança, ao diminuir a representação dos dados ao invés de simplesmente descartar as amostras quando a fila estiver cheia [Gaber et al. 2005, Gama and Rodrigues 2007].

A arquitetura *lambda* mostrou-se acurada e rápida, permitindo que os dados em fluxo sejam tratados em tempo quase real, pois o processamento sobre cada amostra é reduzido, já que o treinamento e adaptação de modelos são realizados em tempo diferenciado. Por depender de treinamento prévio, a arquitetura lambda não é capaz de identificar ameaças inéditas. A principal vantagem da arquitetura de classificação de tráfego com aprendizado adaptativo é identificar ameaças inéditas (*zero-day threats*), permitindo um aperfeiçoamento gradual do modelo conforme novas amostras rotuladas surgem. Por

depender de dados rotulados recebidos em tempo real, usou-se potes de mel (*honeypots*) na rede para capturar o comportamento das ameaças. No entanto, introduz um custo maior de processamento por amostra ao obter um passo de adaptação para os algoritmos incrementais.

Há iniciativas tanto nacionais como internacionais em plataformas para detecção de ameaças usando aprendizado de máquina, tais como:

- CATRACA[18] é uma ferramenta de processamento distribuído de fluxos de dados baseada na plataforma Spark Streaming que realiza detecção de ameaças em tempo real. A ferramenta é desenvolvida pelo Grupo de Teleinformática e Automação / UFRJ e aplica a arquitetura lambda de processamento de fluxo. Os pacotes na rede são abstraídos em fluxos identificados pela 5-tupla (IP de origem e destino, portas de origem e destino e protocolo de transporte) dentro de janelas saltitantes de 2 segundos. As estatísticas dos fluxos são carregadas no sistema de mensagens Kafka que as armazena até serem consumidas na forma de fluxos discretos (D-Streams) pela Spark Streaming. Na Spark Streaming, executam-se algoritmos de seleção de características, para fazer a filtragem das características mais importantes, e o algoritmo de classificação por árvores de decisão que se mostrou o mais adequado para a identificação de ameaças em rede [Andreoni Lopez et al. 2017d, Andreoni Lopez et al. 2017c]. Os fluxos de dados ainda são enriquecidos com informações de geolocalização dos IPs de origem e destino. A ferramenta conta ainda com uma interface de visualização dos resultados da classificação baseada na plataforma Kibana da pilha ElasticSearch[19]. Em caso de detecção de ameaça, a contramedida aplicada aos fluxos considerados maliciosos é o bloqueio imediato dos fluxos.

- Apache Metron[20] é um arcabouço de análise de segurança baseado no processamento de grandes massas de dados. Portanto, sua arquitetura consiste nas camadas de aquisição, consumo, processamento distribuído, enriquecimento, armazenamento e visualização dos dados. A ideia chave desse arcabouço é permitir a correlação de eventos de segurança originados de fontes distintas. Para tanto, o arcabouço emprega sensores na rede como fonte de dados, registros de ações (*logs*) de elementos ativos de segurança em rede e fontes de telemetria. O arcabouço se vale ainda de uma base histórica de ameaças de rede proveniente da Cisco;

- Apache Spot[21] é um projeto semelhante ao Apache Metron ainda em incubação. O Apache Spot utiliza técnicas de telemetria e aprendizado de máquinas para análise de fluxo e pacotes a fim de detectar ameaças. Os criadores afirmam que a grande diferença com o Apache Metron é a capacidade de utilizar modelos comuns de dados abertos para rede;

- Stream4Flow[22] utiliza o Apache Spark com a pilha ElasticStack para fazer monitoramento de redes. O protótipo serve como visualização dos parâmetros da rede.

[18] https://www.gta.ufrj.br/catraca.
[19] https://www.elastic.co/.
[20] http://metron.apache.org/
[21] http://spot.incubator.apache.org
[22] https://github.com/CSIRT-MU/Stream4Flow

Stream4Flow, no entanto, não possui nenhuma inteligência para realizar detecção de anomalias.

- Hogzilla[23] é um sistema de detecção de intrusão (IDS) com suporte para Snort, SFlows, GrayLog, Apache Spark, HBase e libnDPI, que provê detecção de anomalias em rede. Hogzilla permite também realizar a visualização do tráfego da rede.

Os algoritmos de aprendizado incremental atualizam as estatísticas nas quais baseiam a classificação com a chegada de cada nova amostra. Contudo, no caso de uma mudança no comportamento estacionário das estatísticas das características das amostras, a classificação pode apresentar baixa acurácia, chamado de *concept drift*. Um tema de pesquisa importante é o estudo e desenvolvimento de algoritmos de aprendizado de máquina resistentes ao *concept drift*.

