IME - USP

Detecção de Incidentes de Segurança em Redes por meio de uma Arquitetura para Análise de Grandes Volumes de Dados

Victor Andreas Sprengel
NUSP: 9298002
18 de janeiro de 2020

Daniel Macêdo Batista
Orientador

Resumo

Esse trabalho traz uma perspectiva prática para o problema de detecção de incidentes de segurança em redes de computadores. Serão abordados tanto a captação dos dados - em menor proporpção - como o processamento e análise - como foco.

Baseando-se na *arquitetura Lambda* iremos apresentar uma proposta de arquitetura capaz de suportar uma grande escala de dados a serem utilizados por modelos de aprendizagem de máquina que irão realizar a detecção de incidentes. Passaremos por tecnologias atualmente relevantes como *Apache Kafka* e *Apache Spark*, bem como os cuidados necessários na hora de utilizá-las.

Iremos também trazer o estado da arte de aprendizado de máquina, como algoritmos de *Gradient Boosting*, uma larga escala de dados e o uso de uma biblioteca de código aberto, funcional e confiável para colocar algoritmos de ciência de dados em produção. Serão apresentadas as preocupações de se desenvolver um modelo para essa tarefa desde o treinamento até o monitoramento das decisões tomadas.

Experimentos realizados com o objetivo de validar a funcionalidade esperada da arquitetura e o poder preditivo do modelo idealizado mostraram que há grande potencial para que essa arquitetura tenha bons resultados no cenário real. O poder preditivo do modelo criado para os experimentos foi bom dada a quantidade limitada de dados e o foco dividido entre elaboração da arquitetura e modelagem, enquanto a arquitetura cumpriu a função para a qual foi elaborada, mesmo que num ambiente controlado e pequeno de teste.

Sumário

Capítulo 1

Introdução

Falar de segurança em redes hoje em dia é vital para praticamente todas organizações, que hoje em dia dependem diariamente da disponibilidade e segurança da infraestrutura de rede para suas operações. Muitas dessas organizações estão particularmente preocupadas com segurança, pois lidam diariamente com dados sigilosos ou sensíveis. De fato, qualquer brecha de segurança é um grande potencial de perda de dinheiro e credibilidade. Mas os investimentos para prevenção, detecção e reação a ataques não estão sendo suficientes.

Recentemente uma multibillionária empresa de jogos, a **Blizzard Entertainment** [1], sofreu um ataque de negação de serviço distribuído **DDoS** que impediu vários clientes de acessarem seus jogos durante um final de semana inteiro [2]. Também recentemente, no mês de julho de 2019, um grande banco dos Estados Unidos chamado **Capital One** sofreu um ataque em que o(s) atacante(s) conseguiu(ram) acesso a informações de identidade e cartão de crédito de vários clientes do banco [3].

Tendo em vista a crescente necessidade por melhores barreiras de proteção, temos como aspiração do projeto a vontade de aumentar a segurança de uma rede observando os pacotes que trafegam por seus enlaces. Para isso, iremos

abordar o problema do ponto de vista de um CSIRT (*Computer Security Incident Response Team*), cujo papel é providenciar serviços e plataforma que dizem respeito a prevenção, gerenciamento e coordenação de potenciais emergências de cibersegurança.

Em nossa visão, a solução ideal para isso seria monitorar os enlaces apenas no momento exato em que incidentes de segurança estejam sendo iniciados, armazenar apenas os pacotes relevantes em memória volátil e informar ao analista de segurança sobre o possível incidente em um intervalo de tempo que permita ao analista tomar ações que inviabilizam que o incidente de fato tenha sucesso. Em memória não volátil, seria armazenado somente o mínimo necessário de dados que possibilitasse a realização de investigações posteriores e de aprendizado automático.

Dessa solução possivelmente utópica (porém, sendo almejada) este projeto se responsabiliza unicamente pela parte de detecção dos incidentes: utilizando os dados (pacotes de redes) já disponibilizados em uma fila do Kafka, será utilizado o Spark Streaming para processar esses dados e por fim criar modelos de aprendizado de máquina capazes de detectar um incidente em tempo quase real. Faz parte do escopo, portanto, escolher incidentes de rede para serem estudados, reproduzidos e testados contra o modelo a ser criado.

Um dos pontos principais do projeto será considerar ataques recentes, de modo que modelos capazes de detectá-los eficientemente sejam de utilização prática.

1.1 Motivação

A maioria dos sistemas de detecção de intrusões em redes funcionam majoritariamente com regras manualmente criadas ou com detecção de anomalias. Escrever essas regras é extremamente custoso e requer conhecimento profundo do problema ou da área. No caso de detecção de anomalias, acontecem

muitos falsos positivos [4], que geram trabalho de depuração desnecessário.

Quanto à possibilidade de falsos positivos, é "melhor prevenir do que remediar", e já existem projetos que mostram o uso de inteligência artificial para segurança de informação [5], sendo um método eficiente em três sentidos: rápida detecção, menor armazenamento de pacotes de rede (em memória volátil ou não) e capaz de criar soluções mais genéricas do que um conjunto de regras.

1.2 Organização do texto

Esta monografia está organizada da seguinte forma: O Capítulo 1 age como introdução ao problema e à dissertação. Os próximos três capítulos, de números 2,3 e 4 se preocupam com a inspiração para a criação da arquitetura da solução, sua definição no contexto e sua implementação utilizando tecnologias e ferramentas atualmente relevantes. Os capítulos 5,6 e 7 cobrem todas as preocupações e escolhas relecionadas ao modelo de aprendizagem de máquina desde sua concepção, depois utilização e também monitoramento. O penúltimo capítulo - de número 8 - traz os resultados dos experimentos e também as conclusões quanto à utilização de nossa arquitetura e modelo propostos para atingir o propósito final. Por fim, o capítulo 9 é o último e deixa esclarecidas três vias nas quais a solução ou a compreensão quanto ao problema podem ser aprofundados em um trabalho futuro.

Capítulo 2

Arquitetura Lambda

Uma das duas principais contribuições dessa dissertação é a estruturação de uma arquitetura para o problema em mãos - a outra sendo o guia de como se criar um modelo de predição de incidentes de rede. Essa estruturação é feita em cima de uma ideia que hoje em dia é conhecida como *arquitetura Lambda*, que representa uma arquitetura de processamento de dados que é genérica, escalável e tolerante a falhas [7].

A **AL** (arquitetura lambda) pode ser resumida, de uma perspectiva alto nível, a três **camadas** que ficam entre duas pontas do fluxo de dados. Uma ponta é a **entrada de dados novos**, onde todos os dados coletados são orquestrados e despachados para as camadas de processamento. A outra ponta são as **consultas**, que representam o usuário da aplicação que está interessado no resultado do processamento de dados. De forma mais detalhada, a primeira camada de processamento é a **camada de lotes** (*batch layer*), onde duas coisas acontecem: em primeiro lugar, todo dado novo deve ser persistido em um "conjunto de dados mestre", onde os dados são imutáveis, configurando um cenário de acrescentar linhas numa tabela, sem alterar ou deletar as antigas. Em segundo lugar, os dados são agregados e pré-processados para deixar pronto o que será disponibilizado de informação ou visualização no futuro. O resultado desse pré-processamento fica guar-

dado na **camada de serviço** (*serving layer*), que é responsável por indexar as agregações de modo a tornar as consultas rápidas.

A característica de dados imutáveis é especialmente importante para aplicações de ciência de dados, pois torna possível saber qual era o estado de alguma informação em **qualquer instante do passado**. Um exemplo de banco de dados desenvolvido para atender esse tipo de uso é o **Datomic** [8].

A tereceira camada da arquitetura é a **camada de velocidade** (*speed layer*), que é responsável por compensar a latência de processamento da camada de lotes, utilizando informações em tempo real para gerar visualizações ou informações.

2.1 Por que arquitetura lambda?

Não iremos cobrir por completo todas as vantagens (e críticas) à arquitetura Lambda, pois existe material extenso e completo para tal, como o livro *Big data: Principles and best practices of scalable realtime data systems* do idealizador da arquitetura **Nathan Marz** [9]. Iremos, porém, trazer duas perspectivas do por que usar a **AL**.

Em primeiro lugar, é uma arquitetura que está sendo inspiração para resolver problemas atuais e complexos. De fato, o **Yahoo** usa uma solução baseada na **AL** com *Apache Storm*, *Apache Hadoop* e *Druid* [10]. O **Netflix** também se inspirou na ideia e expandiu nela para criar sua própria *pipeline* de dados [11].

Em segundo lugar, é uma arquitetura que aproveita as vantagens de processamento em lote (maior confiabilidade nos dados, maior tolerância a falhas e capacidade de agregação de grandes quantidades de dados) enquanto compensa sua maior desvantagem - a latência - com a camada de velocidade. Ela faz isso consolidando e organizando as informações do processamento

em lote na camada de serviço e utilizando ferramentas de processamento robusto e distribuído em cada etapa. É uma grande combinação de técnicas que já eram utilizadas, porém em contextos diferentes.

2.2 No contexto de detecção de incidentes de rede

Nessa seção iremos descrever como cada uma das camadas da arquitetura lambda está representada na nossa ideia de arquitetura para o problema e, portanto, nos experimentos.

Nós teremos uma fila de mensagens agindo como entrada de dados, onde todo e qualquer dado seria colocado na fila para processamento por ambas camadas de lote e velocidade. Nesse contexto em específico, a fila recebe pacotes de rede (tanto conteúdo quanto metadados) interceptados por alguma ferramenta como o *Wireshark*, um *sniffer* de rede bastante utilizado.

A camada de lote realiza o processo de guardar, processar e gerar algum tipo de resultado dessa agregação dos pacotes. Nesse sentido iremos utilizar todos os dados coletados no passado para criar um modelo de aprendizado de máquina. Esse modelo será serializado, possibilitando que suas previsões possam ser feitas em instantes. Idealmente também guardaríamos todos os dados novos em algum banco transacional, mas essa parte não será abordada pois usaremos um conjunto de dados pronto para realizar os experimentos.

A camada de velocidade será responsável por fazer as agregações do fluxo de dados vindo em tempo real e deixá-los no formato necessário para o modelo serializado fazer sua predição.

As consultas e a camada de serviço não são perfeitamente representadas na nossa arquitetura. Não estamos pensando na gama de informações que análises sobre pacotes de rede trariam para que haja necessidade de um indexador desses resultados. Também não faz sentido consolidar a informação

sobre um possível ataque à rede para que alguém faça essa pergunta no futuro, pois queremos avisar o quanto antes de que algo anormal está acontecendo. Por isso substituímos essas duas camadas por um *daemon* que frequentemente compara o resultado da camada de lote (o modelo serializado) com o resultado da camada de velocidade (entrada para a previsão do modelo) para avisar se um possível incidente foi detectado.

Capítulo 3

Tecnologias e ferramentas

Neste capítulo iremos justificar nossas escolhas de tecnologia e introduzir suas funções dentro da arquitetura proposta.

3.1 Apache Kafka para fila de mensagens

Apache Kafka é uma plataforma de código aberto para processamento de fluxos de dados desenvolvida pela **Apache Software Foundation**, escrita em Scala e Java. O Kafka tem como objetivo fornecer uma plataforma unificada, de alta capacidade e baixa latência para tratamento de dados em tempo real [12]. Uma simplificação é imaginar o Kafka como um grande *log* particionado em **tópicos**. Nesse sentido, cada um dos tópicos age como uma fila ordenada pelo tempo.

O desenvolvimento do Kafka começou primeiro no **LinkedIn** antes de passar a ser uma tecnologia de código aberto que foi adotada pela **Apache Software Foundation** [12]. Esse fato é relevante pois desde o começo seu desenvolvimento foi motivado pelo uso prático.

