PONTIFÍCIA UNIVERSIDADE CATÓLICA DO PARANÁ

ESCOLA POLITÉCNICA

PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA (PPGIa)

ARQUITETURA DE SERVIÇOS ORIENTADOS A APRENDIZAGEM DE MÁQUINA *ONLINE*

Miguel Gustavo Rodrigues

Curitiba

2024

Miguel Gustavo Rodrigues

ARQUITETURA DE SERVIÇOS ORIENTADOS A APRENDIZAGEM DE MÁQUINA *ONLINE*

Projeto de Dissertação apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica do Paraná, como requisito parcial à obtenção do título de mestre em Informática.

Orientador: **Dr. Altair Olivo Santin**

Coorientadores: **Dr. Fabrício Enembreck**

Dr. Eduardo Kugler Viegas

Curitiba

2024

Dados da Catalogação na Publicação
Pontifícia Universidade Católica do Paraná
Sistema Integrado de Bibliotecas – SIBI/PUCPR
Biblioteca Central
Edilene de Oliveira dos Santos CRB 9 / 1636

R696a
2024

Rodrigues, Miguel Gustavo
Arquitetura de serviços orientados a aprendizagem de máquina online / Miguel Gustavo Rodrigues ; orientador: Altair Olivo Santin ; coorientadores: Fabrício Enembreck, Eduardo Kugler Viegas. -- 2024
112 f. : il. ; 30 cm

Dissertação (mestrado) – Pontifícia Universidade Católica do Paraná, Curitiba, 2024
Bibliografia: f.106-112

1. Informática. 2. Arquitetura de software. 3. Aprendizagem do computador. 4. Software – Desenvolvimento. I. Santin, Altair Olivo. II. Enembreck, Fabrício. III. Viegas, Eduardo Kugler. IV. Pontifícia Universidade Católica do Paraná. Programa de Pós-Graduação em Informática. V. Título

CDD 20. ed. – 004

Pontifícia Universidade Católica do Paraná
Escola Politécnica
Programa de Pós-Graduação em Informática

Curitiba, 23 de janeiro de 2025.

13-2025

DECLARAÇÃO

Declaro para os devidos fins, que **MIGUEL GUSTAVO RODRIGUES** defendeu a dissertação de Mestrado intitulada "**Arquitetura de serviços orientados a Aprendizagem de Máquina Online"**, na área de concentração Ciência da Computação no dia 18 de novembro de 2024, no qual foi aprovado.

Declaro ainda, que foram feitas todas as alterações solicitadas pela Banca Examinadora, cumprindo todas as normas de formatação definidas pelo Programa.

Por ser verdade firmo a presente declaração.

gov.br Documento assinado digitalmente
EMERSON CABRERA PARAISO
Data: 23/01/2025 17:10:10-0300
Verifique em https://validar.iti.gov.br

Prof. Dr. Emerson Cabrera Paraiso
Coordenador do Programa de Pós-Graduação em Informática

Agradecimentos

Gostaria de expressar minha mais profunda gratidão à minha família, que esteve ao meu lado em todos os momentos desta jornada. Em especial, agradeço à minha mãe, por seu carinho inabalável e apoio incondicional, que foram pilares fundamentais durante todo o meu percurso. À minha irmã, por seu constante companheirismo e incentivo, que sempre me motivaram a seguir em frente. Ao meu falecido pai, cuja memória permanece viva em mim, inspirando-me diariamente a buscar o melhor de mim em cada desafio.

Agradeço também aos meus professores, Prof. Dr. Altair Santin e Prof. Dr. Fabrício Enembreck, pela confiança depositada e pela oportunidade de crescimento acadêmico que me proporcionaram. Ao Prof. Dr. Eduardo Viegas, expresso meu profundo agradecimento pela orientação atenta e pelo apoio inestimável durante o desenvolvimento desta dissertação, que foram cruciais para sua conclusão.

A todos, o meu sincero e eterno muito obrigado.

Para minha amada mãe.

Lista de figuras

Lista de tabelas

Lista de Abreviaturas

API	*Application Programming Interface*
ARF	*Adaptive Random Forest*
DSRM	*Design Science Research Methodology*
FGCS	*Future Generation Computer Systems*
HT	*Hoeffding Tree*
HTTP	*Hypertext Transfer Protocol*
ID	*Identity*
IA	*Inteligência Artificial*
LGPD	*Lei Geral de Proteção de Dados*
ML	*Machine Learning*
MLOps	*Machine Learning Operations*
NB	*Naive Bayes*
SL	*Stream Learning*
SMB	*Server Message Block*
VM	*Virtual Machine*

Lista de Símbolos

Y	*Conjunto de rótulos possíveis*
X	*Vetor de características de uma instância*
y	*Classe de uma instância*
C	*Rótulo de uma classe*

Sumário

Resumo

As arquiteturas tradicionais para a implementação de operações de *machine learning* (MLOps) frequentemente enfrentam dificuldades para atender às demandas de ambientes de *stream learning* (SL). Os modelos implementados precisam ser atualizados incrementalmente em tempo real e devem ser escaláveis para lidar com fluxos de dados em constante evolução. Este trabalho propõe uma nova arquitetura distribuída adaptada para a implantação e atualização de modelos SL no contexto do MLOps, implementada em duas etapas. Primeiramente, estruturamos os componentes principais como microsserviços implantados em um ambiente de orquestração de contêineres, garantindo baixo *overhead* computacional e alta escalabilidade. Em segundo lugar, propomos uma estratégia de versionamento periódico de modelos para atualizar continuamente os modelos SL sem comprometer a precisão do sistema. Investigamos condições específicas dos modelos antes de acionar a tarefa de versionamento, explorando a natureza dos algoritmos SL. Isso permite que nossa arquitetura seja estendida para lidar com atualizações incrementais dos modelos SL, assegurando alto desempenho e precisão em ambientes de produção. Demonstramos a viabilidade da proposta por meio de um protótipo implementado como uma arquitetura distribuída de microsserviços sobre o *Kubernetes*. Nossa arquitetura mantém a precisão de uma configuração tradicional de um único ponto de inferência. O *throughput* de inferência pode ser escalado e um modelo SL atualizado pode ser entregue em menos de 2,5 segundos.

Palavras-chave: MLOps, Stream Learning, Kubernetes, Microsserviços.

Abstract

Traditional architectures for implementing machine learning operations (MLOps) often struggle to meet the demands of stream learning (SL) environments. The deployed models must be incrementally updated in real-time and scalable to handle constantly evolving data streams. This paper proposes a new distributed architecture tailored for the deployment and updating of SL models within the MLOps framework, implemented in two stages. First, we structure the core components as microservices deployed in a container orchestration environment to ensure low computational overhead and high scalability. Second, we propose a periodic model versioning strategy to continuously update SL models without compromising system accuracy. We investigate specific model conditions before triggering the versioning task by leveraging the characteristics of SL algorithms. This allows our architecture to be extended to handle incremental updates of SL models, ensuring high performance and accuracy in production environments. We demonstrate the proposal's viability through a prototype implemented as a distributed microservice architecture on top of Kubernetes. Our architecture maintains the same accuracy as a traditional single-endpoint configuration. The inference throughput can be scaled, and an updated SL model can be delivered in less than 2.5 seconds.

Keywords: MLOps, Stream Learning, Kubernetes, Microservices.

Capítulo 1

Introdução

Machine Learning (ML), ou aprendizagem de máquina em português, é um campo da Inteligência Artificial (IA) no qual sistemas de computação podem aprender com os dados sem serem explicitamente programados [1]. Com o avanço das abordagens e arquiteturas que viabilizam a criação e implantação dos modelos de ML, gradualmente, mais empresas puderam usufruir de técnicas de ML para compreender melhor seus clientes, aprimorar seus produtos, facilitar suas operações e, assim, tornar-se mais competitivas em suas respectivas áreas de atuação.

Inúmeras aplicações têm sido desenvolvidas nas mais diversas áreas, como automação de processos, atendimento automático por meio de *chatbots*, otimização de cadeias de suprimentos, criação de experiências personalizadas para os usuários por meio de *marketing* direcionado para cada perfil, aprimoramento nos sistemas de cibersegurança, carros autônomos, entre outros [2].

De modo geral, a aprendizagem de máquina consiste em criar modelos capazes de aprender padrões e relações em um conjunto de dados de entrada e, em seguida, espera-se que este modelo seja capaz de generalizar esses padrões para fazer predições em novas entradas de dados desconhecidos [3]. Dentre os tipos de aprendizagem de máquina, a aprendizagem supervisionada é a mais amplamente utilizada. Neste tipo de aprendizagem os modelos são treinados utilizando um conjunto de dados rotulados, ou seja, cada um dos exemplos de entrada (vetor de *features*) tem sua saída real informada (*label*).

A aprendizagem supervisionada trata basicamente dois tipos de problemas: classificação, onde o objetivo é atribuir a um conjunto de dados de entrada uma categoria ou classe específica com base em suas características, ou regressão, onde, em vez de categorizar os dados em classes, tenta-se estimar ou predizer um valor contínuo [4]. Outra classificação das técnicas de ML se refere à forma de treinamento dos modelos. Enquanto abordagens em lote (*batch*) constroem modelos a partir de uma base de dados pré-

determinada, uma abordagem incremental atualiza continuamente o modelo de ML a partir de um fluxo de dados.