Uma nova abordagem é de se realizar o aprendizado incremental por agregados de classificadores [Polikar et al. 2001], cuja a ideia central é realizar a classificação através de diversos classificadores de baixa acurácia e alta variabilidade e, então, realizar a classificação final através de um esquema de votação. A alta variabilidade dos classificadores permite que o agregado de classificação reaja mais prontamente a mudanças no processo de chegada de novas amostras. O aglomerado de classificadores é, então, um possível caminho futuro para o desenvolvimento de sistemas de detecção de intrusão baseados em aprendizado de máquina.

Referências

[Abadi et al. 2005] Abadi, D. J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., and Zdonik, S. B. (2005). The Design of the Borealis Stream Processing Engine. *Cidr*, pages 277–289.

[Akidau et al. 2013] Akidau, T., Balikov, A., Bekiroglu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., and Whittle, S. (2013). MillWheel: Fault-tolerant Stream Processing at Internet Scale. *Proc. VLDB Endow.*, 6(11):1033–1044.

[Andreoni Lopez et al. 2016a] Andreoni Lopez, M., Ferrazani Mattos, D., and Duarte, O. C. M. B. (2016a). An elastic intrusion detection system for software networks. *Annales des Telecommunications/Annals of Telecommunications*, 71(11-12):595–605.

[Andreoni Lopez et al. 2016b] Andreoni Lopez, M., Lobato, A. G. P., and Duarte, O. C. M. B. (2016b). A Performance Comparison of Open-Source Stream Processing Platforms. In *IEEE GLOBECOM*, pages 1–6, Washington, USA. IEEE.

[Andreoni Lopez et al. 2016c] Andreoni Lopez, M., Lobato, A. G. P., and Duarte, O. C. M. B. (2016c). Monitoramento de Tráfego e Detecção de Ameaças por Sistemas Distribuídos de Processamento de Fluxos: uma Análise de Desempenho. *XXI Workshop de Gerência e Operação de Redes e Serviços (WGRS) do SBRC'2016*, pages 103–116.

[23] http://ids-hogzilla.org/

[Andreoni Lopez et al. 2018] Andreoni Lopez, M., Lobato, A. G. P., Duarte, O. C. M. B., and Pujolle, G. (2018). An evaluation of a virtual network function for real-time threat detection using stream processing. In *IEEE Fourth International Conference on Mobile and Secure Services (MobiSecServ)*, pages 1–5.

[Andreoni Lopez et al. 2017a] Andreoni Lopez, M., Lobato, A. G. P., Mattos, D. M. F., Alvarenga, I. D., Duarte, O. C. M. B., and Pujolle, G. (2017a). Um Algoritmo Não Supervisionado e Rápido para Seleção de Características em Classificação de Tráfego. In *XXXV SBRC'2017*, Belém- Pará, PA,.

[Andreoni Lopez et al. 2017b] Andreoni Lopez, M., Sanz, I. J., Ferrazani Mattos, D. M., Duarte, O. C. M. B., and Pujolle, G. (2017b). CATRACA: uma Ferramenta para Classificação e Análise Tráfego Escalável Baseada em Processamento por Fluxo. In *Salão de Ferramentas do XVII Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais - SBSeg'2017*, pages 788–795.

[Andreoni Lopez et al. 2017c] Andreoni Lopez, M., Silva, R., Alvarenga, I. D., Mattos, D. M. F., and Duarte, O. C. M. B. (2017c). Coleta e Caracterização de um Conjunto de Dados de Tráfego Real de Redes de Acesso em Banda Larga. In *XXII WGRS'17*.

[Andreoni Lopez et al. 2017d] Andreoni Lopez, M., Silva, R. S., Alvarenga, I. D., Rebello, G. A. F., Sanz, I. J., Lobato, A. G. P., Mattos, D. M. F., Duarte, O. C. M. B., and Pujolle, G. (2017d). Collecting and characterizing a real broadband access network traffic dataset. In *IEEE/IFIP 1st Cyber Security in Networking Conference (CSNet)*, pages 1–8.

[Arasu et al. 2004] Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., and Widom, J. (2004). STREAM: The Stanford Data Stream Management System. Technical Report 2004-20, Stanford InfoLab.

[ARMOR 2018] ARMOR (2018). The black market report 2018-03. `https://www.armor.com/app/uploads/2018/03/2018-Q1-Reports-BlackMarket-DIGITAL.pdf`. Acessado em 20/03/2018.

[Balazinska et al. 2004] Balazinska, M., Balakrishnan, H., and Stonebraker, M. (2004). Load management and high availability in the Medusa distributed stream processing system. *ACM SIGMOD international conference on management of data*, pages 929–930.