É interessante também citar sua escolha de nome. É chamado *Kafka* como referência e homenagem ao escritor Franz Kafka, por que é **otimizado para**

escrita. Isso significa que seu processamento distribuído é desenhado para tornar fácil a escalabilidade quando é necessário coletar, armazenar e deixar pronto para análises uma grande quantidade de dados.

Nossa escolha da ferramenta para ser a porta de entrada para a arquitetura foi baseada na necessidade de interceptar pacotes de rede de várias máquinas de modo que seja fácil analisar e dar um resultado em *tempo quase real*, o que se encaixa no propósito e otimização do Kafka. Sua implementação garante consistência (um pacote não processado volta para a fila na ordem correta) e é fácil de ser utilizada, pois é uma tecnologia amplamente conhecida com fácil integração a ferramentas de processamento de fluxos de dados via a interface de **Consumers**. Por fim, seu código é aberto, tornando possível ler o mesmo, corroborando para o ponto de fácil uso e acesso.

Para entender nosso uso, basta entender que existem aplicações responsáveis por **publicar** os dados no *Kafka* - os **Producers** - e que há aplicações responsáveis por **consumir** esses dados - os **Consumers**. De fato, não criaremos uma aplicação com **Producers**, o que faremos será criar um script em *python* para repassar cada pacote de rede como mensagem em um tópico do Kafka e usar uma ferramenta de processamento de fluxos de dados para consumir as mensagens. Essa ferramenta será o **Apache Spark**.

3.2 Apache Spark para a camada de velocidade

Apache Spark é um framework de código fonte aberto para computação distribuída. Foi desenvolvido no AMPLab da Universidade da Califórnia e posteriormente repassado para a Apache Software Foundation que o mantém desde então. Spark provê uma interface para programação de clusters com paralelismo e tolerância a falhas [13].

O que isso significa na prática é que com Spark é possível manipular dados tabulares de maneira distribuída e com tolerância a falhas, permitindo

manipulação de uma quantidade enorme de dados que não seria trivial normalmente.

Na versão ideal de nossa arquitetura, a enorme quantidade de pacotes de rede a serem processados e que devem virar uma tabela única a ser utilizada como entrada de algoritmos de aprendizado de máquina poderiam ser gerados via Spark, configurando o uso da ferramenta também na camada de lote. Como já estamos utilizando um conjunto de dados pronto para nosso experimento, isso não será necessário.

Consideramos a ferramenta uma ótima escolha para a arquitetura pois ela nos permite lidar com quantidades enormes de dados, permitindo escalar a solução tanto horizontalmente (número de máquinas) como verticalmente (capacidade das máquinas), além de ser de fácil uso, pois é amplamente utilizada e conhecida além de se integrar bem com o *Kafka*, até por que são ambas tecnologias do grupo Apache. Por fim, a ferramenta nos traz duas abstrações importantes: Spark streaming e DataFrames.

3.2.1 Spark streaming

Acabamos de mencionar que o **Apache Spark** é ótimo para lidar com dados tabulares. Porém, escolhemos a ferramenta para a camada de velocidade, onde não há dados tabulares à primeira vista. De fato, mencionamos que a ideia é interceptar os pacotes de rede conforme eles são enviados, mas precisamos que eles estejam disponíveis como tabela para que sejam entrada para um modelo de aprendizado de máquina, tanto na hora de treinamento quanto de predição.

Spark streaming é uma extensão do núcleo da **API** do Spark que permite ingestão de fluxos de dados como se fossem dados tabulares - permitindo que computações complexas sejam expressas por funções de alto nível como *map, reduce, join* e *window* [14]. Além disso, várias fontes de fluxos de dados são

suportadas, inclusive filas do Kafka. Após processamento, os dados podem ser armazenados em sistemas de arquivos, bancos de dados ou *dashboards*.

3.2.2 DataFrames e SparkSQL

Como uma outra extensão do Spark, usaremos também **SparkSQL** e seus **DataFrames**. O que isso significa é que todo dado tabular será exposto como se fosse uma tabela de banco de dados. Na prática veremos a noção de linhas, colunas e *queries*. Isso permite uma interpretabilidade muito maior do código. Podemos por exemplo escrever:

```
stream.where(col("received_at").cast("date") === current_date())
```

para garantir que estamos analisando apenas pacotes recebidos hoje.

3.3 Uma nota sobre o spark streaming

Mencionamos que usaremos a abstração do **SparkSQL** para lidar com o *streaming* como se fosse um dado tabular. Mas então o que seriam as linhas da tabela que criaremos? A primeira opção - talvez a mais intuitiva - seja que cada pacote de rede corresponda a uma linha. O problema com isso é que nosso modelo se limitaria a dizer se **cada** pacote de rede faz parte de um ataque. Na prática, porém, os ataques que estudaremos serão mais complexos e é necessário observar vários pacotes - com diferentes origens e destinos - para que seja possível chegar a uma conclusão. Essa opção é, portanto, imediatamente descartada.

Para lidar com esse problema nós agregaremos os dados de vários pacotes para formar cada linha. Muito alinhado com o objetivo de que a detecção do incidente aconteça rapidamente, decidimos por fazer a detecção com base em intervalos vindos de uma janela deslizante. Desse modo teremos a agregação de todos os pacotes recebidos nos últimos 10 minutos - atualizada a cada 60 segundos. Agora é possível prever se a rede que estamos protegendo está sendo atacada nos últimos 10 minutos, sendo possível perceber um novo ataque em no mínimo 60 segundos. A Figura 3.1 mostra esse comportamento.

O tamanho dos intervalos citados são iniciais e podem ser modificados de acordo com o que descobrirmos durante o desenvolvimento.

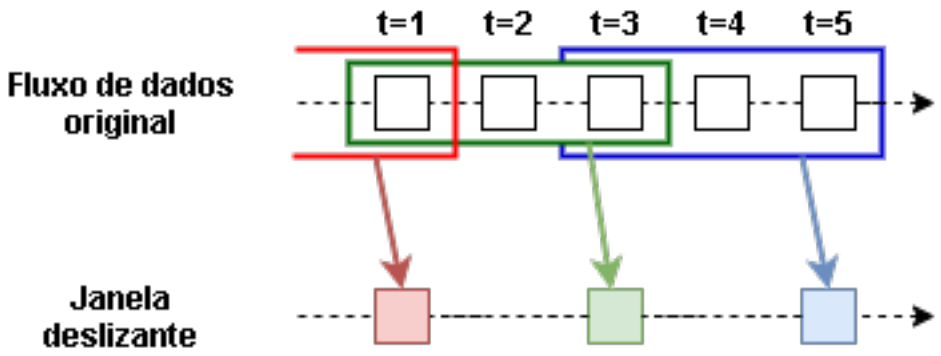

Figura 3.1: Ilustração do comportamento de uma janela deslizante

3.3.1 Cenário real

Segundo a explicação sobre janela deslizante e de acordo com os intervalos que estabelecemos, tudo indica que a cada 60 segundos receberemos uma agregação dos últimos 10 minutos e basta tomar a decisão naquele momento para detectar se um incidente está acontecendo ou não.

A explicação acima é bem simples, mas é necessário destrinchar os cenários possíveis para entender a seguinte afirmação: **Não estaremos sempre olhando para apenas uma janela**.

Suponha por exemplo, seguindo ainda a Figura 3.1, que um pacote de rede chega no momento 3. Esse momento, como delimitado na ilustração, faz parte de duas janelas. Logo, teremos atualização de duas janelas.

A Figura 3.2 traz uma segunda ilustração de fluxo de dados com janela deslizante em que mais de uma janela é atualizada.

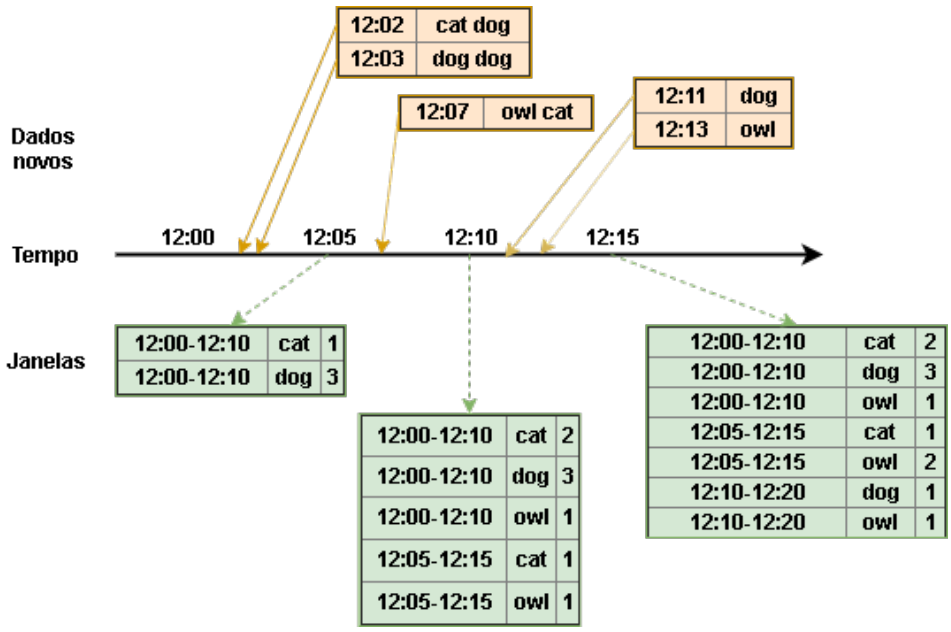

Figura 3.2: Atualização de mais de uma janela sendo demonstrada

3.3.2 Mudança de opinião

De fato, a todo momento estaremos recebendo a informação de **todas janelas que foram atualizadas**. Para complicar ainda mais a situação, nós comentamos várias vezes o quão robusto é o **Apache Spark**, mas ele não deixa de ser um sistema distribuído que está exposto às mais diversas falhas. Logo, o que acontece quando um pacote que foi enviado e captado às 12:04 só é processado às 12:11? A resposta é que ele vai atualizar corretamente as janelas que contém o momento 12:04. O que permite isso é a integração que existe junto ao **Kafka**, de que toda mensagem em sua fila vem acompanhada do exato horário de captura, mesmo que venha a ser processada só muito depois.

Por fim, como podemos decidir se temos a "agregação final"de uma janela, ou se ainda existe algum pacote atrasado que chegará para uma atualização final? A resposta é que, infelizmente, isso é impossível de se saber. Ainda

assim, a **API** do Spark nos permite chegar a uma resposta para esse problema com o conceito de **Watermarking**.

Watermarking nada mais é do que escolher uma tolerância máxima para atrasos. No nosso exemplo, podemos dizer que se algum pacote de rede está atrasado há dois minutos ou mais, ele deve ser ignorado, e as janelas que contém o momento no qual esse pacote foi capturado não serão mais atualizadas **por esse pacote**. A Figura 3.3 traz uma terceira ilustração da nossa proposta de *streaming*, contemplando o caso de um pacote chegar atrasado.