Aprendizado incremental, aprendizado *on-line*, aprendizado em tempo real e aprendizado em fluxo de dados (*Stream Learning* em inglês) são termos comumente associados a algoritmos de ML que atualizam seus modelos com um fluxo contínuo de dados sem executar várias passagens sobre esses dados [5]. O fluxo de dados pode ser gerado a partir de diferentes fontes e possuem as seguintes características: (i) os fluxos de dados têm tamanho ilimitado, (ii) os dados chegam em alta taxa e velocidade variável, (iii) os dados podem evoluir com o tempo, (iv) o processamento de dados tem restrições de memória [6]. Nesta pesquisa, o foco principal trata de modelos de aprendizado em fluxo de dados conforme detalhamentos a seguir.

A eficácia de um modelo de ML não se limita apenas ao seu treinamento. A criação e implantação de um modelo de ML envolvem diversas etapas distintas e sequenciais, que incluem a coleta de dados, o pré-processamento dos dados coletados, o treinamento do modelo, a validação do modelo gerado, a implantação do modelo em um ambiente de produção, o monitoramento de performance e, por fim, as atualizações do modelo para manter sua capacidade de predição.

Essas etapas, conhecidas como o "Ciclo de Vida" do modelo de ML, formam um processo interativo e contínuo que abrange várias áreas do conhecimento [7]. Diante dessas complexidades, torna-se essencial adotar abordagens que visam mitigar os desafios na gestão de todos os processos e equipes envolvidos nas etapas mencionadas. O MLOps, ou DevOps para *Machine Learning*, adapta os princípios do DevOps (que integra desenvolvimento e operações para ciclos ágeis e confiáveis), ao ciclo de vida de modelos de ML [8].

Essa abordagem facilita a automação e orquestração dos fluxos de processos, o gerenciamento de versões dos modelos, a padronização e reutilização de códigos e a escalabilidade do sistema. Dessa forma, o MLOps proporciona maior segurança e conformidade nos processos de criação e implantação dos modelos em ambientes de produção [9].

Um dos pontos mais críticos na utilização de ML é sua implantação em ambientes de produção. Este processo envolve disponibilizar os modelos gerados nas etapas de treinamento e validação para que possam ser utilizados para fazer previsões em novos dados gerados pelos usuários ou sistemas [10], esta fase, também é conhecida como inferência.

De modo geral, as consultas aos modelos de ML são feitas remotamente, onde o grande fluxo de dados é referente à quantidade de consultas individuais dos inúmeros usuários distintos [11]. Como por exemplo, em aplicações de detecção de fraude de cartão de crédito [12], ou em sistemas de recomendações de anúncios [13]. Nestes casos, uma arquitetura deve ser construída visando disponibilizar os modelos para atender às solicitações dos usuários.

Existem diversas ferramentas que viabilizam o desenvolvimento e gerenciamento de uma arquitetura de MLOps para disponibilizar os modelos de ML em produção. Pode-se encontrar desde ferramentas específicas como *MLflow* [14], com foco no rastreamento de experimentos e versionamento de modelos, até ferramentas mais abrangentes como os serviços da AWS *SageMaker* [15], ou *Google Cloud AI Platform* [16], que compreendem todos os componentes de uma estrutura para ML.

Além disso, outras tecnologias podem ser usadas para a construção de arquiteturas para implantação de ML como por exemplo, microsserviços [17] e conteinerização de aplicações [18], que frequentemente são utilizados de forma conjunta, permitindo a criação de sistemas capazes de adaptar as necessidades do ambiente.

Em essência, uma estrutura para servir modelos de ML é composta por quatro componentes, sendo eles: (i) módulo de inferência, onde os modelos são carregados e são responsáveis por receberem as requisições, realizar a predição utilizando o modelo em memória e retornar a predição ao requisitante [19]; (ii) o módulo de atualização do modelo, responsável por retreinar o modelo utilizando as instância que já foram rotuladas. Normalmente, o retreinamento é feito em intervalos de tempo pré-estabelecidos, utilizando lotes de instância rotuladas previamente armazenadas [20]; (iii) o módulo de versionamento, onde os modelos gerados no processo de treinamento e retreinamento são armazenados e ficam à disposição dos módulos de inferência [21]; e (iv) o módulo de monitoramento, responsável por avaliar o desempenho do modelo em produção além de compará-lo com os novos modelos gerados na atualização, possibilitando assim, uma possível substituição do modelo em produção quando este é superado pelos modelos mais atuais [22].

1.1 Problema de pesquisa

De modo geral, as arquiteturas atuais de MLOps, como as apresentadas na seção 3.1 deste trabalho, atendem com eficiência aos requisitos dos cenários de ML em lote tradicional, onde os padrões dos dados mudam lentamente ao longo do tempo. No entanto, em ambientes de *Stream Learning*, as práticas e métodos utilizados no MLOps tradicional podem não satisfazer completamente as necessidades das aplicações. Isso ocorre porque o MLOps tradicional não está preparado para lidar com atualizações incrementais nem com a disponibilização de modelos atualizados em períodos muito curtos.

Em uma arquitetura para ML *Stream*, a predição ou inferência, deve suportar a grande quantidade de solicitações e por isso, normalmente, utiliza técnicas de paralelismo para poder processar a enorme quantidade de dados [23]. Embora essa característica seja a mesma de um sistema de MLOps tradicional, em uma arquitetura para ML *Stream*, os *endpoints* (aplicações que encapsulam os modelos) necessitam da constante substituição do modelo salvo na memória pelos modelos mais atualizados, o que gera um desafio quando o número de *endpoints* é grande ou quando a taxa de substituição do modelo é elevada.

Por isso, o versionamento dos modelos também recebe uma atenção diferenciada, uma vez que o carregamento para os *endpoints* e a serialização das novas versões dos modelos, devem ocorrer o mais rápido possível [24]. Além disso, o módulo de versionamento deve ser capaz de gerenciar de forma eficiente a grande quantidade de versões de modelos gerados com as atualizações constantes.

Dentre as etapas do processo de ML *Stream*, certamente a mais desafiadora é a atualização do modelo. Ao contrário do MLOps tradicional, em que os modelos são atualizados em lotes ou minilotes e com pouca frequência (em alguns cenários as substituições dos modelos podem ocorrer anualmente) [25], em ambientes de ML *Stream* essas atualizações devem ser feitas constantemente. Em um cenário ideal, a atualização deve ser feita de forma incremental.

No entanto, atualizações incrementais são complexas de serem implementadas em um ambiente de produção por diversos motivos [26]. O principal deles, é disponibilizar o modelo para os *endpoints* de inferência cada vez que este atualiza seus parâmetros, ou seja, assim que recebe um novo evento. Isso pode causar interrupções do serviço de inferência,

uma vez que os *endpoints* gastam muito tempo e recurso computacional na substituição dos novos modelos [27].

Devido à alta complexidade dos processos, há uma escassez de trabalhos relacionados ao desenvolvimento de estruturas para ML *Stream* que tratam da atualização dos modelos em tempo real ou quase real, em um contexto de MLOps. Nos trabalhos encontrados, os autores tratam parcialmente ou simplesmente ignoram o problema do processo de atualização constante dos modelos de ML, o que influencia diretamente a performance de predição do sistema ao longo do tempo [28].

Em muitos casos, os autores adotam o procedimento padrão de atualização do MLOps tradicional, fazendo atualizações em lotes periodicamente [29]. O problema desse método é que o modelo pode sofrer degradação de performance antes que a próxima atualização esteja disponível.

Já outros autores adotam estratégias de detecção de *concept drift* [30], e nestes casos, o problema é que o disparo para atualização do modelo, ocorre apenas depois que é identificada a mudança nos padrões dos dados usados na inferência, ou quando a performance do modelo decai. Isso pode causar sérios problemas na capacidade preditiva do sistema, uma vez que, quando o modelo atualizado fica disponível para a produção, pode não estar mais condizente com a realidade dos dados atuais, devido à demora para a geração de um novo modelo, utilizando os métodos do MLOps tradicional.

Assim, a escassez de trabalhos relacionados a ML *Stream* na literatura, juntamente com a indisponibilidade de ferramentas e procedimentos que auxiliem na implantação dos modelos, inviabilizam a expansão do uso de ML *Full Stream* (onde a inferência e a atualização do modelo são feitas utilizando fluxos de dados e de forma contínua) em ambientes de produção.

1.2 Objetivos

O objetivo geral deste trabalho é desenvolver uma arquitetura escalável e parametrizável que implemente as práticas de MLOps para predição, atualização e versionamento do modelo por meio do *Stream Learning* em ambientes de produção.

Para tanto, será necessário atingir os seguintes objetivos específicos:

- Definir uma arquitetura que viabilize a implementação distribuída de um modelo de predição para *Stream Learning*.
- Desenvolver uma arquitetura de MLOps que suporte atualização incremental baseada em *Stream Learning*.
- Definir um mecanismo de disponibilização de modelos atualizados quase em tempo real.
- Prototipar, desenvolver, testar e avaliar a arquitetura proposta.