[Ballard et al. 2014] Ballard, C., Brandt, O., Devaraju, B., Farrell, D., Foster, K., Howard, C., Nicholls, P., Pasricha, A., Rea, R., Schulz, N., and others (2014). IBM InfoSphere Streams. *Accelerating Deployments with Analytic Accelerators, ser. Redbook. IBM*.

[Batista et al. 2004] Batista, G. E. A. P. A., Prati, R. C., and Monard, M. C. (2004). A study of the behavior of several methods for balancing machine learning training data. *SIGKDD Explor. Newsl.*, 6(1):20–29.

[Buczak and Guven 2015] Buczak, A. and Guven, E. (2015). A Survey of Data Mining and Machine Learning Methods for Cyber Security Intrusion Detection. *IEEE Communications Surveys Tutorials*, (99):1–26.

[Carbone et al. 2015a] Carbone, P., Ewen, S., Haridi, S., Katsifodimos, A., Markl, V., and Tzoumas, K. (2015a). Apache Flink: Unified Stream and Batch Processing in a Single Engine. *Data Engineering*, pages 28–38.

[Carbone et al. 2015b] Carbone, P., Fóra, G., Ewen, S., Haridi, S., and Tzoumas, K. (2015b). Lightweight Asynchronous Snapshots for Distributed Dataflows. *Computing Research Repository (CoRR)*, abs/1506.0.

[Cárdenas and Baras 2006] Cárdenas, A. A. and Baras, J. S. (2006). Evaluation of classifiers: Practical considerations for security applications.

[Cárdenas et al. 2013] Cárdenas, A. A., Manadhata, P. K., and Rajan, S. (2013). Big data analytics for security intelligence. *University of Texas at Dallas@ Cloud Security Alliance*.

[Carney et al. 2002] Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., and Zdonik, S. (2002). Monitoring Streams: A New Class of Data Management Applications. In *28th International Conference on Very Large Data Bases*, pages 215–226.

[Chandrasekar et al. 2017] Chandrasekar, K., Cleary, G., Cox, O., Lau, H., Nahorneyand, B., Gorman, B. O., O'Brien, D., Wallacen, S., Wood, P., and Wuees, C. (2017). Internet security threat report-symantec corporation,v22. `https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf`. Acessado em 20/03/2018.

[Chandrasekaran et al. 2003] Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S. R., Reiss, F., and Shah, M. A. (2003). TelegraphCQ: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, page 668. ACM.

[Chandrashekar and Sahin 2014] Chandrashekar, G. and Sahin, F. (2014). A survey on feature selection methods. *Computers & Electrical Engineering*, 40(1):16–28.

[Chen et al. 2000] Chen, J., DeWitt, D. J., Tian, F., and Wang, Y. (2000). NiagaraCQ: A scalable continuous query system for internet databases. In *ACM SIGMOD Record*, volume 29, pages 379–390. ACM.

[Cheng et al. 2010] Cheng, Z., Caverlee, J., and Lee, K. (2010). You Are Where You Tweet: A Content-based Approach to Geo-locating Twitter Users. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, CIKM '10, pages 759–768. ACM.

[Clay 2015] Clay, P. (2015). A modern threat response framework. *Network Security*, 2015(4):5–10.

[Costa et al. 2012] Costa, L. H. M. K., de Amorim, M. D., Campista, M. E. M., Rubinstein, M., Florissi, P., and Duarte, O. C. M. B. (2012). Grandes Massas de Dados na Nuvem: Desafios e Técnicas para Inovação. In *SBRC 2012 - Minicursos*.

[Demers et al. 2007] Demers, A. J., Gehrke, J., Panda, B., Riedewald, M., Sharma, V., White, W. M., and others (2007). Cayuga: A General Purpose Event Monitoring System. In *Proceedings of the Conference on Innovative Data Systems Research*, volume 7, pages 412–422.

[Domingos 2012] Domingos, P. (2012). A few useful things to know about machine learning. *Commun. ACM*, 55(10):78–87.

[Fontugne et al. 2010] Fontugne, R., Borgnat, P., Abry, P., and Fukuda, K. (2010). MAWILab: Combining diverse anomaly detectors for automated anomaly labeling and performance benchmarking. In *ACM CoNEXT '10*, Philadelphia, PA.

[Franklin 2013] Franklin, M. (2013). The Berkeley Data Analytics Stack: Present and future. In *IEEE International Conference on Big Data*, pages 2–3. IEEE.

[Gaber et al. 2005] Gaber, M. M., Zaslavsky, A., and Krishnaswamy, S. (2005). Mining data streams: A review. *SIGMOD Rec.*, 34(2):18–26.