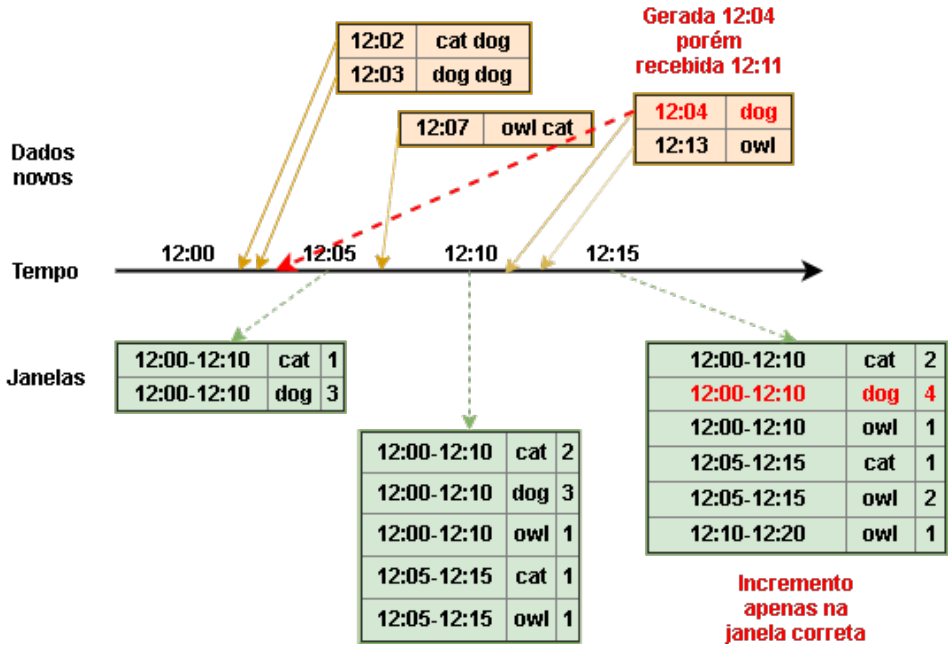

Figura 3.3: Atualização de uma janela causada por atraso

O que isso tudo significa para nosso sistema é que nós teremos que avaliar a mesma janela várias vezes - de fato, toda vez que ela sofrer uma atualização -, e que é possível que nosso sistema de detecção **mude de opinião**. Ou seja, informar que segundo uma janela de tempo a rede está sob ataque, mas em um segundo momento decidir o contrário. Isso não é muito problemático a princípio, desde que nós tenhamos isso em mente na hora de implementar

nosso sistema.

3.4 Python e suas bibliotecas como camada de lote

Por fim, como última ferramenta que iremos utilizar e que merece uma descrição, temos aquela necessária para a criação do modelo de aprendizado de máquina, que é a nossa camada de lote. Mais detalhes sobre o algoritmo em si serão apresentadas adiante, e essa seção é sobre as bibliotecas que usaremos para isso.

Nós escolhemos usar **Python** para a criação dos modelos. Porém, essa escolha tem mais a ver com as ferramentas que essa linguagem nos proporciona do que com a linguagem em si. É até estranho defender essa escolha, visto que ela é de longe a mais comum hoje em dia para esse propósito [15], mesmo que outras linguagens como **R** também tenham implementação dos melhores e mais famosos algoritmos de aprendizado de máquina. Nas próximas subseções vamos listar e justificar as três ferramentas que julgamos essenciais no nosso uso de **Python**

3.4.1 scikit-learn

Uma das bibliotecas de aprendizado de máquina mais famosa da linguagem é código-aberto, simples, eficiente e contém implementação de todos algoritmos, filtros e técnicas que precisamos [16]. A escolha vem desses fatos, da familiaridade já existente com a ferramenta e com a integração com outra ferramenta que será citada posteriormente.

3.4.2 pandas

Como não escolhemos por construir a camada de lote *ideal* - com um conjunto de dados mestre e agregações ou pré-processamentos persistidos, possivelmente usando **Spark** - pois escolhemos apenas criar um modelo serializado, será necessário fazer as mesmas agregações e transformações que

a nossa camada de velocidade fará. O motivo é que a entrada para treinamento do modelo e os dados que serão avaliados devem ter o mesmo formato. Talvez contra-intuitivamente, o trabalho de agregação vem efetivamente em dobro - uma vez em Spark Streaming e outra em Python.

Para tornar essa tarefa o mais fácil possível, escolhemos utilizar uma biblioteca em Python que - apesar de não ser tão eficiente e versátil quanto o Spark - utiliza a mesma API de DataFrames para lidar com dados tabulares. Além disso, dados no formato de tabela do `pandas` servem de entrada para os algoritmos do `scikit-learn`.

3.4.3 fk-learn

Por fim, já mencionamos que a biblioteca `scikit-learn` irá providenciar a implementação dos algoritmos de aprendizado de máquina. Porém, uma recentemente publicada biblioteca de código aberto da empresa de pagamentos **Nubank** traz uma interface para o uso do `scikit-learn` [17]. O nome dessa biblioteca, `fklearn`, vêm de `scikit-learn` e de **functional**. Nas palavras do próprio repositório:

"*fklearn usa o princípio de functional programming para tornar mais fácil a solução de problemas reais com Machine Learning.*"

Seus princípios são:

- Validação deve refletir cenários reais;
- Modelos em produção devem bater com modelos validados;
- Deve ser muito fácil deixar modelos prontos a serem colocados em produção;
- Reprodutibilidade e análises profundas dos resultados devem ser fáceis;

O jeito mais resumido de se apresentar o `fklearn` é que ele torna mais fácil e mais robusto o uso da biblioteca `scikit-learn`. De fato, ela é uma

interface que não expõe maneiras que não acha corretas de se realizar uma tarefa, trazendo uma maneira confiável de se utilizar modelos de aprendizado de máquina em produção.

Capítulo 4

Arquitetura

Agora que já destrinchamos todas as ferramentas no nosso arsenal para atacar o problema e também as peculiaridades do Spark streaming, iremos esclarecer todas as partes da nossa solução. Isso é o que estamos chamando de **Arquitetura**.

A nossa solução - levando em consideração o escopo que iremos atacar, que deixa de fora a captura dos pacotes - está composta de duas partes bastante distintas.

Primeiro é necessário construir o modelo de aprendizado de máquina. Usaremos um conjunto de dados com pacotes de rede já capturados para treinar e validar um modelo. O entregável dessa parte é somente um arquivo **pickle**, que é o resultado de uma serialização de um objeto Python - no nosso caso, a função de predição que resulta da aprendizagem.

Depois validaremos a nossa ideia de detecção em tempo real. Para isso vamos colocar artificialmente pacotes de rede do mesmo conjunto de dados em filas do Kafka, que serão processadas via Spark Streaming. Após agregação, um **daemon** será responsável por avaliar cada uma das janelas utilizando o `.pickle` gerado na primeira parte.

Vamos descrever essas duas partes um pouco mais a fundo nas próximas seções.

4.1 Criação do modelo

A Figura 4.1 esquematiza como se dará o fluxo de desenvolvimento dessa parte. Cada quadrado é como uma "entidade", ou melhor, um estado dos dados. Cada seta representa uma transformação dos dados até chegarmos no que nos interessa - o modelo serializado.

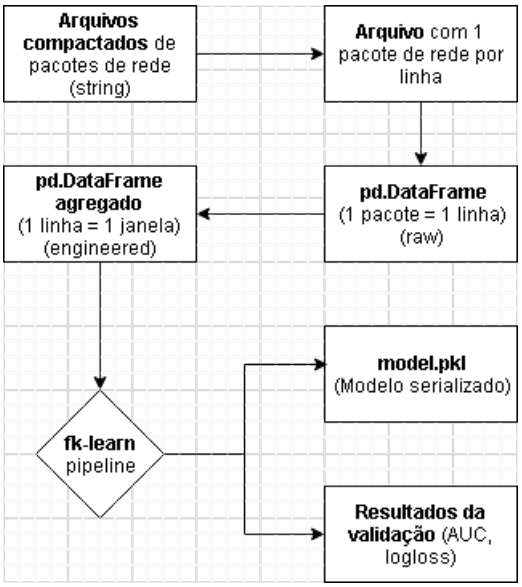

Figura 4.1: Fluxo de desenvolvimento do modelo

Começamos com o nosso conjunto de dados de pacotes de rede, que (como será devidamente apresentado no **capítulo 5**) é um conjunto de dados já pronto que iremos utilizar para a nossa instância do problema, e o jeito no qual ele é disponibilizado dita os primeiros passos do desenvolvimento.

A primeira dificuldade é que temos várias pastas, com vários arquivos `.zip`,

que contém arquivos `.pcap`[1], e cada um deles tem vários pacotes. A primeira tarefa é juntar tudo isso em um arquivo `.txt` onde cada linha é um pacote.

Feito isso, precisamos repassar o resultado para um `pd.DataFrame` (interface de tabelas do pandas). Ou seja, precisamos analisar cada pacote (uma `string`) gerando uma linha com suas colunas.

Agora que temos todas as informações já como objetos python, a tarefa é agregar os pacotes de acordo com nossas janelas deslizantes e as *features* de entrada do modelo.

O importante dessa parte, após agregação, é que devemos dar a classificação desejada para cada linha do nosso DataFrame. A regra para isso será colocar a categoria `not_incident` para todas as janelas de tempo em que ataques não estão sendo realizados. De fato, num primeiro momento diremos só se cada pacote faz ou não parte de um incidente, e testar a classificação binária, e num segundo momento tentar um multiclassificador para nos dizer qual incidente está sendo pretendido. Ficaremos com a abordagem que nos dá melhor resultado na validação.

Com os dados já prontos para servirem de entrada para um modelo, temos que criar uma *pipeline* do `fklearn`. O importante agora é entender que a saída dessa *pipeline* - e dessa parte - são ambos o modelo serializado e os resultados de desempenho.

4.2 Detecção em tempo real

A arquitetura real do projeto fica nessa segunda etapa, onde teremos *clusters* do Kafka e do Spark rodando em tempo real e simularemos os pacotes

[1] Arquivos `.pcap` são resultantes de capturas utilizando a biblioteca libpcap. O Wireshark, que utiliza essa biblioteca, gera arquivos `.pcap`

de rede sendo interceptados para testar se é uma ferramenta viável para cenários reais. A Figura 4.2 ilustra todas as partes do sistema. Tudo começa com a responsabilidade de "interceptar pacotes de rede"(que nesse cenário será simulado, pegando pacotes de rede do *dataset*) e agir como **producer**, armazenando os pacotes como mensagens em tópicos do Kafka. Será um script `.py` que ficará lendo um arquivo com os pacotes e simulando uma passagem de tempo.

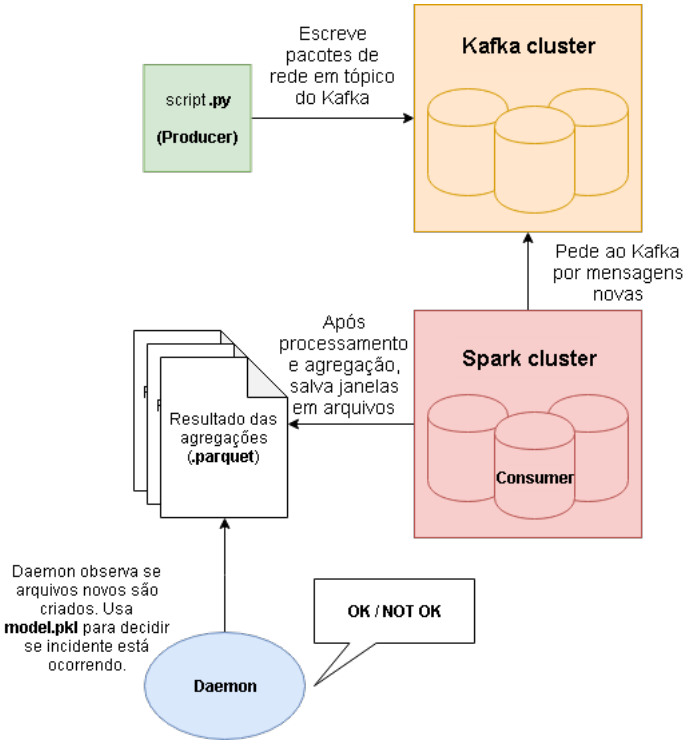

Figura 4.2: Sistema de detecção

Em seguida, nosso sistema usará o Spark Streaming para **consumir** essas mensagens, agregar os dados em janelas e deixar tudo no formato pronto para ser avaliado pelo nosso modelo. Feito isso, o resultado será salvo em um arquivo `.parquet`.