1.3 Método de pesquisa

Esta seção apresenta a metodologia de pesquisa adotada para conduzir este trabalho. A escolha do método científico para este trabalho foi motivada pela necessidade de uma abordagem estruturada e eficiente que combine a criação de soluções práticas com a contribuição para o conhecimento teórico. Sendo assim, optou-se pela *Design Science Research Methodology* (DSRM) [31] para guiar o desenvolvimento e avaliação dos artefatos propostos, que é particularmente adequado para a criação e avaliação de soluções práticas em tecnologia da informação e engenharia.

A DSRM segue uma sequência de seis etapas, cada uma desempenhando um papel crucial no desenvolvimento de soluções eficazes e validadas cientificamente. As etapas são:

1. Identificação do Problema e Motivação.

Identifica-se claramente o problema a ser resolvido e articula-se a motivação para sua solução. O problema abordado nesta pesquisa é a dificuldade de atualizar modelos de ML *Stream* de forma incremental e disponibilizar esse modelo atualizado em uma frequência próxima a ideal em um ambiente de produção.

2. Definição dos Objetivos da Solução.

Estabelecem-se objetivos claros e mensuráveis para a solução. Neste trabalho, busca-se criar um sistema que permita a atualização incremental de modelos de ML *Stream*, garantindo a disponibilidade do modelo atualizado para atender às solicitações dos usuários

de forma eficiente. Além disso, avaliam-se os impactos das taxas de disponibilização dos modelos e do paralelismo na performance preditiva e no *throughput* do sistema.

3. Projeto e Desenvolvimento.

Desenvolve-se o artefato proposto, que neste caso, trata-se de um sistema modular para atualização de modelos de ML *Stream*, dividido em dois módulos principais: um para atender as solicitações de predição dos usuários e outro para atualizar o modelo com instâncias rotuladas e disponibilizá-los em um repositório de modelos. Ambos os módulos devem operar de maneira eficiente e integrada.

4. Demonstração.

Realizam-se demonstrações para validar a eficácia do artefato. Nesta pesquisa, utilizam-se simulações e estudos de caso para mostrar como o sistema proposto pode atualizar modelos de ML *Stream* de forma incremental, mantendo a capacidade de atender às solicitações dos usuários.

5. Avaliação.

Empregam-se testes de desempenho para avaliar a eficácia do sistema. Nessa fase, os testes são conduzidos em cenários simulados de um ambiente de ML *Stream*, garantindo que todos os módulos sejam testados exaustivamente. Essa abordagem busca fornecer evidências de que o sistema resolve o problema identificado e aprimora a eficiência operacional.

6. Comunicação.

Os resultados da pesquisa e o conhecimento adquirido são comunicados. Nesta pesquisa os resultados serão apresentados no respectivo capítulo desta dissertação, além de serem submetidos para publicação científica e apresentados em conferências.

A aplicação do DSRM nesta pesquisa proporciona uma abordagem estruturada para abordar o problema de pesquisa em questão. A metodologia guia o desenvolvimento do sistema proposto e assegura que cada etapa do processo seja fundamentada em princípios científicos sólidos e rigorosamente validados, contribuindo para a construção de um conhecimento útil e aplicável em contextos reais.

1.4 Contribuições

Esse trabalho apresenta como contribuição científica uma nova arquitetura que viabiliza a implementação de S*tream Learning* considerando as práticas de MLOps. A arquitetura proposta adiciona mecanismos que viabilizam a realização da predição de forma escalável, versionamento dos modelos e atualização incremental de modelos de *Stream Learning*. De forma mais resumida, o trabalho apresenta as seguintes contribuições:

- Uma arquitetura de *Stream Learning* no contexto de MLOps com atualização incremental e versionamento de modelos.
- Definição de uma frequência adequada de serialização, que permita balancear o uso dos recursos computacionais e a qualidade dos modelos em produção.
- Definição da melhor relação entre o paralelismo e performance preditiva do sistema.
- Abordagem para substituição de modelos em ambiente de produção sem a exigência de recriar a aplicação por completo.

Dadas as contribuições, este trabalho também resultou na submissão de um artigo ao periódico *Future Generation Computer Systems* (FGCS), que está atualmente em análise.

1.5 Organização do Documento

O restante deste documento está elaborado da seguinte forma: No Capítulo 2, é feita a fundamentação necessária para o desenvolvimento do projeto. No Capítulo 3 é apresentado uma revisão da literatura mostrando os trabalhos já realizados na área de implantação de *Stream Learning*. No Capítulo 4, é apresentada a proposta da pesquisa e o protótipo desenvolvido. Já no Capítulo 5 são apresentados os experimentos realizados juntamente com uma análise dos resultados obtidos. Por fim, no Capítulo 6 é apresentado a conclusão do projeto proposto, além de uma descrição das atividades futuras.

Capítulo 2

Fundamentação

Este capítulo busca estabelecer as bases conceituais e contextuais para compreender os princípios e teorias que embasam a abordagem do problema de pesquisa proposto. Para tanto, serão discorridos sobre o conceito de *Machine Learning*, *Stream Learning*, MLOps e Ferramentas e *Frameworks* para *Stream Learning*.

2.1 *Machine Learning* (ML)

Machine Learning (ML) é um subcampo da inteligência artificial (IA) que se concentra no desenvolvimento de algoritmos e modelos computacionais que permitem que os sistemas aprendam a partir de dados e melhorem seu desempenho em tarefas específicas sem serem explicitamente programados. O processo de aprendizado em ML envolve a identificação de padrões e a generalização a partir dos dados, o que permite que os sistemas façam previsões, tomem decisões ou automatizam tarefas com base em experiências passadas [32].

A mineração de dados usando técnicas de ML tem revolucionado a forma como empresas e organizações extraem *insights* valiosos a partir de grandes conjuntos de dados, possibilitando a tomada de decisões mais informadas e eficazes em diversas áreas, como a análise de mercado, medicina, previsão de demanda e muito mais [33]. Essa técnica consiste em processar grandes quantidades de dados em busca de padrões consistentes que auxiliem na descoberta de conhecimento. Aprendizagem de máquina, inteligência artificial e técnicas de estatísticas são ferramentas utilizadas em mineração de dados para a construção de modelos capazes de prever resultados ou agrupar instâncias conforme semelhanças de características, possibilitando assim, compreender de forma mais precisa

comportamento de clientes, prever movimentações de mercado, detectar tendências, modelar públicos-alvo etc. [34].

O processo de ML é concebido através de etapas denominadas ciclo de vida do ML, como mostra a Figura 1.

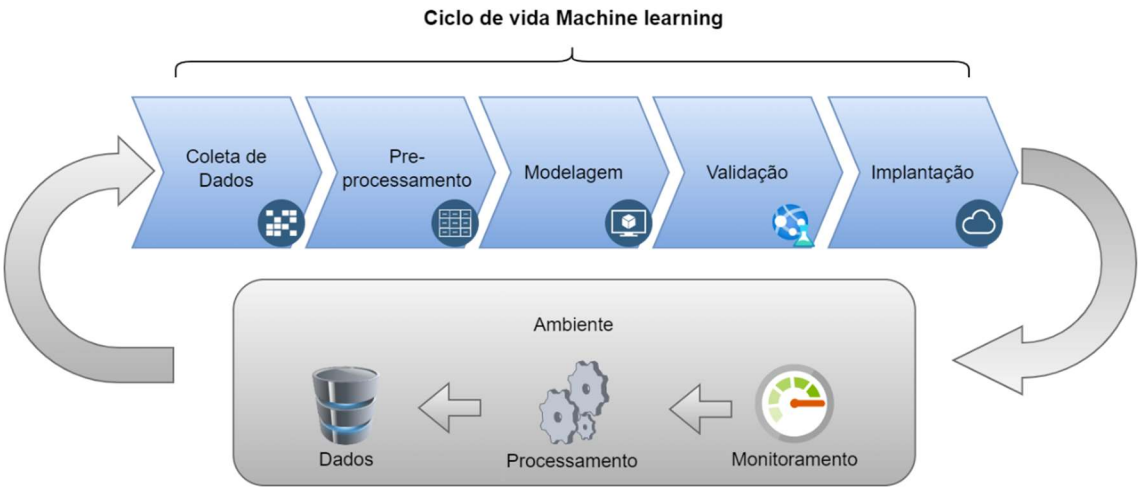

Figura 1 - Ciclo de Vida do Machine Learning. Fonte: autor.

O número de etapas envolvidas no processo pode variar de acordo com a literatura, mas basicamente podem ser definidas em 6 etapas [35]:

1. Definição de objetivo.

Nessa etapa, define-se os objetivos que se pretende alcançar com a implantação do ML. Para isso, é necessário fazer o levantamento de informações relacionados ao caso ou ao ambiente onde se pretende aplicar o ML, com o intuito de criar as metas, métricas de avaliação, questões relevantes e problemas que devem ser sanados [35].