[Gaffney and Ulvila 2001] Gaffney, J. E. and Ulvila, J. W. (2001). Evaluation of intrusion detectors: a decision theory approach. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S P 2001*, pages 50–61.

[Gama and Rodrigues 2007] Gama, J. and Rodrigues, P. P. (2007). Data stream processing. In *Learning from Data Streams*, pages 25–39. Springer.

[Garcia et al. 2014] Garcia, S., Grill, M., Stiborek, J., and Zunino, A. (2014). An empirical comparison of botnet detection methods. *computers & security*, 45:100–123.

[Gluhak et al. 2011] Gluhak, A., Krco, S., Nati, M., Pfisterer, D., Mitton, N., and Razafindralambo, T. (2011). A survey on facilities for experimental Internet of things research. *IEEE Communications Magazine*, 49(11):58–67.

[Gu et al. 2006] Gu, G., Fogla, P., Dagon, D., Lee, W., and Skorić, B. (2006). Measuring intrusion detection capability: An information-theoretic approach. In *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security*, ASIACCS '06, pages 90–101, New York, NY, USA. ACM.

[Haines et al. 2001] Haines, J. W., Lippmann, R. P., Fried, D. J., Zissman, M., and Tran, E. (2001). 1999 darpa intrusion detection evaluation: Design and procedures. Technical report, Massachusetts Inst Of Tech Lexington Lincoln Lab.

[Heidemann and Papdopoulos 2009] Heidemann, J. and Papdopoulos, C. (2009). Uses and challenges for network datasets. In *Conference For Homeland Security, 2009. CATCH'09. Cybersecurity Applications & Technology*, pages 73–82. IEEE.

[Henke et al. 2011] Henke, M., Santos, C., Nunan, E., Feitosa, E., dos Santos, E., and Souto, E. (2011). Aprendizagem de máquina para segurança em redes de computadores: Métodos e aplicações. *Minicursos do XI Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais (SBSeg 2011)*, 1:53–103.

[ICA 2017] ICA, I. C. A. (2017). Assessing russian activities and intentions in recent us elections,ica 2017-01d. `https://www.dni.gov/files/documents/ICA_2017_01.pdf`. Acessado em 20/03/2018.

[Jiang et al. 2010] Jiang, W., Ravi, V. T., and Agrawal, G. (2010). A Map-Reduce system with an alternate API for multi-core environments. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 84–93. IEEE Computer Society.

[Kala Karun and Chitharanjan 2013] Kala Karun, A. and Chitharanjan, K. (2013). A review on Hadoop—HDFS infrastructure extensions. In *IEEE Conference on Information & Communication Technologies (ICT)*, pages 132–137. IEEE.

[Kamburugamuve et al. 2013] Kamburugamuve, S., Fox, G., Leake, D., and Qiu, J. (2013). Survey of distributed stream processing for large stream sources.

[Lee et al. 1999] Lee, W., Stolfo, S. J., and Mok, K. W. (1999). Mining in a data-flow environment: Experience in network intrusion detection. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 114–124. ACM.

[Lippmann et al. 2000a] Lippmann, R. P., Fried, D. J., Graf, I., Haines, J. W., Kendall, K. R., McClung, D., Weber, D., Webster, S. E., Wyschogrod, D., Cunningham, R. K., and others (2000a). Evaluating intrusion detection systems: The 1998 DARPA off-line intrusion detection evaluation. In *Proceedings of DARPA Information Survivability Conference and Exposition. DISCEX'00.*, volume 2, pages 12–26. IEEE.

[Lippmann et al. 2000b] Lippmann, R. P., Fried, D. J., Graf, I., Haines, J. W., Kendall, K. R., McClung, D., Weber, D., Webster, S. E., Wyschogrod, D., Cunningham, R. K., and Zissman, M. A. (2000b). Evaluating intrusion detection systems: the 1998 darpa off-line intrusion detection evaluation. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, volume 2, pages 12–26 vol.2.

[Lobato et al. 2016] Lobato, A. G. P., Andreoni Lopez, M., and Duarte, O. C. M. B. (2016). Um Sistema Acurado de Detecção de Ameaças em Tempo Real por Processamento de Fluxos. In *SBRC'2016*, pages 572–585, Salvador, Bahia.

[Lobato et al. 2017] Lobato, A. G. P., Andreoni Lopez, M., Rebello, G. A. F., and Duarte, O. C. M. B. (2017). Um Sistema Adaptativo de Detecção e Reação a Ameaças. In *Anais do XVII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais - SBSeg'17*, pages 400–413.