Por fim, haverá um **daemon** (na prática, outro script `.py`) que observará

a aparição de arquivos novos, lerá eles e decidirá usando o `model.pkl` se algum incidente está ocorrendo ou não. Ele será responsável por alertar, no caso positivo.

Capítulo 5

Construção do modelo

Esse capítulo se dedica a esclarecer todo o processo de criação do modelo. Os resultados do desempenho do modelo serão abordados num capítulo futuro.

5.1 Dados

Decidimos logo de início que nosso conjunto de dados deve conter dados realistas e recentes de pacotes de redes de ataques realizados, juntamente com tráfego de "não-ataque".

Boa parte dos conjuntos utilizados para pesquisa nessa área são muito antigos, vide **KDD-99**, amplamente utilizado em artigos científicos até hoje mas coletado em 1999. Justamente por esse motivo queremos trazer algo mais recente para nosso trabalho. O cenário ideal é termos os pacotes inteiros e sem modificações, no máximo com anonimização mas sem perder dados dos cabeçalhos. O mais comum, mas também ideal nesse sentido, é que os dados estejam no formato `pcap`, um formato utilizado para armazenar pacotes de redes e suportado pelos principais analisadores de tráfego como o `tcpdump` e o `wireshark`.

O conjunto de dados que utilizaremos será o **CSE-CIC-IDS2018** [6], que

contém todos os pacotes de rede recebidos e enviados de um conjunto de máquinas, tanto em períodos "normais"quanto em períodos onde há uma tentativa de invasão em andamento. Ele satisfaz todos requisitos que haviamos colocado. Esse conjunto de dados é resultado de um projeto realizado entre o *Communications Security Establishment* (CSE) e o *Canadian Institute for Cybersecurity* (CIC). As capturas foram realizadas em 2018 utilizando hardware e software realistas e representativos do que se utiliza até os dias de hoje, tanto em termos de protocolos quanto em termos de versões.

5.2 Alvo da predição

Como já brevemente descrito na seção anterior, o alvo do modelo será - num primeiro momento - se naquela janela de tempo está acontecendo uma tentativa de invasão ou não. Isso nos dá um problema de classificação binária. Num segundo momento, diremos **qual** tipo de invasão está sendo tentada e testar o desempenho de um multiclassificador.

5.3 Algoritmo de predição

Uma ideia inicial do projeto era tentar primeiro um algoritmo mais simples e tradicional para termos noção de desempenho. Um candidato comum para classificação seria o uso da **regressão logística**. É um modelo bem conhecido, presente em qualquer livro introdutório de aprendizado de máquina, com boa interpretabilidade de suas decisões. Porém, nós descartamos essa ideia e preferimos usar o **LightGBM**. Ele é um algoritmo de **gradient boosting** (a ser explicado na seção seguinte) que está mostrando excelente desempenho para dados tabulares - também nesse domínio que estamos estudando. O motivo do descarte está relacionado às dificuldades de se utilizar a regressão logística num cenário real. Por ser um algoritmo que se baseia puramente na distância de vetores, temos que nos preocupar com:

- Valores faltando (*missing values*)

- Escala das *features*
- *Outliers*
- *Features* categóricas

Devido a ser baseado em árvore, tipo de algoritmo que não olha para a distância de vetores, mas também pelos detalhes de implementação, usar o **LightGBM** significa parar de se preocupar com tudo isso acima, pelo menos parcialmente.

5.4 Gradient Boosting

Essa seção se preocupa em introduzir o tema de *gradient boosting* de modo a justificar a escolha do algoritmo e sua implementação (LightGBM). Para isso elaboraremos uma explicação geral do tópico e depois elencaremos fontes onde há um aprofundamento maior do assunto.

5.4.1 Introdução à *gradient boosting*

Boosting é uma técnica para algoritmos de aprendizagem de máquina, tanto para regressão quanto para classificação. A ideia central é de que o modelo preditivo é composto pela junção de vários outros modelos, de poder preditivo menor e mais específico, chamados de *weak learners*. A definição mais precisa de um *weak learner* é um modelo cujas predições estão apenas levemente correlacionadas com o alvo, ou seja, marginalmente melhores do que uma escolha aleatória. O modelo final, resultado da composição, é chamado de **strong learner**. Além disso, é comum nessa categoria de algoritmos que a construção do modelo final seja iterativa, crescendo a cada etapa.

Gradient boosting, como o nome sugere, é um algoritmo de *boosting* e herda todas essas características, mas especifica como os *weak learners* devem ser construídos. Todos devem ser **árvores de decisão**, de profundidade máxima pequena, a serem construídas de modo a minimizar os residuais

(diferença entre alvo e predição) dessa iteração, ou seja, utilizando a técnica conhecida de **gradiente descendente** numa função de perda apropriada.

5.4.2 Referências para aprofundamento

Para uma curta introdução ao meta-algoritmo de *boosting*, existe um artigo chamado **A Short Introduction to Boosting** [19] que explica a teoria por trás de *boosting* usando como exemplo um algoritmo mais tradicional, o **AdaBoost**.

O segundo passo seria entender como o *gradient boosting* aplica as técnicas descritas para o domínio de algoritmos envolvendo árvores e gradiente. O artigo **A Gradient Boosting Machine** [20] introduz essa noção ao mundo, mas ainda deixa pontas soltas no sentido de detalhes de implementação e otimizações (de velocidade ou memória).

Por fim, o artigo que apresenta o algoritmo **LightGBM** [21] discute a implementação do algoritmo, como se difere das outras implementações e otimizações que foram implementadas. É particularmente interessante como essa é uma das poucas implementações que nativamente suporta *features* categóricas, e o procedimento para isso.

5.5 *Features*

A construção das *features* do modelo teve duas etapas. Primeiro foi necessário transformar cada pacote de rede capturado para uma tabela, e já nesse ponto tivemos que escolher quais informações presentes no arquivo `.pcap` queremos colocar na tabela, e como. O segundo passo foi agregar os vários pacotes de rede em janelas (de acordo com o alvo da predição) e decidir como agregar as informações de cada pacote para informações que o modelo usará.

O pacote de rede no formato `.pcap` traz as informações sobre todas as camadas de rede[1] pelas quais passou. Muitas dessas camadas são raríssimas de serem utilizadas, ou trazem informações pouquíssimo relevantes. O primeiro passo foi listar todas as camadas que aparecem no conjunto completo de dados (no total são 54), depois observamos quais camadas apareciam em pelo menos 1% dos pacotes, de modo a poupar trabalho que traria pouco poder preditivo ao modelo, e finalmente observamos as camadas restantes para entender quais informações eram relevantes por pacote.

As camadas das quais extraimos informações foram `frame`[2], `eth`[3], `ip`[4], `tcp`[5], `data`[6] e `ssl_record`[7]. Apesar de não termos informação sobre as outras camadas, nós anotamos para cada pacote se ela estava presente. Também evitamos o uso de qualquer endereço IP em específico para evitar que o modelo se apoie em IPs que muito provavelmente não serão os mesmos em outras instâncias de incidente na rede. O conjunto de *features* para os **pacotes** é o seguinte:

[1] No jargão do formato `.pcap`, o termo camadas de rede não se refere às cinco camadas do TCP/IP, mas sim a todos os protocolos pelos quais cada pacote passa

[2] Informacões quanto ao quadro de dados, ou seja, sobre o próprio pacote

[3] Informações sobre utilização do protocolo *ethernet*

[4] Informações sobre utilização do *internet protocol* (**ip**)

[5] Informações sobre utilização do protocolo **TCP** na camada de transporte

[6] Informações sobre o *payload* enviado junto aao pacote

[7] Informações sobre os *records* a serem utilizados no protocolo de criptografia

feature	tipo	descrição
is_sender	numérica	1 se pacote foi enviado, 0 c.c.
layers_count	numérica	número de camadas de protocolo presentes
<layer>_present	numérica	1 se protocolo está presente, 0 c.c
frame_timestamp	timestamp	Horário de chegada/envio
frame_length	numérica	Tamanho do quadro
ip_header_length	numérica	Tamanho do *header* do pacote no protocolo IP
tcp_length	numérica	Tamanho do conteúdo passado via TCP
ssl_record_length	numérica	Tamanho do *record* a ser usado em SSL
ssl_record_handshake_length	numérica	Tamanho do *handshake* para estabelecer conexão SSL
data_length	numérica	Tamanho do *payload*
eth_type	categórica	Tipo de protocolo *ethernet*
ip_version	categórica	Versão do protocolo **IP** utilizado
tcp_srcport	categórica	Porta utilizada na fonte do pacote para TCP
tcp_dstport	categórica	Porta utilizada no destino do pacote para TCP
ssl_record_content_type	categórica	Tipo de conteúdo do *record* da camada SSL

Na hora de agregar os pacotes em suas respectivas janelas de tempo, tivemos que consolidar essas informações, e o tipo de consolidação variou com a *feature*. Para todas as colunas numéricas, nós utilizamos a média, desvio padrão e valor máximo. Para todos os valores categóricos, nós calculamos o número de categorias únicas na janela, e a porcentagem de pacotes que tinham a categoria mais comum. Quanto às camadas, criamos uma coluna para cada possível camada com a porcentagem de pacotes que tinham passado por aquela camada. Também foi incluído na agregação o número absoluto de pacotes que estava contribuindo para aquela janela. Por fim, incluimos o valor de **todas** as *features* citadas acima duas janelas (10 minutos) atrás e seis janelas (30 minutos) atrás, efetivamente triplicando o número de *features*, chegando no número total de 276[8]

[8]Numéricas: $9 * 3 = 27$. Categóricas: $5 * 2 = 10$. Camadas e contagem: $1 + 54 = 55$. Total considernado $lags = (27 + 10 + 55) * 3 = 276$

5.6 Treino e validação

Há várias técnicas para dividir o conjunto de dados em treino e validação, mas elas não cabem perfeitamente aqui por um motivo: falta de dados. Apesar da quantidade de pacotes de rede no conjunto ser **enorme**[9], o número de janelas de tempo para as quais o modelo fará a predição é, na realidade, bem baixo. Todos os dados do conjunto escolhido representam apenas uma semana onde cerca de 15 incidentes acontecem. Não temos muitos exemplos de incidentes, além do que eles podem ser substancialmente diferentes uns dos outros. A depender do jeito que o conjunto de treino for selecionado, seria possível que não existisse certos incidentes na validação, por exemplo.

Na falta de um bom modo de se dividir em treino e validação devido à falta de dados, e tentando manter o princípio de que todo incidente ocorrido deveria aparecer tanto na validação quanto no treino, acabamos por aleatoriamente escolhendo 30% das linhas da tabela para validação, e o restante para treino.

5.7 Pesos na função de perda

No cenário de classificação binária, o número de janelas que continham incidente foi cerca de $7,5\%$, o que configura um **conjunto de dados desbalanceado**. Para se ganhar uma intuição do problema da situação, uma função de predição que sempre afirma que não está ocorrendo nenhum incidente acertaria $92,5\%$ das vezes. Isso pode prejudicar o desempenho do modelo, especialmente se ele não for capaz de atingir um poder preditivo que seria melhor que isso, de acordo com sua função de perda.