2. Coleta de dados.

Os dados são o substrato essencial para a implantação do processo de ML, sendo a quantidade e qualidade desses dados elementos cruciais para a qualidade da solução implantada. Nessa fase, é feita a coleta dos dados que serão utilizados na criação dos modelos, esses dados podem ser coletados de diversas fontes geradoras, como: arquivos de logs, tráfego de redes, banco de dados, mídias sociais, relatórios, sensores etc. [35].

3. Pré-processamento.

O conjunto de dados deve estar padronizado e livre de ruídos para geração de modelos eficientes. Existem uma infinidade de problemas relacionados a geração de dados, como: falha de leitura de sensores, preenchimento errado de formulários, valores faltantes,

dados desbalanceados entre outros. Sendo assim, nessa etapa são aplicadas técnicas para normalização e limpeza nos conjuntos de dados antes de utilizá-los na criação dos modelos [35].

4. Modelagem.

Nessa etapa, são aplicadas as técnicas de modelagem para identificação de padrões e correlações nos dados selecionados, gerando assim os modelos matemáticos que serão usados nas predições ou agrupamentos de novas instâncias de dados [35].

5. Validação dos resultados.

O ML é um processo cíclico, onde os modelos são avaliados segundo as métricas estabelecidas na etapa 1. Caso os objetivos especificados não sejam alcançados, as etapas anteriores devem ser refeitas, quantas vezes forem necessárias para a obtenção do objetivo proposto [35].

6. Implantação.

Depois de validar o modelo gerado nas etapas anteriores, é necessário colocá-lo em produção, permitindo que os usuários do sistema o utilizem para realizar inferências. Além disso, é crucial monitorar o desempenho do modelo para garantir sua capacidade preditiva ao longo do tempo. Se houver uma degradação no desempenho, pode ser necessário retreinar o modelo com dados mais recentes.

Na maioria das aplicações, o ML utiliza dados estáticos históricos armazenados em um grande conjunto de dados para alimentar os algoritmos que irão gerar os modelos matemáticos, esse método é conhecido como aprendizagem em lote ou aprendizagem *offline*. Embora seja o método mais utilizado, trabalhar com essa grande quantidade de dados exige muito recurso computacional e tempo de processamento, além de gerar um modelo estático, que não é capaz de se adaptar às mudanças do ambiente [30]. Porém, em alguns cenários, o modelo precisa trabalhar com fluxos de dados contínuos (*Stream Learning*). Nestes casos, é necessário um sistema com respostas em tempo real ou próximo disso. Outro aspecto importante, é que os sistemas de ML para *Stream Learning* devem ser capazes de se adaptarem às mudanças de padrões ocorridas durante a geração dos dados.

2.2 *Stream Learning*

Ao contrário do processamento em lote onde os modelos são estáticos, uma vez que se utiliza dados históricos imutáveis na criação dos modelos, em um cenário de *Stream Learning* os dados mudam de padrão conforme as alterações ocorridas no ambiente [30]. Em um ambiente de *streaming*, os dados gerados são disponibilizados em um fluxo contínuo e ordenado e são transmitidos em uma velocidade muito alta. Nestes casos, os exemplos chegam continuamente na forma de um fluxo de dados S [36].

Entre as tarefas possíveis, como agrupamento, regressão, mineração de grafos, detecção de *outliers* e sistemas de recomendação, a classificação é a tarefa mais proeminente em fluxos de dados. Em problemas de classificação, cada instância $\boldsymbol{X}_\tau \in X$ é associada a um rótulo de classe $y_t \in Y$, onde Y contém l rótulos possíveis, $Y = \{C_1, C_2, \dots C_l\}$. Assim, é possível considerar o fluxo de dados em problemas de classificação como um conjunto de pares $DS = \{(\boldsymbol{X}_1, y_1), (\boldsymbol{X}_2, y_2), \dots (\boldsymbol{X}_t, y_t), \dots\}$ [37].

Nessas situações, os dados podem ser acessados apenas uma vez ou retidos por um curto período. Com isso, cada etapa do *stream learning* (também conhecido como aprendizagem *online* ou incremental) deve ser rápida e consumir pouco recurso computacional [36]. Com a atualização sendo feita a cada nova instância de treinamento, os modelos *online*, naturalmente se adaptam às mudanças de conceito nos dados, também conhecido desvio de conceito ou *concept drift* em inglês (embora para detecção de desvios de conceitos mais abruptos, um detector de desvio pode ser necessário para uma adaptação mais rápida). Com isso, os modelos de ML *Stream*, diferentemente dos modelos de ML tradicionais (*offline*), tendem a se adaptar melhor às alterações inerentes aos fluxos de dados [38].

Gerenciar o ciclo de vida de um modelo de ML *Stream* é mais complexo que gerenciar o ciclo de vida de um modelo de ML tradicional, isso ocorre porque, sistemas *online* precisam lidar com a ingestão de dados continuamente, além de garantir a atualização e inferência do modelo em tempo real [39]. Outro aspecto que contribui para essa complexidade é a quantidade de algoritmos disponíveis para aprendizagem *online*, que é muito menor em relação à aprendizagem em lote. Sendo assim, mesmo obtendo uma maior eficiência em diversas aplicações, a implantação da aprendizagem incremental em ambientes de produção, ainda é um grande desafio para os profissionais de ML [40].

Todos esses desafios inerentes ao *Stream Learning*, fazem com que o aprendizado *online* tenha uma maior aderência em ambientes acadêmicos, onde o cenário pode ser controlado e o desempenho em relação a atraso na inferência são menos relevantes, do que em ambientes comerciais. Sendo assim, atualmente, em cenários de *streaming*, empresas costumam adotar o conceito de aprendizagem em tempo quase real visando mitigar parcialmente os obstáculos existentes na aprendizagem incremental [41].

2.3 MLOps

MLOps, que é uma abreviação de "*Machine Learning Operations*," é uma abordagem que visa facilitar o desenvolvimento, implantação, monitoramento e gerenciamento contínuo de sistemas de aprendizado de máquina em ambientes de produção. Também podemos descrever o MLOps, como uma extensão do DevOps (Desenvolvimento e Operações), adaptada especificamente para atender às necessidades do ciclo de vida de modelos de *Machine Learning*, como mostra a Figura 2. O objetivo do MLOps é garantir que os modelos de *Machine Learning* sejam implantados com sucesso em produção, mantendo seu desempenho e que sejam atualizados de maneira eficiente à medida que novos dados chegam [9].

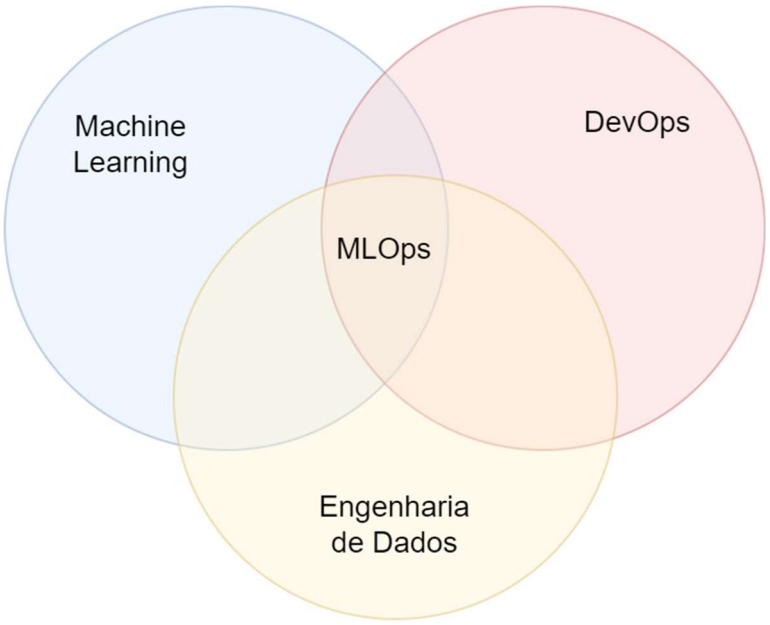

Figura 2 - Representação gráfica do conceito de MLOps. Fonte: adaptado de D. Kreuzberger et al. [9].

Em outras palavras, o MLOps combina práticas e ferramentas especialmente desenvolvidas para atender as peculiaridades do ciclo de vida dos modelos de ML [3].

Uma das etapas mais críticas no MLOps é a implantação de modelos em ambientes de produção. Esta fase envolve a disponibilização dos modelos para serem utilizados para inferência, pelos usuários do sistema. Essa disponibilização pode ocorrer de diversos modos, como serviços web (aplicações conteinerizadas), aplicações locais (através de servidores *on-premise*), ou até mesmo dispositivos IoT (computação de borda). No entanto, além da implantação, os modelos exigem monitoramento contínuo para garantir um desempenho adequado e a identificação precoce de quaisquer desvios de conceito nos dados, ou no comportamento do modelo [10]. Por isso, o MLOps enfatiza a necessidade de revisões e retreinamento de modelos, baseados em informações de monitoramento dos modelos em produção.