[Lobato et al. 2018] Lobato, A. G. P., Andreoni Lopez, M., Sanz, I. J., Cárdenas, A., Duarte, O. C. M. B., and Pujolle, G. (2018). An adaptive Real-Time architecture for

Zero-Day threat detection. In *to be published in IEEE ICC NGNI 2018*, Kansas City, USA.

[Marz and Warren 2013] Marz, N. and Warren, J. (2013). *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Manning Publications Co., Greenwich, CT, USA, 1st edition.

[Mattos et al. 2018] Mattos, D. M. F., Velloso, P. B., and Duarte, O. C. M. B. (2018). Uma infraestrutura Ágil e efetiva de virtualização de funções de rede para a internet das coisas. In *XXXVI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos - SBRC'2018*.

[Mayhew et al. 2015] Mayhew, M., Atighetchi, M., Adler, A., and Greenstadt, R. (2015). Use of machine learning in big data analytics for insider threat detection. In *IEEE Military Communications Conference, MILCOM*, pages 915–922.

[Mladenić 2006] Mladenić, D. (2006). Feature Selection for Dimensionality Reduction. In Saunders, C., Grobelnik, M., Gunn, S., and Shawe-Taylor, J., editors, *Subspace, Latent Structure and Feature Selection (SLSFS): Statistical and Optimization Perspectives Workshop.*, pages 84–102. Springer Berlin Heidelberg, Bohinj, Slovenia.

[Neumeyer et al. 2010] Neumeyer, L., Robbins, B., Nair, A., and Kesari, A. (2010). S4: Distributed Stream Computing Platform. In *IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 170–177. IEEE.

[Polikar et al. 2001] Polikar, R., Upda, L., Upda, S. S., and Honavar, V. (2001). Learn++: an incremental learning algorithm for supervised neural networks. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 31(4):497–508.

[Robnik-Šikonja and Kononenko 2003] Robnik-Šikonja, M. and Kononenko, I. (2003). Theoretical and Empirical Analysis of ReliefF and RReliefF. *Machine Learning*, 53(1/2):23–69.

[Sanz et al. 2017] Sanz, J. I., Andreoni Lopez, M., Mattos, D. M. F., and Duarte, O. C. M. B. (2017). A Cooperation-Aware Virtual Network Function for Proactive Detection of Distributed Port Scanning. In *IEEE/IFIP 1st Cyber Security in Networking Conference (CSNet'17)*.

[Schölkopf et al. 1999] Schölkopf, B., Smola, A. J., and Müller, K.-R. (1999). Kernel principal component analysis. In *Advances in kernel methods*, pages 327–352. MIT Press.

[Shiravi et al. 2012] Shiravi, A., Shiravi, H., Tavallaee, M., and Ghorbani, A. A. (2012). Toward developing a systematic approach to generate benchmark datasets for intrusion detection. *Computers and Security*, 31(3):357 – 374.

[Sommer and Paxson 2010] Sommer, R. and Paxson, V. (2010). Outside the closed world: On using machine learning for network intrusion detection. In *IEEE Symposium on Security and Privacy (SP)*, pages 305–316. IEEE.

[Stonebraker et al. 2005] Stonebraker, M., Çetintemel, U., and Zdonik, S. (2005). The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47.

[Stonebraker and Kemnitz 1991] Stonebraker, M. and Kemnitz, G. (1991). The POSTGRES next generation database management system. *Communications of the ACM*, 34(10):78–92.

[Tavallaee et al. 2009a] Tavallaee, M., Bagheri, E., Lu, W., and Ghorbani, A.-A. (2009a). A detailed analysis of the KDD CUP 99 data set. In *Proceedings of the Second IEEE Symposium on Computational Intelligence for Security and Defence Applications*. IEEE.

[Tavallaee et al. 2009b] Tavallaee, M., Bagheri, E., Lu, W., and Ghorbani, A. A. (2009b). A detailed analysis of the kdd cup 99 data set. In *2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications*, pages 1–6.

[Toshniwal et al. 2014] Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J. M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., Bhagat, N., Mittal, S., and Ryaboy, D. (2014). Storm@Twitter. In *ACM SIGMOD International Conference on Management of Data*, pages 147–156. ACM.

[Van Der Maaten et al. 2009] Van Der Maaten, L., Postma, E., and den Herik, J. (2009). Dimensionality reduction: a comparative. *Journal of Machine Learning Research*, 10:66–71.

[Widom 1992] Widom, J. (1992). The Starburst rule system: Language design, implementation, and applications. *IEEE Data Engineering Bulletin*.