[9] O conjunto de dados está disponibilizado como arquivos compactados `.zip` que juntos totalizam mais de `300Gb` de dados. Uma vez descompactados, o tamanho quase que dobra e os arquivos ainda estão num formato de compactação, o `.pcap`

Existem algumas técnicas para se lidar com conjuntos de dados desbalanceados. As três mais conhecidas são **sobreamostragem**, **subamostragem** e **pesos na função de perda**.

A técnica de **sobreamostragem** consiste em criar artificialmente mais dados que representam a classe menos representativa. O jeito mais fácil de se fazer isso é simplesmente replicar os dados existentes. Existe, porém, meios mais rebuscados de se criar esses dados artificias, geralmente olhando para a topologia dos dados das classes menos representativas e usando suas características para criar dados novos. Essa é a base da técnica conhecida como **SMOTE** [22]. Já a subamostragem se baseia em ignorar a existência de alguns dados da classe mais representativa, de modo que os dados que serão utilizados para treino estarão balanceados. O problema de se utilizar subamostragem é que nós já temos uma quantidade baixa de dados. Se fossemos utilizar **sobreamostragem**, é muito importante que a sobreamostragem seja feita de maneira cuidadosa [23], de modo a não criar dados que causem aumento de variância (*overfit*) do modelo, já que ele estará aprendendo com dados que **não são reais**. A maneira cuidadosa envolve mais etapas de validação, que significaria reservar uma porcentagem maior dos dados para essa parte, o que nos leva de volta ao problema de quantidade de dados.

Por esses motivos escolhemos utilizar **pesos na função de perda**. O *gradient boosting*, como algoritmo de gradiente descendente, cresce a cada etapa de modo a minimizar a função de perda. O método de pesos se baseia em, na hora de calcular a função de perda, utilizar uma média ponderada entre as amostras, pesando mais para a classe menos representativa. Deste modo, errar a predição para 50% das amostras da classe menos representativa se torna "tão ruim quanto"errar a mesma porcentagem na classe mais representativa, mesmo que o número absoluto seja bem diferente. Como será mostrado mais a frente, isso traz ganhos representativos no poder preditivo.

5.8 Avaliação

A parte de avaliação é uma parte extremamente importante da construção do modelo. Ela dá um norte para a confiança de que os resultados estão bons, bem como dita os passos de otimização (parte que será coberta no capítulo seguinte).

No nosso contexto, a pergunta que majoritariamente ditará a avaliação é a seguinte: quantos dos incidentes realmente serão detectados pelo sistema (**sensitividade**) versus quantos alarmes falsos ocorrerão (que trazem ineficiência e falta de confiança no sistema, possivelmente criando uma situação de "Pedro e o Lobo"). Existe uma métrica que balanceia ambas preocupações, conhecida como **Area under Receiver Operating Characteristics** ou simplesmente **AUC**. A interpretação do número proveniente da métrica **AUC** é a probabilidade de que, amostrados aleatoriamente um exemplo onde há incidente e um onde não há, o exemplo positivo terá um *ranking* maior do que o exemplo negativo. Ela é, olhando somente o número, uma métrica de **ordenação**, dando uma noção de quão bem o modelo está separando os dois grupos. A parte mais interessante da métrica, porém, é o jeito que ela é gerada. A Figura 5.1 exemplifica uma curva ROC, que tem relação com a AUC.

Para entender essa figura, precisamos lembrar que a saída de um modelo de *gradient boosting* não nos dá apenas a classificação, mas sim uma **probabilidade de que aquela amostra seja positiva**, e cabe ao usuário decidir qual será a porcentagem exigida para considerar que a amostra seja positiva. A ideia da **ROC** é graficamente mostrar como a **taxa de verdadeiros positivos** (sensitividade) e a **taxa de falsos positivos** variam de acordo com esse limiar. Se você escolher o limiar como sendo 0.0, toda amostra é considerada positiva, e portanto você tem 100% de taxa de verdadeiro positivo, mas também 100% de taxa de falso positivo, cenário inverso porém análogo ao cenário onde se escolhe o limiar 1.0. A curva, portanto, tem a propriedade de deixar claro o *trade-off* na hora de se decidir o limiar.

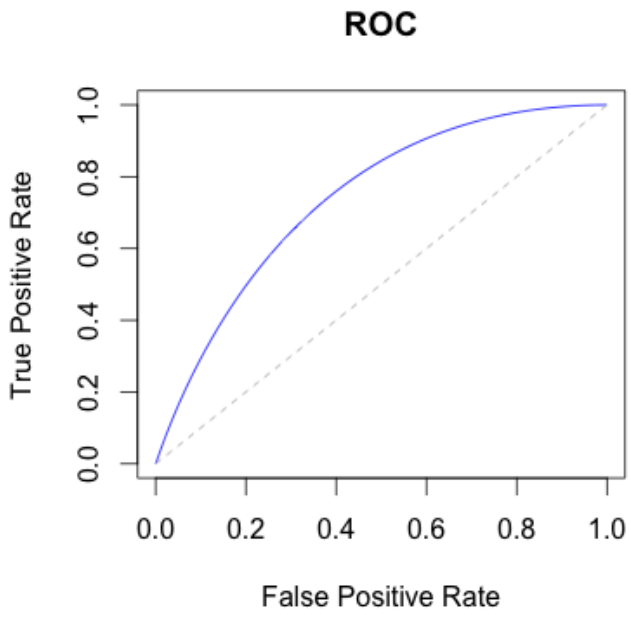

Figura 5.1: Característica de Operação do Receptor, ou *Receiver Operating Characteristic*

Um exemplo de uso é especificar a taxa de verdadeiro positivo esperada e otimizar para a redução da taxa de falso positivo.

Por fim, essa métrica também pode ser usada para avaliar um multiclassificador. Apesar de não ser possível calcular um **AUC** único, dado que nesse cenário o modelo dá como saída uma predição para cada possível classe, é possível gerar essa métrica para cada classe, utilizando uma técnica chamada "um contra todos". Essa técnica consiste em considerar que uma classe é o exemplo positivo e todas as outras são o exemplo negativo para que se possa calcular o **AUC** naturalmente nesse cenário, e depois repetir o processo para todas as outras classes, de modo que cada classe ganha sua avaliação.

Por causa da alta interpretabilidade, balancear as duas preocupações que giram em torno da implantação de um sistema de detecção de incidentes de rede e ser possível em ambos cenários (classificador binário e multiclassificador), essa será a métrica de avaliação para nosso modelo.

Capítulo 6

Aperfeiçoamento do modelo

Após a construção do modelo, há certas otimizações que podem ser feitas para melhorar seu desempenho. Nesse capítulo iremos elaborar como duas dessas otimizações podem ser feitas. Os resultados que essas otimizações tiveram no modelo construído para nosso experimento virão num outro capítulo, mais a frente.

6.1 Necessidade de seleção de *features*

A necessidade de seleção de features é contraintuitiva. Num primeiro momento, faz sentido pensar que quanto mais informações são adicionadas ao modelo, maior a capacidade preditiva dele. Isso, porém, não é verdade. Com uma quantidade fixa de amostras, o poder preditivo de um classificador ou regressor aumenta com o número de dimensões (*features*) até certo ponto, e depois passa a diminuir [24], fato conhecido como fenômeno de Hughes [25].

Dentre os motivos que existem para se realizar a seleção de *features*, além do fenômeno de Hughes, as principais são a redução do tempo de treinamento do modelo, maior interpretabilidade da saída do mesmo (as predições) e redução de variância.

O jeito mais intuitivo de entender o por que o aumento do número de dimensões pode ser danoso, é observar que certas *features* podem ser redundantes (alta correlação com alguma outra *feature*, ou com um conjunto de outras) ou irrelevantes (não há correlação entre a *feature* e o alvo). O processo de seleção de *features* se baseia na identificação desses casos.

6.2 SHAP para seleção de *features*

Para realizar essa seleção de features, vamos utilizar o **SHAP (SHapley Additive exPlanations)** [26]. Essa biblioteca nomeada a partir do método utilizado se propõe a ser uma abordagem unificada de interpretabilidade para **qualquer** modelo de aprendizagem de máquina. Essa afirmação, apesar de forte, tem certo embasamento dado que é uma técncia bem robusta que combina sete diferentes métodos [27][28][29][30][31][32][33] para se chegar na interpretação da saída de um modelo.

Dada a complexidade do SHAP, basta entender que ele é capaz de afirmar, para cada amostra do seu conjunto de dados, quanto cada coluna (*feature*) corroborou para o modelo "empurrar"essa amostra para alguma das possíveis classes do alvo. O valor que quantifica esse fenômeno é chamado de **valor shap** pela própria biblioteca. Tomando a média do valor shap para cada coluna, temos uma medida razoável de importância daquela coluna para a predição do modelo. O passo final é cortar todas as dimensões cuja importância média fica abaixo de algum patamar.

6.3 Otimização de hiperparâmetros

A implementação de *gradient boosting* do **LightGBM**, como todo algoritmo de aprendizagem de máquina, possui parâmetros que devem ser atribuídos de forma pseudo-arbitrária, não por que eles não possuem interpretação nos mecanismos do algoritmo, mas por que seu valor exato não tem motivação racional. Por exemplo, sabemos que quanto mais profundas forem as árvores

agindo como *weak learners*, mais suscetível a aumento de variância (*overfit*) seu modelo está, mas não sabemos se profundidade máxima 7 terá desempenho melhor do que profundidade máxima 8.

O processo de explorar os possíveis valores desses parâmetros de modo a aumentar o desempenho do modelo no conjunto de dados de validação é chamado de **otimização de hiperparâmetros**. A abordagem mais simples, chamada de **Grid Search**, é elencar, para cada possível parâmetro, todos os valores para os quais o desempenho deve ser testado, e para toda combinação possível de atribuição de valor ao conjunto de todos parâmetros, testar o desempenho do modelo. O LGBM, porém, possui uma quantidade enorme de parâmetros [34], a maioria podendo assumir qualquer valor inteiro ou real, então é fácil ver que - ainda mais com um conjunto de dados grande - essa busca exaustiva tende a demorar mais do que é razoável.

Um luxo que nosso experimento não tem é um conjunto de dados de validação de tamanho grande. Como a otimização de hiperparâmetros procura o modelo de melhor desempenho para o conjunto de validação, uma busca muito extensa que apenas marginalmente muda o desempenho pode não ter ganhos significativos no desempenho com dados reais, se esse conjunto for pequeno. Para tentar contornar esse problema, emprega-se a técnica de **validação cruzada**, onde o desempenho do modelo é testado em diferentes separações de validação e treino. Dada essa limitação, não testamos técnicas de otimização mais rebuscadas, e preferimos nos apoiar no **grid search** e no guia de otimização que a própria documentação do LGBM provém [35].

É importante lembrar que o processo de otimização de hiperparâmetros pode ser uma tarefa eterna. Modelos em produção cuja aplicação vê ganhos tangíveis com pequenas otimizações em seu desempenho tendem a estar sempre à procura de um conjunto de atribuições melhor para os seus parâmetros. O mais comum, porém, é que essa busca seja feita com um balanço entre tempo de busca e melhoria de desempenho, logo após o treinamento do

modelo.

Capítulo 7

Monitoramento do modelo

Apesar do monitoramento do modelo construído não fazer parte dos experimentos que fizemos, é uma questão extremamente importante para se atentar quanto à saúde da solução conforme o tempo passa, pós implantação da mesma. Nesse capítulo vamos apresentar os monitoramentos mais comuns e o que eles significam.