Uma das contribuições mais significativas do MLOps é a garantia de que os sistemas de ML permaneçam robustos e confiáveis. O monitoramento contínuo permite a detecção proativa de problemas, enquanto a retroalimentação rápida e a colaboração eficiente entre as equipes de desenvolvimento e operações, garantem que os modelos se ajustem às demandas em constante evolução. Além disso, a automação de tarefas repetitivas reduz a probabilidade de erros humanos e permite uma implantação mais rápida e eficiente [9]. O controle de versão de modelos e dados proporciona rastreabilidade e a capacidade de reverter para versões anteriores quando necessário. Isso não apenas aumenta a eficiência operacional, mas também melhora a segurança e conformidade com regulamentações [42].

Diversas ferramentas foram desenvolvidas visando atender os diversos propósitos de cada etapa do MLOps, algumas ferramentas têm funções mais específicas, já outras funcionam de forma mais abrangente e atende praticamente todo o ciclo de vida de ML. *AirFlow* [43], *Kubeflow* [44], *Seldon* [45], são exemplos de ferramentas que visam proporcionar a orquestração e automatização dos processos envolvidos no ciclo de vida do ML. Por outro lado, algumas ferramentas podem ser usadas em etapas específicas do MLOps, como por exemplo o *MLflow* [46], que permite o rastreamento de experimentos e o versionamento e gerência dos modelos. Existem também, plataformas como AWS *SageMaker* [15], *Azure Machine Learning* [47] *e Google Cloud AI Plataform* [48], que possibilitam um gerenciamento mais amplo, oferecendo diversas ferramentas para atender todo o ciclo de vida do ML tradicional (*offline*).

O MLOps é uma abordagem essencial para escalar e manter sistemas de aprendizado de máquina em ambientes de produção, garantindo que eles forneçam valor contínuo e atendam às necessidades das organizações de forma eficiente. A implementação bem-sucedida do MLOps pode reduzir erros, economizar tempo e recursos, e melhorar a governança de modelos de ML.

2.4 *Frameworks* para *Stream Learning*

Para uso em ambientes acadêmicos ou de simulações, a aprendizagem online conta com algumas ferramentas a sua disposição, que visam auxiliar a construção de modelos de ML *Stream*. O *Massive Online Analysis* (MOA) [49] foi desenvolvido para implementação de algoritmos e execução de experimentos para aprendizado *online* a partir de fluxos de dados em evolução [10]. O MOA é um *framework* de código aberto com foco em ambientes acadêmicos e simulação, com capacidade de processar fluxos de dados massivos utilizando uma quantidade de memória limitada. Por ser escrito em *Java* [50], o MOA tem uma boa portabilidade, podendo ser executado em qualquer ambiente com uma máquina virtual *Java* apropriada. Além de diversos algoritmos de classificação, regressão, agrupamento e outros, o MOA também possui detectores de desvio de conceito e *outliers* além de geradores de *Streams* e métodos de avaliação e estatísticas.

Embora tenha foco em ambientes acadêmicos, o MOA pode ser adaptado para implantar modelos na prática dependendo dos requisitos do problema [51]. O MOA possui uma interface gráfica, mas também pode ser usada via linha de comando. Todos esses fatores contribuíram para tornar o MOA um dos principais *Frameworks* para simulação e testes de ML *Stream*, porém com pouca aderência em aplicações reais, onde a escalabilidade é um fator essencial.

Para aplicações onde o tráfego de dados é muito alto, é fundamental uma ferramenta com arquitetura distribuída, ou seja, capaz de paralelizar o processamento dos dados. O SAMOA [52] é uma plataforma para mineração de grandes fluxos de dados escrita em *Java* baseada no MOA. Ele oferece suporte às tarefas de aprendizado de máquina mais comuns, como classificação, agrupamento e regressão [53].

Ele também fornece uma API (*Application Programming Interface*) para os desenvolvedores que simplifica a implementação de algoritmos de *streaming* distribuídos [54]. O SAMOA permite que várias ferramentas de processamento distribuído como *Apache Storm* [55] e *Apache Samza* [56], se conectem facilmente à estrutura. A ferramenta possibilita a aprendizagem incremental, ou seja, cada nova instância de dados pode ser usada para atualizar o modelo. Todos esses fatores tornam o SAMOA uma ferramenta muito versátil para aprendizagem em grandes fluxos de dados. Porém, assim como o MOA, seu uso continua mais restrito a ambientes acadêmicos ou de simulação.

Embora diversos *frameworks*, sejam escritos em *Scala* [57] ou *Java* [50] devido sua maior velocidade de execução, atualmente, a adoção da linguagem *Python* [58] pelos cientistas e engenheiros de dados cresce vertiginosamente. Por isso, *frameworks* baseados em *Python*, como *Scikit-multiflow* [59], *Creme* [60] e *River* [61], vêm sendo muito utilizados pela comunidade científica e em aplicações comerciais.

O *Scikit-multiflow,* é a versão *Stream* do já conhecido *Scikit-learn* [62], utilizado em aprendizagem em lote. O *Scikit-multiflow* é um framework baseado em *Python* para implementar algoritmos e realizar experimentos na área de *Machine Learning Stream*, que se baseia em estruturas já conhecidas de *software* livre, como *Scikit-learn*, MOA e WEKA. Ele apresenta diversos algoritmos de classificação, regressão e agrupamento além de interoperar com os pacotes numéricos e científicos *Python*, como o *NumPy* [63] e *SciPy* [64]. O *Scikit-multiflow* também contém geradores de fluxo, métodos de aprendizagem, detectores de mudança e métodos de avaliação, além de poder ser usado com o *Jupyter Notebook* [65].

Assim como o *Scikit-multiflow*, o *Creme*, é um *framework* baseado em *Python* que também pode ser usado em qualquer interface de programação. Sua grande vantagem é a simplicidade de implantação, onde, com poucas linhas de código é possível criar um pipeline para criação de um modelo de ML *Stream* [60]. Porém, não possui uma grande variedade de algoritmos e tem um baixo desempenho, o que o torna mais indicado para ambientes acadêmicos e de pesquisa.

Visando absorver o melhor dos dois conceitos, foi criado o *River*, que é o resultado da fusão do *Creme* e *Scikit-multiflow*. O *River* conta com diversos algoritmos de classificação, regressão e agrupamento além de, detectores de desvio de conceito e anomalias, métodos de balanceamentos, métodos de transformação de dados e pré-processamento e diversas métricas de avaliação. Assim como o *Creme* sua implantação é bastante simples [61]. O *River* pode ser utilizado em ambientes acadêmicos e industriais.

Embora essas bibliotecas possibilitem a atualização incremental, sua aplicabilidade em ambientes de produção fica condicionado à utilização de outras ferramentas, uma vez que estas não dão suporte a todo o ciclo do ML *Stream*.

2.5 Considerações Finais

Neste capítulo foi introduzido os conceitos necessários para a ambientação no que diz respeito à aprendizagem *online* e implantação de modelos em produção.

Inicialmente, elucidou-se o conceito de *Machine Learning* e suas principais etapas de desenvolvimento. Em seguida foi discorrido sobre o *Stream learning* e seus principais desafios. Depois, foi explicado o conceito de MLOps e sua aplicabilidade. Por fim, foram apresentados os principais *frameworks* para processamento de *Stream Learning*.

No capítulo a seguir, serão apresentados os trabalhos relacionados a implantação de modelos de ML em produção, onde será realizado uma descrição detalhada de cada um, além da apresentação de uma tabela comparativa, elencando as principais características dos trabalhos encontrados.

Capítulo 3

Trabalhos relacionados

Nesta seção serão apresentados os trabalhos encontrados na literatura que apresentam ferramentas que buscam auxiliar na implantação de *Stream Learning* ambiente de produção. Em seguida será apresentada uma análise dos trabalhos encontrados e apresentada uma tabela comparativa.

3.1 Trabalhos encontrados na literatura que apresentam estruturas de *Stream Learning*

Alguns trabalhos encontrados na literatura, apresentam bibliotecas de ML *Stream* que visam auxiliar na construção e disponibilização de modelos preditivos tendo como entrada fluxos de dados contínuos. Normalmente, essas bibliotecas são utilizadas em conjunto com outras ferramentas ou *Frameworks* para construção de arquiteturas que possibilitem, criar e servir modelos de ML *Stream*, como é o caso do *StreamDM*, apresentada por Alberto Bifet et al. [66], que foi projetado sobre o *Spark Streaming* [67]. O *StreamDM* é uma biblioteca de código aberto desenvolvida em *Scala* no *Ark Lab da Huawei Noah* [68], que pode ser usada tanto em ambientes acadêmicos quanto em reais. Por ser projetado com base no *Spark Streaming* o *StreamDM* se beneficia de estar dentro do ecossistema de código aberto *Hadoop* [69], além de processar os dados de forma distribuída. Porém, o *StreamDM* trabalha apenas com minilotes, por isso, sua latência pode chegar na casa de segundos, reduzindo sua aplicabilidade em ambientes reais.

Outra biblioteca projetada sobre um *Framework* específico é o SOLMA, apresentado por W. Jamil et al. [70], que se beneficia do ecossistema do *Apache Flink* [71], usando os recursos de processamento de dados distribuídos para escalabilidade e tolerância a falhas.