[Zaharia et al. 2013] Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., and Stoica, I. (2013). Discretized streams: Fault-tolerant streaming computation at scale. In *XXIV ACM Symposium on Operating Systems Principles*, pages 423–438. ACM.

[Zhai et al. 2014] Zhai, Y., Ong, Y.-S., and Tsang, I. W. (2014). The emerging "big dimensionality". *Comp. Intell. Mag.*, 9(3):14–26.

[Zhao et al. 2015] Zhao, S., Chandrashekar, M., Lee, Y., and Medhi, D. (2015). Real-time network anomaly detection system using machine learning. In *11th International Conference on the Design of Reliable Communication Networks (DRCN)*, pages 267–270. IEEE.

Apêndice
A Ferramenta CATRACA e a Atividade Prática

Nesta seção é apresentada a ferramenta CATRACA, uma ferramenta baseada no processamento de fluxos que utiliza algoritmos de aprendizado de máquina para filtrar tráfego benigno de ameaças e caracterizar o comportamento da rede [Andreoni Lopez et al. 2017b]. Um processo de enriquecimento de contexto permite a visualização em tempo real do comportamento da rede assim como a origem e destino das ameaças. O CATRACA é uma função virtual de rede (*Virtual Network Function* - VNF), que permite o encadeamento com outro tipo de funções virtuais, como por exemplo um *firewall*, para a prevenção de ameaças. Outra facilidade provida pelo CATRACA é a capacidade de migração, que objetiva a sua localização o mais perto possível da origem da ameaça a fim de evitar retardos na detecção.

A arquitetura da ferramenta CATRACA, mostrada na Figura 3.26, é composta por três camadas: captura, processamento e visualização. A **camada de captura**, é responsável pela captura dos pacotes, através do espelhamento de tráfego, pela biblioteca *libpcap*. Uma aplicação, escrita em Python e baseada no *flowtbag*[24], abstrai os pacotes em fluxos, que são definidos como uma sequência de pacotes que possuem a mesma quíntupla IP de origem, IP de destino, porta de origem, porta de destino e protocolo, durante uma janela de tempo. Ao todo, 46 características de cada fluxo são extraídas em tempo real do tráfego que atravessa a VNF e publicadas em uma fila de um serviço produtor/consumidor de mensagens *Kafka*, com baixa latência, para serem consumidos pela camada de processamento.

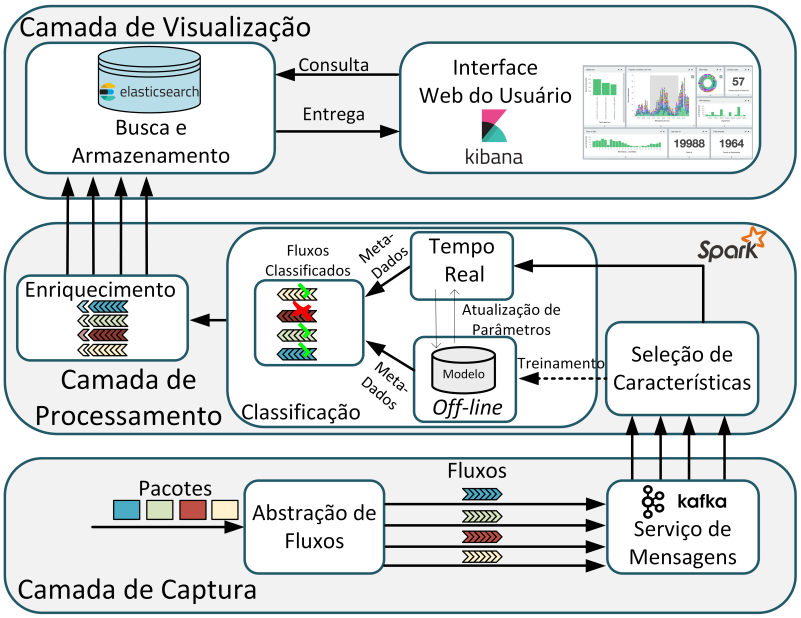


Figura 3.26. A arquitetura em camadas da ferramenta CATRACA: a camada de captura, a camada de processamento e a camada de visualização.

[24]Estatisticas de fluxos *flowtbag*: https://dan.arndt.ca/projects/flowtbag/

A **camada de processamento** é instanciada em uma nuvem dedicada para a classificação que possui o *Spark* como núcleo principal de processamento. A plataforma `Spark` foi escolhida entre as diferentes plataformas de processamento de fluxos por possuir o melhor comportamento em relação a tolerância a falhas [Andreoni Lopez et al. 2016b], tornando a ferramenta CATRACA mais robusta em caso de falha de nós de processamento. O *Spark* é implementado em um aglomerado de máquinas no modelo mestre/escravo, em que os escravos possuem a capacidade de expansão e redução de recursos, tornando o sistema escalável. Uma vez que os fluxos chegam na camada de processamento, um algoritmo seleciona as características mais importantes para a classificação das ameaças [Andreoni Lopez et al. 2017a]. Na etapa de processamento, os dados são enriquecidos através de distintas informações como a localização geográfica dos IPs analisados. Em seguida, os fluxos são classificados em maliciosos ou bem-comportados através de um algoritmo de aprendizado de máquina.