7.1 Monitoramento de distribuição

O monitoramento de distribuição nada mais é do que observar a distribuição de probabilidade que a saída do modelo apresenta. A ideia é comparar a distribuição observada durante treinamento com a distribuição que está sendo gerada no dia-a-dia.

No caso do **LGBM** (e outros algoritmos de classificação, como a regressão logística) a saída da função de predição é, na realidade, uma probabilidade. Podemos aproximar, portanto, $P[X] <= x$ onde x é algum valor entre 0.0 e 1.0 e X é a variável aleatória referente à predição do modelo. Aplicando isso no conjunto de treino nós estabelecemos uma distribuição de probabilidade esperada para a saída de nosso modelo.

O segundo passo é: de tempos em tempos (quanto maior o fluxo de dados, mais frequente é possível realizar essa validação) calcular a distribuição que foi gerada a partir dos dados novos e comparar com a distribuição "esperada". A comparação se dá por alguma métrica que dê informação quanto à diferença das duas distribuições, seja o quão próxima estão, ou quanta informação perdemos ao aproximar a original pela nova, etc. Bons exemplos são a divergência de **Kullback-Leibler** ou a estatística de **Kolmogorov-Smirnov**.

No nosso caso, um mal-comportamento do modelo pode preveni-lo de detectar incidentes, logo se a distribuição observada está bem longe da esperada isso é causa para preocupação de quem cuida da saúde do modelo.

Outro cenário a se considerar é a possibilidade de sazonalidade. Muitas vezes a distribuição observada depende da época em que está se observando. Nesse caso é possível aplicar diferenciação e observar a diferença dessa métrica em comparação ao - por exemplo - mês passado.

O monitoramento de distribuição, quando não saudável, indica uma mudança significativa nos dados que estão entrando para receber a predição, ou às vezes um problema no código que está passando essa predição a frente.

7.2 Monitoramento de execução

O monitoramento de execução se preocupa em uma coisa simples: o modelo está sendo executado? Como a arquitetura é complexa, há diversos cenários que fariam o nosso *daemon* parar de avisar se a rede está sendo atacada ou não. Por exemplo, um mal funcionamento na camada de velocidade pode impedir que informações cheguem ao modelo, de modo que ele se torna incapaz de gerar uma predição. Nesse contexto basta observar número de predições (no nosso caso, número de janelas que foram recebidas e devidamente classificadas) por dia ou hora.

7.3 Monitoramento de desempenho

O monitoramento de desempenho responde a seguinte pergunta: "Supondo que não há nenhum problema operacional na arquitetura, as predições que saem do modelo continuam tão boas quanto o esperado (treinamento)?". Para respondê-la, devemos olhar duas coisas: a métrica de avaliação do modelo e a métrica de negócio.

Conforme cada nova janela de tempo é classificada como segura ou não, (ou classificada como o tipo de incidente que está acontecendo), é possível reavaliar o seu modelo (usando as mesmas métricas que foram discutidas no **Capítulo 5**) nos períodos mais recentes e - principalmente - fora do conjunto de dados de treino. É natural que essas métricas piorem com o tempo, mas é necessário controlar o quanto estão piorando.

A parte mais importante, porém, quando se trata de monitoramento de desempenho, é em relação às métricas de negócio. Quanto tempo está sendo gasto para averiguar falsos positivos? Quantos dos incidentes que realmente aconteceram foram avisados cedo o bastante para que uma ação fosse tomada? Essas são duas perguntas que fazem imediato sentido no contexto, mas as perguntas que devem ser feitas vem dos objetivos ao se implantar um sistema desse tipo. O importante é avaliar o modelo periodicamente quanto à sua eficiência ao resolver o problema para o qual foi construído.

7.4 Monitoramento de *feature*

O monitoramento de *feature* se preocupa com duas coisas: o perfil do portfólio de entrada (comportamento esperado dos pacotes de rede, segundo o que foi alimentado no treinamento) e a saúde dos "geradores de dados". A primeira preocupação diz respeito a como o tempo muda o que configuramos como uma situação normal na rede (Exemplo: com o crescimento da empresa o fluxo de pacotes enviados e recebidos aumenta, isso pode gerar

uma falsa detecção de DDoS) e a segunda se preocupa com o código que está capturando esses dados.

Para monitorar ambas as coisas é natural olhar para a média e desvio padrão no caso de *features* numéricas e a proporção no caso de *features* categóricas, além do percentual de dados faltando (`null`) em ambos os casos. Como o número de *features* costuma ser grande (e portanto, a atenção dedicada a cada uma não pode ser muita) e as mudanças com o tempo são mais esperadas do que num monitoramento de distribuição de saída, não é comum ver métricas muito rebuscadas aqui.

O caso de uma *feature* (ou, no geral, um conjunto delas) estar gradualmente mudando suas métricas configura uma mudança de portfólio, ou seja, o comportamento geral observado dos dados de entrada não bate mais com o esperado. É importante olhar esse monitoramento em conjunto com o monitoramento de desempenho para decidir se é necessário retreinar o modelo com dados mais recentes.

No caso de mudanças bruscas nessas métricas, é mais provável que exista um erro em nível de código. Por exemplo, se o serviço que captura dados de rede está falhando, e enviando apenas nulos para a fila de mensagem. Outro possível cenário fictício - onde a métrica teria mudança brusca - é se o utilitário que captura os pacotes de rede sofreu uma atualização, e agora manda o tamanho dos pacotes em bytes ao invés de bits, causando uma redução na média do tamamho dos pacotes. Não há uma solução concreta para problemas identificados dessa forma, pois esses costumam ser mais particulares.

Capítulo 8

Experimentos e conclusões

Nesse capítulo iremos mostrar os resultados obtidos com os experimentos e o que isso significa para o objetivo inicial de se elaborar uma arquitetura capaz de detectar incidentes de rede de maneira eficiente.

8.1 Poder preditivo do modelo

Seguindo os passos que foram descritos nos capítulos **Construção do modelo** e **Aperfeiçoamento do modelo**, iremos descrever como cada etapa se deu na prática e seus resultados.

8.1.1 Primeira iteração: classificador binário

O primeiro passo foi utilizar todas as 276 *features* para construir um modelo de classificação binária utilizando todos os parâmetros no valor padrão e sem atribuir pesos na função de perda.

Já nesse primeiro passo tivemos que fazer duas concessões, quando comparado com o plano inicial. O poder de computação necessário para se processar todo o conjunto de dados foi subestimado, e só conseguimos pro-

cessar 5 dos 7 dias de dados, contendo um total de 9[1] tipos de incidentes diferentes. Além de conseguir computar apenas um subconjunto dos dias, tivemos que mudar nossa janela deslizante para deslizar a cada 5 minutos, ao invés de todo minuto, mudança que pode tanto diminuir quanto aumentar o poder preditivo (e logo, algo que poderia ser testado numa próxima iteração da arquitetura), mas de qualquer forma aumenta a latência inerente da detecção de incidentes. Apesar dessas concessões não serem de maneira alguma algo positivo, consideramos o conjunto de dados processado suficiente para a validação do modelo e, consequentemente, da arquitetura.

O resultado foi surpreendentemente positivo para uma primeira iteração: **Um AUC de 88,6%** conforme ilustrado na Figura 8.1. É importance lembrar que, por definição, um classificador aleatório tem um AUC de 50%, o que coloca o nosso classificador bem acima de não usar modelo algum.

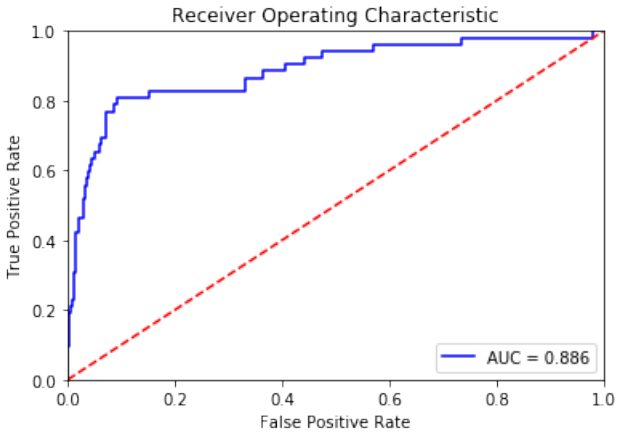

Figura 8.1: Curva da característica de operação do receptor para primeria iteração do modelo

Note como o balanço entre taxa de verdadeiro positivo e taxa de falso positivo fica claro com essa métrica: É possível escolher detectar 80% dos

[1]Os ataques que constam são `ssh_bruteforce`, `dos_hulk`, `ddos_loic_udp`, `ddos_hoic`, `brute_force_web`, `brute_force_xss`, `sql_injection`, `infiltration` and `botnet`

incidentes com esse modelo, porém cerca de 10% das janelas que não contém incidente seriam falsamente acusadas.

8.1.2 Segunda iteração: Pesos na função de perda

Como um primeiro passo de melhoria, resolvemos lidar com nosso conjunto de dados desbalanceado utilizando pesos na função de perda. A adição desses pesos no modelo melhorou seu desempenho em $0,3$ p.p. como pode ser visto na Figura 8.2.

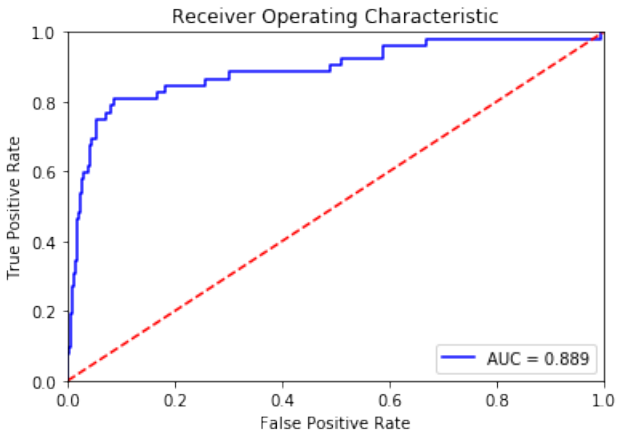

Figura 8.2: Curva da característica de operação do receptor para segunda iteração do modelo

Apesar do ganho de desempenho não ser estatisticamente significativo, a escolha de manter esses pesos se dá não só pelo possível ganho marginal, mas também pela preocupação de que o desbalanceamento será especialmente danoso no cenário de multiclassificação, visto que haverá muitos poucos exemplos de cada tipo de incidente.

8.1.3 Terceira iteração: seleção de features

Para a segunda etapa de melhoria do modelo, utilizamos as métricas do **SHAP** para reduzir o número de *features*. De fato, 276 colunas para apenas

cerca de 2000 janelas de tempo é absolutamente desnecessário. Além disso, ganhar interpretabilidade e deixar de usar o modelo como caixa preta é um ganho secundário desse procedimento. A Figura 8.3 exibe as *features* mais importantes de forma ordenada.