Utilizando linguagens *Java* e *Scala*, o SOLMA conta com algoritmos de classificação, regressão e agrupamento, além do processamento paralelo. O SOLMA permite inferência em tempo real e possibilita o paralelismo na atualização dos modelos, porém os autores não deixam claro se o SOLMA permite atualizações incrementais dos modelos. Embora tenha uma proposta interessante, o SOLMA tem uma baixa aderência pela comunidade e conta com uma documentação deficiente, além de alta complexidade de configuração e baixa disponibilidade de algoritmos, dificultando assim sua aplicação.

Também construído sob um *Framework* específico, temos o trabalho apresentado por Donna Xu et al. [72], que descreve uma arquitetura geral para fornecer serviços de análise em tempo real, onde, os modelos são treinados inicialmente em lote, mas podem ser atualizados continuamente através do fluxo de dados utilizando o conceito de minilotes. O trabalho apresenta uma arquitetura criada utilizando microsserviços, o que permite escalabilidade e flexibilidade do sistema. O serviço *RESTful* é utilizado para inferência do modelo e na comunicação entre os módulos que compõem a arquitetura. A arquitetura também possui um design que permite ao usuário não especialista em ML o fácil acesso à tecnologia de ML apenas enviando solicitações HTTP aos serviços disponíveis na arquitetura. O artigo apresenta um exemplo onde são utilizados o *Apache Spark* [73] e a biblioteca *MLlib* [74] para análise dos dados e criação de modelos de ML respectivamente. Porém, segundo os autores, a arquitetura é capaz de generalizar para outras estruturas de *Big Data* e bibliotecas. Embora não possibilite a atualização dos modelos de forma incremental, sua estrutura permite o paralelismo garantindo assim um bom desempenho em grandes fluxos de dados.

Também são encontrados trabalhos que apresentam plataformas de gerenciamento de ML que fornecem recursos para implantação de modelos para inferência em tempo real, porém com atualização do modelo sendo realizada apenas com aprendizagem em lote, como é o caso da ferramenta *Looper*, apresentada por Igor L. Markov et al. [29]. *Looper* é uma plataforma para gerenciamento do ciclo de vida de modelos ML projetada para atender todas as etapas do ciclo de vida de ML para tomada de decisão em produtos de software, visando sua simplificação. O *Looper* possibilita a automação dos processos permitindo que as tarefas repetitivas sejam automatizadas, economizando tempo e recursos, visando assim, melhorar a experiência do usuário. Embora o *Looper* ofereça suporte à inferência em tempo real, a atualização do modelo é feita em lotes.

Nesta mesma linha, a ferramenta *MLPacker* desenvolvida por Raúl Miñón et al. [75], fornece mecanismos para empacotar e implantar pipelines analíticos tanto em APIs REST

quanto em modo *streaming*, o que facilita o gerenciamento e a reutilização de *pipelines* complexos. Os *pipelines* analíticos podem ser implantados automaticamente (executados em um único nó) ou de maneira distribuída (executados em vários nós de um *cluster*). A ferramenta é desacoplada da etapa de treinamento, evitando assim que blocos de códigos de experimentos sejam integrados na operacionalização. Além do modo de pacote (API REST ou *streaming*), a ferramenta pode ser configurada para realizar os *deploys* utilizando contêineres, como *Docker*, para encapsular *pipelines* e suas dependências [18], tornando-os portáteis e independentes do ambiente de execução. A ferramenta também suporta várias linguagens de programação, possibilitando que os usuários criem seus *pipelines* usando sua linguagem preferida, como *Python* ou *Scala*. Embora a ferramenta ofereça o suporte para inferência do modelo em modo *Stream*, não existe ainda um suporte para atualizar o modelo em tempo real.

Com uma menor frequência, encontramos ferramentas para implantação de ML *Stream* dedicadas a casos específicos, como por exemplo o *Lambda Learner*, desenvolvido por Rohan Ramanath et al. [76]. O *Lambda Learner* é utilizado para prever a taxa de cliques em anúncios publicitários em uma famosa rede social de negócios. Ele utiliza uma junção de processamento em lote (*offline*), que proporciona um modelo robusto, com a aprendizagem em minilotes, possibilitando o *update* do modelo com dados atualizados, tornando-o adaptável às variações de conceito dos dados. O *Lambda Learner* se destaca pela sua eficiência computacional. Ela utiliza uma abordagem *lambda*, que consiste em dividir o processo de aprendizado em duas partes: um componente rápido que atualiza o modelo com novos dados e um componente lento que reavalia periodicamente o modelo completo para garantir a qualidade. No entanto, a ferramenta não disponibiliza uma plataforma para uma utilização mais genérica do sistema, sendo necessário adaptação via código para utilização da ferramenta em outros cenários e embora o termo aprendizagem incremental seja utilizado no artigo, o treinamento *online* é feito utilizando minilotes de dados.

Com um conceito parecido, encontramos o SCARFF (*Scalable Real-time Fraud Finder*), apresentado por Fabrizio Carcillo et al. [77]. Esta é uma plataforma de código aberto para processamento e análise de dados de transações de cartões de crédito, que retorna alertas confiáveis quase em tempo real. O SCARFF integra ferramentas de *Big Data* (*Apache Kafka* [78], *Apache Spark* [73] e *Apache Cassandra* [79]) com uma abordagem de aprendizado de máquina, que lida com desequilíbrio e não estacionariedade. Além disso, o SCARFF foi projetado para manter baixa latência, permitindo a detecção

rápida de fraudes sem atrasos significativos. Ele usa o algoritmo *Random Forest* [5] para classificação de risco de fraude nas transações de cartões e a atualização dos modelos é feita utilizando o conceito de janelas deslizantes (minilotes). Embora a utilização do *Apache Spark*, torne o SCARFF escalável horizontalmente, permitindo assim, tratar volumes crescentes de transações, sua aplicabilidade é exclusiva e sua estrutura não é customizável, tornando assim sua aplicação bastante restritiva.

Encontraram-se também ferramentas que utiliza a abordagem *low-code* (possibilita criar aplicações com pouca ou nenhuma programação), como a plataforma *ClowdFlows* apresentada por Janez Kranjc et al. [80], que é executada como uma aplicação Web e oferece suporte a fluxos de trabalhos de mineração de dados através de uma interface gráfica, abstraindo a necessidade de codificação por parte do usuário. A estrutura é dividida em uma interface gráfica (*frontend*) escrita em HTML e *JavaScript* [81] e um servidor (*Backend*) construído em *Python* com o *framework Django* [82]. *ClowdFlows* apresenta implementações de algoritmo de *Weka* [83], *Orange* [84] e *Scikit-learn* e tem como componente principal para criação dos fluxos de trabalhos uma unidade de processamento chamada de *widget*. Cada *widget* executa uma tarefa considerando suas entradas e parâmetros e armazena os resultados da tarefa em suas saídas. Embora tenha sido concebido inicialmente para trabalhar apenas com processamento em lote, foram implementados *daemons* especiais que executam fluxos de trabalho em um intervalo de tempo fixo, possibilitando assim sua utilização em ambientes de *stream.* Porém, na plataforma *ClowsFlows* apenas as inferências podem ser feitas utilizando um fluxo de dados, sendo o treinamento realizado utilizando o conceito de lote.

Foram encontrados outros trabalhos na literatura que são baseados em *Frameworks* bastante conhecidos, como a arquitetura *Big Data Engine*, apresentada por Mikołaj Komisarek et al.[28]. O *Big Data Engine* é uma ferramenta para processamento de *Stream* e seu *backend* possui diversas ferramentas, como o *Apache Spark* para processamento dos dados, o *Apache Kafka* para ingestão de dados, o *Elasticsearch* [85] como banco de dados *NoSQL*, o *Kibana* [86] para visualização de resultados e o HDFS [69] para armazenamento distribuído de dados. O *Big Data Engine* é dedicado à detecção de atividades e padrões de rede que podem indicar atividades maliciosas ou suspeitas, como ataques cibernéticos. Além do processamento em lote, a ferramenta também é capaz de processar dados de rede em tempo real permitindo a detecção imediata de comportamento anômalo. Devido às ferramentas para processamento distribuído em seu *backend*, o *Big Data Engine* possibilita o escalonamento vertical para atender o grande volume de dados das redes. Embora tenha

capacidade para previsão em tempo real, seus autores não especificam como são realizadas as atualizações do modelo, citando apenas sua capacidade para treinamento utilizando o fluxo de dados.