Finalmente, a **camada de visualização** é implementada utilizando a pilha de *software Elastic*[25]. A pilha *Elastic* permite a visualização personalizada de eventos em tempo real. Assim, a saída da camada de processamento é enviada ao componente *elasticsearch* que fornece um rápido serviço de busca e armazenamento. A interface do usuário, que é executada no componente *Kibana* da mesma pilha de *software*, comunica-se com *elasticsearch* através de consultas. O *Kibana* processa os resultados das consultas e gera a visualização final.

Toda a documentação e a instalação da ferramenta CATRACA encontra-se disponível no site `https://www.gta.ufrj.br/catraca`.

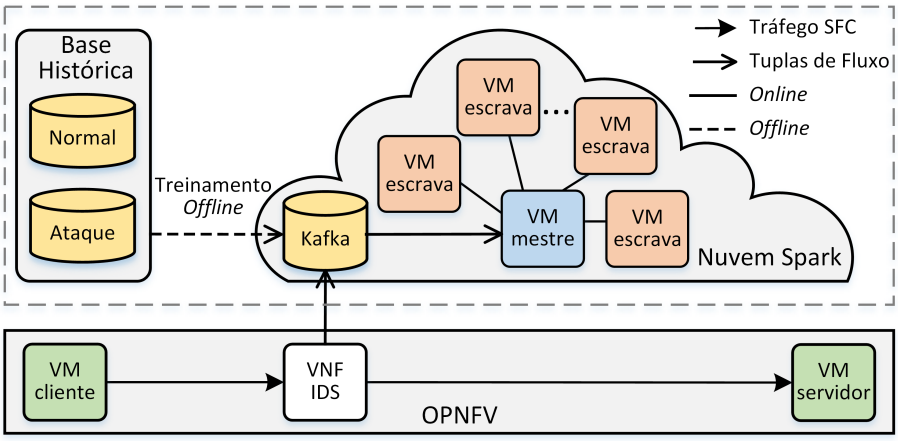


Figura 3.27. Demostração dos experimentos *Online* e *Offline*.

A demonstração da ferramenta CATRACA será efetuada através de dois experimentos como mostra a Figura 3.27. Primeiramente será realizado um experimento *offline*. Para isso, um algoritmo de árvore de decisão é treinado de forma prévia por um conjunto de dados. A segunda etapa é a implementação do modelo de classificação gerado em fluxo e análise da classificação em tempo real do tráfego que atravessa uma máquina virtual, que será executada em um computador pessoal[26]. A captura dos pacotes será enviada

[25] *Elastic Stack*: https://www.elastic.co/

[26] A demonstração requer um computador pessoal com acesso à Internet para ser configurado com os

ao nó central, nó *master*, localizado na nuvem OPNFV no Grupo de Teleinformática e Automação (GTA/UFRJ), no Rio de Janeiro. O nó localizado no GTA é responsável por receber os fluxos provenientes do *Kafka*, gerenciá-los e distribuí-los para os nós escravos do aglomerado *Spark*.

O objetivo da demonstração é evidenciar a acurácia, a agilidade e a escalabilidade da ferramenta na detecção de tráfego malicioso. Além disso, a ferramenta permite a visualização em tempo real de quando é capturado tráfego normal ou de quando é injetado tráfego malicioso através de uma interface de rede na máquina virtual monitorada. O classificador é previamente treinado com os parâmetros do modelo da árvore de decisão. Um conjunto de dados de uma operadora de telecomunicações [Andreoni Lopez et al. 2017c] da cidade do Rio de Janeiro é usado para gerar tráfego na máquina virtual.