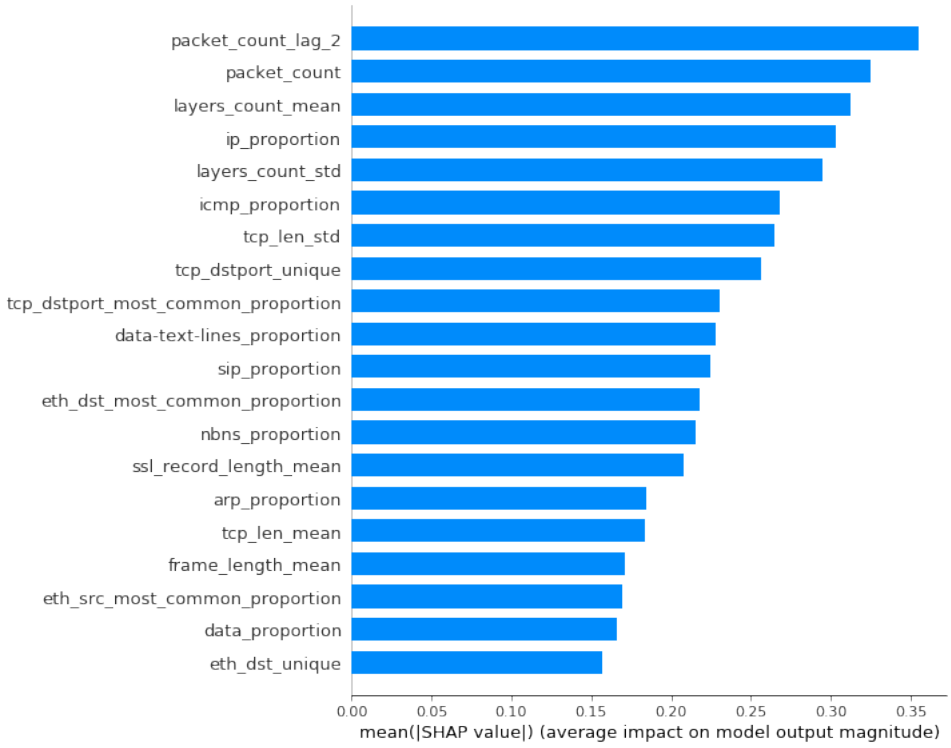

Figura 8.3: Dimensões mais importantes para o modelo de classificação binária

Por meio de tentativa e erro, nós colocamos $0,1$ como limiar mínimo de importância absoluta média no **SHAP** para manter uma *feature*. Isso reduziu o número de features de 276 para **apenas** 33 e também causou um aumento no desempenho do modelo, de $88,9\%$ para $89,9\%$ conforme pode ser observado na Figura 8.4.

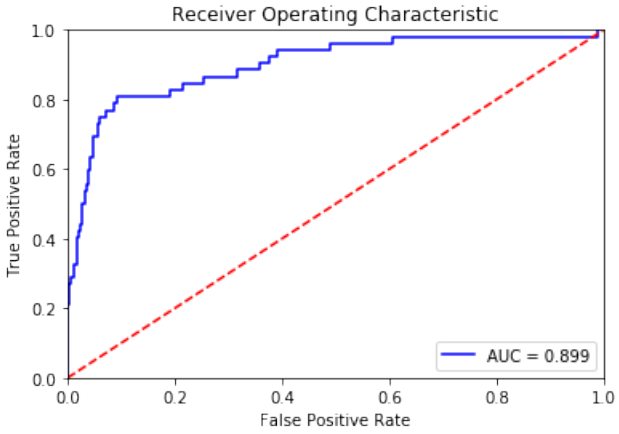

Figura 8.4: Curva da característica de operação do receptor para terceira iteração do modelo

8.1.4 Última iteração: otimização de hiperparâmetros

A última etapa foi, dado o novo conjunto de colunas e os pesos na função de perda, otimizar os hiperparâmetros de acordo com a métrica de avaliação. Para evitar uma busca demasiadamente extensa em **todos** os parâmetros possíveis, nos apoiamos no guia que a própria documentação do LGBM provém [35] e tentamos variar apenas os parâmetros `num_leaves`, `min_data_in_leaf`, `max_bin`, `learning_rate`, `num_iterations` e `min_child_weight`.

Foi interessante ver como a variação dos hiperparâmetros utilizados era capaz de oscilar a performance entre pouco acima de 70% e o melhor resultado encontrado: 91%. Com a otimização, o valor de quatro parâmetros foram mudados (`num_leaves`, `min_data_in_leaf`, `max_bin` e `min_child_weight`) enquanto dois deles ficaram com o valor padrão (`learning_rate` e `num_iterations`). A Figura 8.5 ilustra o resultado.

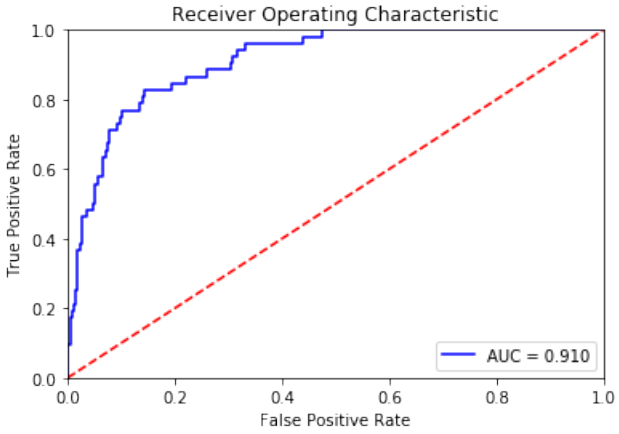

Figura 8.5: Curva da característica de operação do receptor para última iteração do modelo

Esse é o desempenho final do modelo de classificação binária, que acreditamos ser não só suficiente para que uma implantação do sistema de detecção ocorra numa escala maior, mas também suficiente para ser considerada evidência de que o potencial do poder preditivo de um modelo melhor construído (no sentido de usar mais dados, ter explorado mais *features* e ter tido mais iterações) é grande o bastante para que o sistema como envisionado se torne realidade.

8.1.5 Desempenho do multiclassificador

A função de um multiclassificador no contexto de detecção de incidentes de rede é poder dizer não só **se** um incidente está ocorrendo, mas também **qual**. A princípio não é claro se esse multiclassificador seria inerentemente menos performático, ao mesmo tempo que uma perda de desempenho pequena pode ser justificada pela maior interpretabilidade, que resulta num menor tempo de resposta ao incidente.

Com um modelo de classificação binária já construído e otimizado, nos sentimos confortáveis em testar sua utilização para o cenário de multiclassi-

ficação. O ideal seria passar por **todos** os procedimentos de criação do modelo novamente, com o alvo de predição atualizado, para se obter o melhor desempenho possível. Sabemos, porém, que ao separarmos cada incidente acabamos com pouquíssimos exemplos de cada um deles, problema que é exacerbado na validação. Por esse motivo vamos olhar para o **AUC** de cada classe para observar o potencial de um multiclassificador, **sabendo que esse resultado não significa quase nada devido ao número irrisório de amostras na validação**. A Figura 8.6 exibe os resultados.

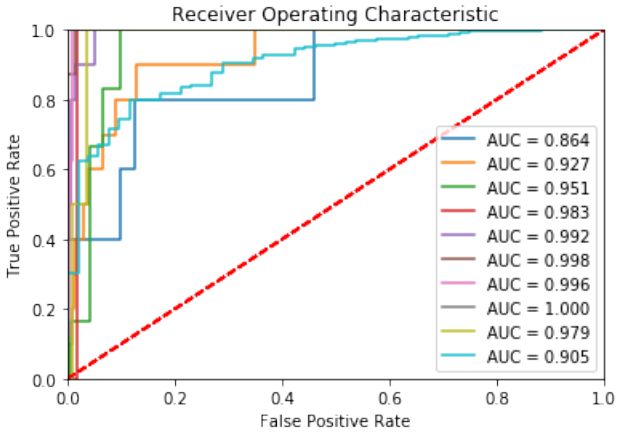

Figura 8.6: Curva da característica de operação de cada classe do multiclassificador

É possível observar que apenas uma classe forma uma curva razoavelmente suave, que é a classe de **não-incidente**. Todas as outras classes tem "curvas"com grandes saltos, consequência da falta de dados. O interessante é que, apesar disso, a métrica para todas as classes chega num nível bom, com apenas uma das classes sendo pior predita do que a média, representada pelo classificador binário. Por mais que essa não seja um validação suficientemente boa para que seja considerada uma boa representação de um cenário real, é prova de que um multiclassificador tem potencial de ter bom desempenho, sendo capaz de dar direcionamento na hora de combater algum incidente.

8.2 Viabilidade da arquitetura

Os experimentos realizados para validar a arquitetura, como descritos no **capítulo 4** foram um sucesso, ou seja, foram capazes de corretamente consumir mensagens do Kafka, agregar pelo Spark e persistir o `.parquet` de modo que o *daemon* conseguisse avaliar cada janela utilizando o modelo serializado. No cenário dos experimentos, a arquitetura cumpre o propósito.

É importante ressaltar, porém, quais preocupações não foram validadas pelos experimentos, devido a seu caráter de pequena escala. O cluster de Spark e de Kafka consistiam ambos de apenas uma máquina, aquela que rodou os experimentos, e não configurando - portanto - um cenário de computação distribuída com todas suas fraquezas, mesmo que estejamos utilizando ferramentas tolerantes à falha. Além disso, o volume de dados testado não é o volume esperado numa implantação real do sistema, ou seja não foi possível validar a escalabilidade do sistema, mesmo que essa dependa majoritariamente da quantidade e capacidade das máquinas utilizadas. Por fim, foi testado apenas o envio de mensagens ao Kafka cujo conteúdo vem de pacotes de rede que foram utilizados no treino do modelo, então não é possível ter certeza de que um pacote capturado num cenário real não fosse causar erros em algum dos passos, por exemplo contendo valores não esperados na agregação ou um tipo de incidente que não faz parte do conjunto de treino. Note que a última afirmação implica que o comportamento da arquitetura e do modelo é desconhecido para incidentes não antes ocorridos, validação não feita devido à quantidade pequena de dados e escopo reduzido dos experimentos.

Acreditamos ainda assim que os experimentos têm seu mérito como prova de conceito, garantindo que a arquitetura como desenhada de fato realiza a tarefa almejada, mesmo que não exista certeza quanto aos problemas que podem surgir numa escala maior.

8.3 Como ferramenta de detecção de incidentes

Antes de avaliar a ferramenta segundo os experimentos como ferramenta de detecção de incidentes, vamos recapitular quais eram os requisitos do projeto bem como suas ambições. Como mencionado no capítulo **Introdução**, a solução ideal deve:

1. Armazenar apenas os pacotes relevantes em memória volátil;
2. Armazenar em memória não-volátil apenas os dados que possibilitassem a realização de investigações posteriores e aprendizado automático;
3. Informar que há um possível incidente num intervalo de tempo curto;
4. Ter implantação e manutenção fácil, de modo que o analista de redes não precise entender profundamente da solução;

De um ponto de vista de utilização de memória, é verdade que a arquitetura só precisa gastar memória com um pacote em específico enquanto ele estiver na fila do Kafka, uma vez que tudo que interessa é a predição que o modelo serializado dá de saída, e portanto após ser utilizado para agregação da janela de tempo que pertence, um pacote pode ser jogado fora. Também é verdade que seria possível armazenar todos os pacotes que fazem parte de uma janela de tempo alertada como suspeita para posterior investigação, mesmo que isso não tenha sido feito durante os experimentos. Essas duas afirmações parecem contemplar os itens 1. e 2., e portanto todas preocupações quanto a memória, desde que **o modelo serializado seja disponibilizado por quem provê a ferramenta**.

A arquitetura como idealizada pode informar que um incidente está acontecendo praticamente o quão cedo quanto desejado, que significa apenas que esse gargalo não está na arquitetura e/ou ferramentas utilizadas, mas sim no poder preditivo do modelo. É possível escolher uma janela deslizante de

duração e deslizamento quão pequenos ou grandes quanto desejáveis, porém isso possivelmente afetará o desempenho do modelo. Os resultados do experimento indicam que um tempo de detecção de 5 a 10 minutos é possível em cenário não-fictício, que já é por si só uma conquista, porém possivelmente não suficiente, uma vez que existem ataques que demoram menos que isso para acontecerem, e falhar em detectá-los rapidamente pode ter efeitos catastróficos. Em resumo, seria necessário mais tempo na construção do modelo para intervalos de tempo menores e refazer os experimentos nesse cenário, mas a não-dependência da arquitetura no tamanho dessa janela deixa o caminho aberto para que o requerimento seja cumprido.