Seguindo um conceito parecido, a ferramenta *Spring XD*, apresentada por Sabby Anandan et al. [87], é baseada em módulos, sendo cada módulo usado para interagir com uma das tecnologias envolvidas na estrutura da ferramenta. Rodando em seu *backend* de forma transparente para o usuário, o *Spring XD* conta com o *Apache Spark*, *Apache Kafka*, *RabbitMQ* [88], *Hadoop*, *Redis* [89] e *Apache Zookeeper* [90]. Os módulos são acionados através de sequências de comandos baseados em conceito Unix. O fluxo de trabalho pode ser integrado ao *Spark Stream*, possibilitando a utilização da biblioteca *MLLib* para inferência em tempo real. Ele inclui ferramentas de monitoramento e gerenciamento para acompanhar o desempenho do sistema e solucionar problemas conforme necessário. Embora consiga processar dados em tempo real, a atualização do modelo é feita utilizando o modo padrão de minilotes de dados do *Apache Spark.*

Outro *framework* baseado em módulos é o *STREAMER*, apresentado por Sandra Garcia Rodriguez et al. [91]. O *STREAMER* é um sistema de processamento de *Streams* que pode ser instalado em qualquer sistema operacional e aceita integração de algoritmos em diversas linguagens de programação. Assim como em alguns trabalhos já apresentados, o *STREAMER* se baseia em outras ferramentas como, *Apache Kafka* para ingestão de dados, o *InfluxDB* [92] para armazenamento de dados, *Redis* para armazenamento de modelos e o *Kibana* para apresentações de resultados na interface gráfica. O STREAMER é escalável, tolerante a falhas e proporciona alta disponibilidade. Seu objetivo é disponibilizar recursos para que os cientistas se abstraiam da implantação dos modelos para se concentrar apenas nos algoritmos, pré-processamento de dados e funções de avaliação. O *STREAMER* não possibilita a atualização do modelo de forma incremental, sendo ela feita através de minilotes de dados ou utilizando todo o conjunto de dados armazenados no *InfluxDB*.

O *Kafka-ML* apresentado por Cristian Martín et al. [93], é uma estrutura de código aberto para o gerenciamento de *pipelines* de ML por meio de fluxos de dados, que oferece suporte ao *TensorFlow* [94] e *PyTorch* como estrutura de ML. A ferramenta também oferece uma interface *Web* que possibilita que os usuários treinem, comparem e façam inferência em modelos de ML com poucas linhas de código *Python*. Todos os componentes da estrutura podem ser executados como container *Docker* [95], possibilitando assim a portabilidade da estrutura além de gerenciamento e monitoramento por meio de *Kubernetes*

[95]. Embora o *Kafka-ML* utilize o fluxo de dados para treinamento e inferência dos modelos, o treinamento não é feito de forma incremental, mas sim em lotes. Os dados do fluxo são desserializados para criação do lote que então é usado no treinamento do modelo. Outro fator importante é que uma vez treinado o modelo, não é mais possível atualizá-lo, podendo apenas salvá-lo ou implantá-los para posterior inferência.

A ferramenta *Clipper*, apresentada por Daniel Crankshaw et al. [96] é um sistema de previsão de baixa latência voltado para implantações *online*. O objetivo principal é facilitar a implementação de modelos de ML em produção que requerem previsões em tempo real e com baixa latência. O *Clipper* é projetado com uma arquitetura modular que suporta diferentes tipos de modelos de aprendizado de máquina e estratégias de implantação. Ele oferece uma interface unificada que permite aos desenvolvedores implantarem modelos em várias linguagens de programação (*Python*, *Java* etc.) e estruturas de aprendizado de máquina (*TensorFlow*, *PyTorch* etc.). O *Clipper* também possibilita o gerenciamento de recursos, como CPU e memória, para evitar gargalos de desempenho e garantir que os modelos sejam escalonados adequadamente, além de oferecer balanceamento de carga dinâmico para distribuir as previsões de maneira uniforme e evitar sobrecarga em servidores. Embora o *Clipper* permita que vários modelos coexistam e possibilite a implantação gradual de novas versões de modelos, ele não realiza atualizações de forma incremental. Em vez disso, o *Clipper* adota uma abordagem baseada em minilotes para atualizações de seus modelos.

O *TensorFlow Extended* (TFX) apresentado por Denis Baylor et al. [97], é uma plataforma de código aberto desenvolvida para facilitar a implementação e gerenciamento de *pipelines* de *Machine Learning* em produção. Trata-se de uma plataforma de aprendizado de máquina de uso geral baseada no *TensorFlow*, que integra os componentes de análise, transformação, validação de dados, além do treinamento, avaliação e implantação do modelo em produção em uma única plataforma, com configuração e utilitário partilhado, fornecendo assim uma configuração simples e unificada para os usuários. O TFX oferece suporte a diversas estratégias para atualização contínua dos modelos. Uma de suas principais vantagens é a aprendizagem por transferência, onde é possível transferir os parâmetros gerais de uma rede base para uma rede alvo, possibilitando assim a partida a quente, o que reduz consideravelmente o tempo total de treinamento de um modelo. Embora o autor mencione a possibilidade da atualização contínua, no caso de estudo apresentado, foram utilizados lotes de dados com milhões de exemplos na

atualização diária dos modelos, não sendo mencionado se atualizações incrementais são viáveis.

Alok Pareek et al. [98] apresentam o *Striim*, uma plataforma de nível empresarial, com arquitetura de expansão distribuída que é capaz de ingerir dados massivos continuamente de uma ampla variedade de fontes, incluindo banco de dados, arquivos de logs, dispositivos IoT, filas de mensagens entre outros. Ele fornece uma coleção de modelos de aprendizado de máquina para aprendizado supervisionado e não supervisionado, especialmente para previsão em tempo real e detecção de anomalias. O *Striim* não utiliza modelos que consomem muito recurso computacional, como rede neural profunda, que é mais adequado para análise em lote com conjunto de dados e recursos de computação suficientes. Em vez disso, ele escolhe modelos mais eficientes, como floresta aleatória e regressão de processo gaussiano. No *Striim* também é possível usar uma infinidade de bibliotecas de terceiros baseadas em *Java*, como por exemplo o *Weka* e o MOA. Embora apresente uma boa capacidade de generalização, o *Striim* também utiliza o conceito de janelas deslizantes para atualização dos modelos.

Por fim, o *StreamMLOps* apresentado por Mariam Barry et al. [25] propõe uma arquitetura para aprendizagem online utilizando a biblioteca *River* e a plataforma *Dominio* [96], que permite aprendizagem contínua e implantação de modelos de aprendizado de máquina em aplicações em tempo real. Sua arquitetura é sustentada por ferramentas de código aberto, tais como; *Apache Kafka* para ingestão de dados, *Apache Flink* para processamento distribuído, *MLFlow* para versionamento do modelo, *XRAY* para paralelização de experimentos e *River* para aprendizagem incremental. Segundo os autores o *StreamMLOps* é uma ferramenta genérica, porém necessita de adaptações para se adequar a certos tipos de aplicação. O *StreamMLOps* se assemelha à proposta apresentada neste trabalho em diversos aspectos, ambos usam ferramentas de código aberto como o *River* e o *MLFlow*, além de terem módulos e funcionamento parecidos. Entretanto, divergem em relação a aplicabilidade, uma vez que no *StreamMLOps* são necessários ajustes na estrutura, conforme a necessidade de cada ambiente.

3.2 Discussão dos Trabalhos Relacionados

Conclui-se com a revisão da literatura que a maioria dos estudos identificados realiza atualizações dos modelos em modo *offline*, armazenando previamente os dados rotulados e recorre periodicamente a eles, para criar modelos mais alinhados com as informações mais recentes. Além disso, há também pesquisas em que a atualização dos modelos é conduzida em tempo real, recorrendo ao conceito de janelas deslizantes para formar minilotes de dados que são empregados na construção dos novos modelos. Apesar da menção ao aspecto em tempo real nestes trabalhos, a atualização não é realizada de maneira incremental.

Sendo assim, a análise da literatura revelou que, em geral, os estudos empregam a abordagem convencional de MLOps para realizar atualizações periódicas em seus modelos, adotando a estratégia de processamento em lote ou minilotes. Mesmo nos casos em que se faz uso do fluxo de dados para aprimorar os modelos, observa-se uma ausência de implementações incrementais.

Evidencia-se então que, implantar modelos de ML *Stream* em um ambiente de produção é uma tarefa complexa. Entre as inúmeras dificuldades que surgem, destacam-se os desafios relacionados a servir os modelos para fins de inferência, assegurando a capacidade de resposta do sistema mesmo em face de uma alta demanda por previsões. Outras complicações incluem a necessidade de atualizar os modelos incrementalmente, tão logo os rótulos das instâncias estejam disponíveis, bem como a gestão de múltiplas versões dos modelos produzidos ao longo do processo.

A Tabela 1 compara todos os trabalhos relacionados com a proposta.

Tabela 1 - Comparação dos trabalhos relacionados.