Exemplo de Detecção de Ameaças em Tempo Diferenciado (*Offline*)

Conectar-se ao nó mestre do Spark

```
ssh -i cloud.pem ubuntu@<master-ip>,
```

carregar o conjunto de dados escolhido para o *Haddop Distributed File System* (HDFS)

```
hdfs dfs -put dataset.csv /user/app
```

e submeter o programa Python `new-offline.py`[27] no aglomerado. O código `new-offline.py` é o aplicativo que executa a classificação sobre o Spark. Assim, é esse aplicativo carrega os dados do HDFS e realiza a classificação *offline* dos dados.

```
bin/spark-submit -master spark://master:7077
new-offline.py hdfs://master:9000/user/app/dataset.csv
```

Por padrão, o nó mestre do aglomerado responde pela porta 7077 e o arquivo carregado no HDFS se encontra na porta 9000. Os resultados da classificação serão exibidos na tela da seguinte forma:

```
Summary Stats:
Accuracy = 0.990038032436 Test Error = 0.00996196756424
Precision = 0.990038032436 Recall = 0.990038032436
F1 Score = 0.990038032436
```

Além disso, os resultados são armazenados no HDFS, para obtê-los em um arquivo */tmp/Results.txt*:

```
hdfs dfs -getmerge hdfs://master:9000/user/app/Results_<inputfile>
/tmp/Results.txt
```

Exemplo de Classificação em Linha (*Online*) e Visualização

Nesse exemplo, são capturados dados de uma máquina virtual e o tráfego é espelhado para a nuvem Spark localizada no GTA/UFRJ. Na máquina virtual da qual serão capturados os pacotes executar:

---

softwares de teste.

[27] código disponível https://github.com/tinchoa/catraca/blob/master/processing-layer/new-offline.py

`python read_network.py <kafka topic> <kafka-server>`

com os endereços Kafka-topic: Spark e Kafka-server: 10.10.10.3

conectar-se ao Spark Master:

`ssh -i cloud.pem ubuntu@<master-ip>` desde o $SPARK_HOME:

```
spark-submit -master spark://master:7077 -packages
<package elasticSearch> -jars <connector elastic search>,
<connector kakfa> <python code> hdfs://master:9000/user/app/dataset
<kafka server> <kafka topic>
```

- package elasticSearch: TargetHolding:pyspark-elastic:0.4.2
- connector elastic search: /opt/spark/jars/elasticsearch-spark-20_2.10-5.5.1.jar, connector kafka: /opt/spark/jars/spark-streaming-kafka-0-8-assembly_2.11-2.1.1.jar
- código python: detection-with-elastic.py[28],
- kafka server: kafka01:2181,
- kafka topic: spark

para visualizar os dados de classificação e as métricas customizadas em tempo real, acessar a interface Kibana Dashboard, mostrada na Figura 3.28.

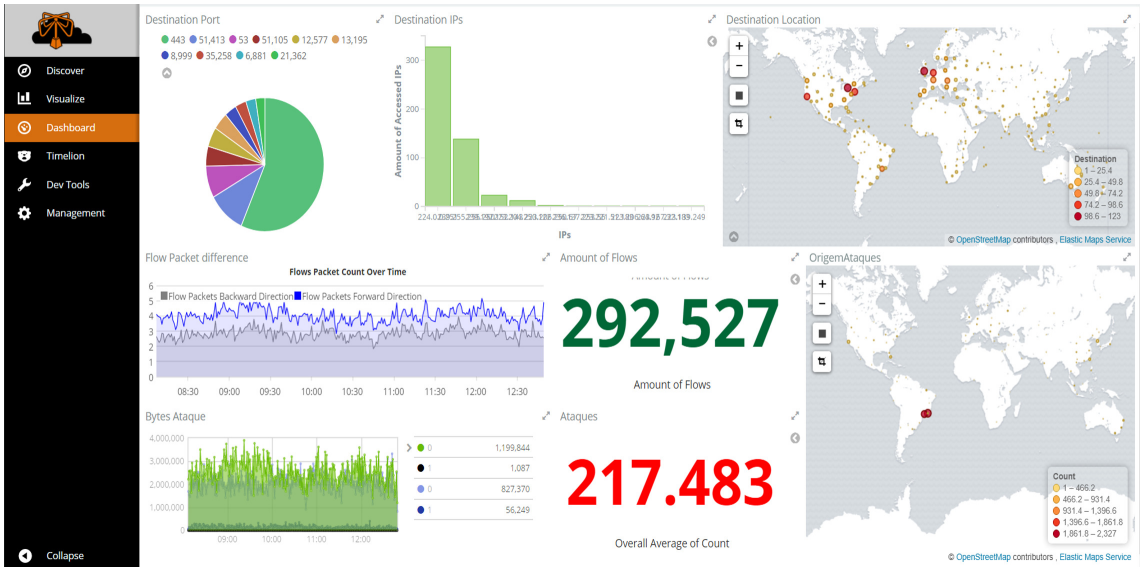


Figura 3.28. Visualização do painel de controle da ferramenta.

[28]disponível em https://github.com/tinchoa/catraca/blob/master/processing-module/detection-with-elastic.py