Como a arquitetura proposta é um sistema de detecção de incidentes para uma rede inteira, a instalação do sistema para cada máquina que faz parte da rede já pode configurar e utilizar essas próprias máquinas como parte do *cluster* de Spark **e** Kafka, tornando o trabalho preparatório para a implantação do sistema prticamente não existente. Com o sistema implantado, e com ajuda do **SHAP**, ele é capaz de dar direcionamento para a investigação, uma vez que algum pacote é classificado como possível incidente, pois os valores shap dizem quais dimensões estão sendo responsáveis pela predição final.

Existem, porém, dois pontos importantes que podem tornar o uso desse sistema a longo prazo mais difícil do que o desejado. Em primeiro lugar, é uma arquitetura que configura, mesmo que por baixo dos panos, um cenário bem complexo de processamento de dados, com direito a todas fraquezas de se utilizar computação distribuída tanto como fonte de dados quanto como camada de velocidade. Qualquer *bug* de perda de pacotes no formato de mensagem do Kafka, por exemplo, vai ser extremamente difícil de ser depurado por um analista de redes sem conhecimento profundo. Em segundo lugar, a criação, manutenção e monitoramento do modelo devem ser feitas por especialistas de aprendizagem de máquina, e não é imediatamente claro como o possível provedor da arquitetura seria capaz de obter dados

e criar um modelo de poder preditivo geral o bastante para funcionar em qualquer cenário, apontando para a possível necessidade de que a própria pessoa implantando o sistema deve criar seu modelo e, consequentemente, guardar todos os pacotes de rede capturados para que ocorra o treinamento do mesmo.

Juntando todos os pontos levantados, apesar de não ser possível concluir agora se a arquitetura elaborada cumpre todos os requerimentos iniciais em *qualquer cenário*, especialmente pela validação necessária do desempenho do modelo num conjunto de dados real e de outra rede, é claro que a proposta atual é um passo na direção correta, já acompanhada de ideias para próximas iterações que foram descritas tanto ao longo da monografia, quanto no próximo capítulo.

Capítulo 9

Trabalho futuro

Como fim dessa monografia, iremos deixar possíveis próximos passos para ampliar o conhecimento sobre o problema ou destrinchar detalhes de implantação do mesmo.

9.1 Ampliar a arquitetura

A arquitetura descrita aqui é inspirada na arquitetura Lambda e se propõe a ser uma solução robusta, escalável e tolerante a falhas para o problema de detecção de incidentes de rede. Ela, porém, não configura um ecossistema completo de análise de dados de rede.

As partes onde isso é mais notável são na camada de lote e na camada de serviço. Quanto à primeira, nós não temos um banco como "conjunto de dados mestre"para guardar os dados - pois pegamos o conjunto de dados pronto - e também não usamos uma boa ferramenta para o processamento em lote desses dados (como o Apache Spark, que estamos utilizando só na camada de velocidade). Quanto à última, ela é praticamente não-existente, dado que não existe a necessidade no escopo atual, como explicado no **Capítulo 2**, mas seria interessante utilizar uma ferramenta de indexação de visualizações e agregações anteriormente construídas como o **Apache Druid** [18].

Parte dessas adições se justificam por si, quando se pensa em implantar um sistema desses em um cenário real. Os dados precisam ser guardados em algum lugar, e calcular o conjunto de dados de entrada do modelo pode ser uma tarefa demasiadamente pesada se o seu conjunto de treino crescer. Seria interessante, porém, explorar quais outras visualizações que poderiam ser calculadas pela camada de lote e que ajudariam no papel de detecção.

9.2 Dados que não sejam pacotes de rede

Há uma grande oportunidade de melhorar o poder preditivo do modelo que não coube no escopo de nosso experimento: utilizar dados que não sejam pacotes de rede (junto a eles) no modelo. É possível, por exemplo, explorar *logs* de todas as máquinas envolvidas. Utilização de memória, CPU, disco, alarmes de firewall, etc.

A ideia por trás disso é dar contexto para a informação que os pacotes de rede estão providenciando. Se o fluxo de pacotes chegando aumentar, mas o uso da CPU não, talvez não seja uma situação preocupante.

É particularmente atrativo tomar esse ponto como o próximo passo pois o conjunto de dados que utilizamos para o experimento já disponibiliza dados do tipo, apesar de que a qualidade ou utilidade dos dados presentes não foram validados.

9.3 Um conjunto de dados real

O conjunto de dados que encontramos - o **CSE-CIC-IDS2018** - cumpria todos os requisitos para ser suficiente no escopo de nosso experimento. Nele havia dados de incidentes modernos, envolvia várias máquinas (algumas locais, outras na nuvem) e tinha os dados de pacotes de rede no formato `.pcap`, ou seja, os pacotes puros, antes de qualquer processamento.

Não é claro, porém, se ele seria suficiente para ser utilizado num cenário real. Algoritmos de aprendizagem de máquina sofrem de um viés forte aos dados que enxergaram. Seria necessário validar se o conhecimento do que é um incidente nessa rede analisada configura um cenário parecido com o mesmo incidente na rede em que seria implantada o sistema.

Outro problema é que temos apenas sete dias de dados coletados onde 23 incidentes ocorreram, representando 13 tipos de invasões diferentes, o que significa que o seu modelo pode não ter um poder de generalização muito forte, dado que você não está olhando para um período de tempo suficientemente longo e nem para várias instâncias do mesmo tipo de invasão.

Seria interessante validar se um bom modelo construído em cima desse conjunto de dados seria igualmente (ou suficientemente) efetivo em uma outra rede.

Referências Bibliográficas

[1] Activision Blizzard, Inc. Form 10-K (https://investor.activision.com/node/32301/html)

[2] Tweeted by Blizzard CS - The Americas, Blizzard's official twitter customer support channel (https://twitter.com/BlizzardCS/status/1170431100155584513)

[3] Capital One Breach Shows a Bank Hacker Needs Just One Gap to Wreak Havoc (https://www.nytimes.com/2019/07/30/business/bank-hacks-capital-one.html)

[4] Detecting Encrypted TOR Traffic with Boosting and Topological Data Analysis, HJ van Veen (https://kepler-mapper.scikit-tda.org/TOR-XGB-TDA.html)

[5] Machine Learning in Cyber-Security - Problems, Challenges and Data Sets, Idan Amit, John Matherly, William Hewlett, Zhi Xu, Yinnon Meshi, Yigal Weinberger, 19 Dec 2018 (https://arxiv.org/abs/1812.07858v3)

[6] CSE-CIC-IDS2018 on AWS, a collaborative project between the Communications Security Establishment (CSE) and the Canadian Institute for Cybersecurity (CIC) (www.unb.ca/cic/datasets/ids-2018.html)

[7] A repository dedicated to the Lambda Architecture (LA) (lambda-architecture.net)

[8] A transactional database with a flexible data model, elastic scaling, and rich queries. (www.datomic.com)

[9] Big Data: Principles and Best Practices of Scalable Real-time Data Systems. 2015. Nathan Marz, James Warren

[10] Rao, Supreeth; Gupta, Sunil. "Interactive Analytics in Human Time". 17 June 2014

[11] Bae, Jae Hyeon; Yuan, Danny; Tonse, Sudhir. "Announcing Suro: Backbone of Netflix's Data Pipeline", Netflix, 9 December 2013

[12] Artigo da Wikipedia sobre Apache Kafka (https://en.wikipedia.org/wiki/Lambda$_a rchitecture$)

[13] Artigo da wikipedia sobre Apache Spark (https://pt.wikipedia.org/wiki/Apache$_S park$)

[14] Spark Streaming Programming Guide (https://spark.apache.org/docs/2.2.0/streaming-programming-guide.html)

[15] The State of the Octoverse: machine learning, 24 Janeiro 2019, Thomas Elliott (https://github.blog/2019-01-24-the-state-of-the-octoverse-machine-learning/)

[16] scikit-learn - Machine Learning in Python (https://scikit-learn.org/stable/)

[17] Introducing fklearn: Nubank's machine learning library (Part I), Lucas Estevam, 23 Abril 2019, (https://medium.com/building-nubank/introducing-fklearn-nubanks-machine-learning-library-part-i-2a1c781035d0)

[18] Apache Druid, a high performance real-time analytics database (https://druid.apache.org/)

[19] Yoav Freund, Robert E. Schapire (1999). A Short Introduction to Boosting

[20] Friedman, Jerome. (2000). Greedy Function Approximation: A Gradient Boosting Machine. The Annals of Statistics. 29. 10.1214/aos/1013203451.

[21] Guolin Ke1, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen1, Weidong Ma1, Qiwei Ye1, Tie-Yan Liu. LightGBM: A Highly Efficient Gradient Boosting Decision Tree

[22] N. V. Chawla, K. W. Bowyer, L. O. Hall, W. P. Kegelmeyer - Jun 1, 2002 - SMOTE: Synthetic Minority Over-sampling Technique

[23] Marco Altini - 17 de Agosto de 2015 - DEALING WITH IMBALANCED DATA: UNDERSAMPLING, OVERSAMPLING AND PROPER CROSS-VALIDATION (https://www.marcoaltini.com/blog/dealing-with-imbalanced-data-undersampling-oversampling-and-proper-cross-validation)

[24] Trunk, G. V. (July 1979). "A Problem of Dimensionality: A Simple Example". IEEE Transactions on Pattern Analysis and Machine Intelligence. PAMI-1 (3): 306–307. (doi:10.1109/TPAMI.1979.4766926)

[25] Hughes, G.F. (January 1968). "On the mean accuracy of statistical pattern recognizers". IEEE Transactions on Information Theory. 14 (1): 55–63. (doi:10.1109/TIT.1968.1054102)

[26] Github repository slundberg/shap - A unified approach to explain the output of any machine learning model. (A unified approach to explain the output of any machine learning model.)

[27] LIME: Ribeiro, Marco Tulio, Sameer Singh, and Carlos Guestrin. "Why should i trust you?: Explaining the predictions of any classifier."Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, 2016

[28] Shapley sampling values: Strumbelj, Erik, and Igor Kononenko. "Explaining prediction models and individual predictions with feature contributions."Knowledge and information systems 41.3 (2014): 647-665.

[29] DeepLIFT: Shrikumar, Avanti, Peyton Greenside, and Anshul Kundaje. "Learning important features through propagating activation differences."arXiv preprint arXiv:1704.02685 (2017).

[30] QII: Datta, Anupam, Shayak Sen, and Yair Zick. "Algorithmic transparency via quantitative input influence: Theory and experiments with learning systems."Security and Privacy (SP), 2016 IEEE Symposium on. IEEE, 2016.

[31] Layer-wise relevance propagation: Bach, Sebastian, et al. "On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation."PloS one 10.7 (2015): e0130140.

[32] Shapley regression values: Lipovetsky, Stan, and Michael Conklin. "Analysis of regression in game theory approach."Applied Stochastic Models in Business and Industry 17.4 (2001): 319-330.

[33] Tree interpreter: Saabas, Ando. Interpreting random forests. (http://blog.datadive.net/interpreting-random-forests/)

[34] This page contains descriptions of all parameters in LightGBM - LGBM docs - (https://lightgbm.readthedocs.io/en/latest/Parameters.html)

[35] This page contains parameters tuning guides for different scenarios - LGBM docs - (https://lightgbm.readthedocs.io/en/latest/Parameters-Tuning.html)