Trabalhos	Criação dos modelos	Atualização dos modelos	Processamento do Fluxo	Aplicabilidade	Modelo de Inferência	Modelo de Versionamento
Alberto Bifet et al. (2015)	Stream	Minilotes	Minilotes	Genérica	Distribuído	N/A
W. Jamil et al. (2018)	Stream	*Incremental	Stream	Genérica	N/A	N/A
Donna Xu et al. (2015)	Lotes	Minilotes	Minilotes	Genérica	Distribuído	N/A
Igor L. Markov et al. (2022)	Lotes	Lotes	*Stream	Específica	Centralizado	N/A
Raúl Miñón et al. (2022)	N/A	N/A	Stream	Genérica	Centralizado	N/A
Rohan Ramanath et al. (2021)	Lotes ou Minilotes	Minilotes	Stream	Específica	N/A	N/A
Fabrizio Carcillo et al. (2018)	Lotes	Lotes	Minilotes	Específica	Distribuído	N/A
Janez Kranjc et al. (2017)	Lotes ou Minilotes	Lotes ou Minilotes	Stream	**Genérica	Distribuído	Disco Distributed Filesystem
Mikołaj Komisarek et al. (2020)	Lotes ou Stream	N/A	Minilotes	Específica	Distribuído	N/A
Sabby Anandan et al. (2015)	Minilotes	Minilotes	Minilotes	Genérica	N/A	N/A
Sandra Garcia Rodriguez et al. (2020)	Lotes	Minilotes	Stream	Genérica	N/A	Redis
Cristian Martín et al. (2022)	Lotes	Lotes	Stream	Genérica	Distribuído	*TensorFlow Serving
Daniel Crankshaw et al. (2017)	Lotes	Minilotes	Minilotes	Genérica	Distribuído	N/A
Denis Baylor et al. (2017)	Lotes	Lotes	Stream	Genérica	Distribuído	TensorFlow Serving
Alok Pareek et al. (2019)	Minilotes	Minilotes	Stream	Genérica	N/A	N/A
Mariam Barry et al. (2023)	Stream	Incremental	Stream	**Genérica	Distribuído	MLFlow
Proposta	Stream	Incremental	Stream	Genérica	Distribuído	MLFlow

N/A - Não se aplica ou não foi mencionado no trabalho apresentado.
* - Informação inferida do texto.
** - Necessário modificações na estrutura.

3.3 Considerações Finais

Neste capítulo, foram apresentados trabalhos encontrados na literatura que abordam ferramentas que auxiliam na implementação de modelos de aprendizado *online* em ambientes de produção. Cada trabalho foi detalhadamente descrito, e uma tabela comparativa foi apresentada. As informações coletadas nesta pesquisa contribuíram para o desenvolvimento da proposta apresentada no capítulo subsequente.

No próximo capítulo, será descrita a arquitetura proposta. Inicialmente, será apresentada uma visão geral da estrutura, seguida por uma descrição detalhada de cada módulo que a compõem e da aplicação criada para auxiliar a execução dos testes. Por fim, será realizada uma discussão abordando os pontos relevantes da estrutura desenvolvida.

Capítulo 4

Proposta

O presente trabalho propõe uma arquitetura destinada à implantação de modelos de *Stream Learning* em produção. Esta arquitetura possibilita o escalonamento horizontal do módulo de inferência, viabiliza a atualização incremental do modelo de *Stream Learning* através do fluxo de dados e disponibiliza um processo de versionamento de modelos. Além disso, busca prover mecanismos para otimizar o gerenciamento e a disponibilização de modelos atualizados com uma frequência ideal para o cenário implantado.

4.1 Visão Geral

A arquitetura proposta neste trabalho visa atender as demandas do *Stream Learning* seguindo os princípios e processos do MLOps para implantação de modelos em produção. Para tanto, a arquitetura proposta considera o seguinte cenário de implantação:

- Assume que existem diversos clientes que necessitam fazer predições.
- Assume que o rótulo do evento será disponibilizado com determinado atraso.
- Assume que o sistema de inferência deve responder o mais próximo possível do tempo real.
- Assume que o modelo deve ser atualizado de maneira incremental
- Assume que o modelo deve ser disponibilizado para inferência com a maior frequência e menor atraso possível.

De modo geral, a arquitetura proposta se concentra em resolver três problemas intrínsecos a ambientes de *Stream Learning*:

- **Atender às altas demandas de solicitações de inferência:** Para isso, foi introduzido à estrutura um sistema de escalonamento horizontal, onde é

possível dimensionar a quantidade de *endpoints* de inferência de acordo com demanda do ambiente. Além disso, foi introduzido também um método de carregamento de modelo, onde o módulo de inferência busca os novos modelos direto do repositório de modelos, sem a necessidade de reiniciar as aplicações, garantindo assim uma maior disponibilidade do sistema.

- **Atualização incremental e janela de serialização dos modelos:** Ao contrário das estruturas tradicionais de ML que atualizam os modelos utilizando o conceito lotes ou minilotes, foi introduzido à estrutura proposta um sistema de atualização incremental dos modelos, através de um fluxo de dados contínuo. Também foi introduzido um método de janelamento de serialização de modelos, onde um modelo atualizado pode ser serializado no repositório de modelos e então disponibilizado para inferência, em janelas de eventos quase ideais.
- **Versionamento dos modelos:** O processo de atualização e versionamento dos modelos pode gerar centenas ou até mesmo milhares de versões de um único modelo. Sendo assim, para a arquitetura proposta foi introduzido um sistema de versionamento capaz de gerenciar grande quantidade de modelos, além de permitir o rápido carregamento de uma versão do modelo pelo módulo de inferência.

Considerando o cenário vislumbrado, os desafios inerentes e as propostas de soluções apresentadas, a Figura 3 exibe a visão geral da arquitetura proposta.

Resumidamente a arquitetura proposta se divide em dois módulos, denominados (i) módulo de inferência e (ii) módulo de atualização e versionamento do modelo. O primeiro módulo é responsável por responder as solicitações de predições dos usuários. Já o segundo módulo é responsável por atualizar o modelo de forma incremental, utilizando as instâncias rotuladas e disponibilizar o modelo atualizado no repositório de modelos considerando uma janela de frequência.

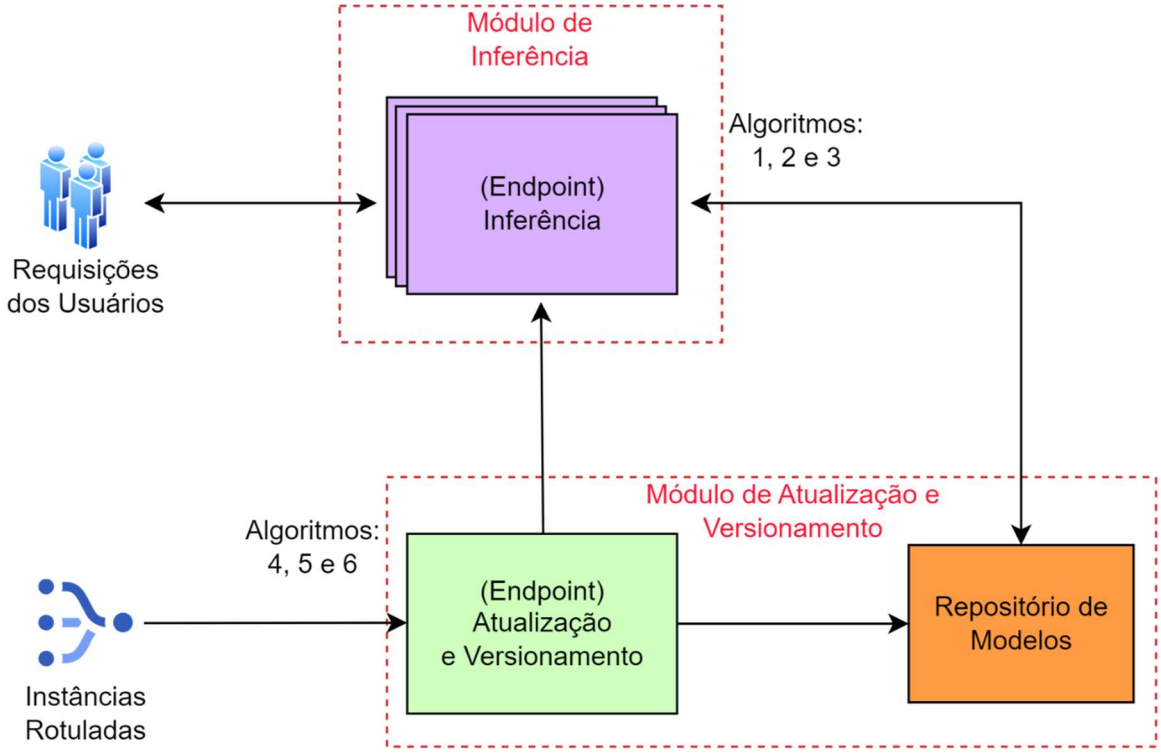

Figura 3 - Visão geral da arquitetura proposta. Fonte: autor.

Ao iniciar, ambos os módulos buscam o modelo original no repositório de modelos. O módulo de inferência recebe requisições dos usuários, realiza a predição e então retorna o valor predito ao usuário. Em um processo paralelo, quando um novo modelo fica disponível no repositório de modelos, o módulo de inferência recebe uma mensagem do módulo de atualização e versionamento com o ID do novo modelo e então, carrega o novo modelo para memória, sem a necessidade de recriar toda a aplicação.

Por outro lado, o módulo de atualização e versionamento recebe do fluxo de dados as instâncias rotuladas e as utiliza para atualizar o modelo de forma incremental. Então, em uma janela de disponibilização parametrizável, ele primeiro serializa o modelo atualizado no repositório de modelos e em seguida envia uma mensagem ao módulo de inferência com o ID do novo modelo para que este possa carregá-lo para memória.

4.2 Módulo de Inferência

O módulo de inferência é responsável por carregar o modelo gerado nas etapas de criação e retreino e disponibilizá-lo para atender as solicitações de inferência dos usuários

do sistema. Para isso, o modelo é encapsulado em um ou mais *endpoints* que recebem as solicitações dos usuários e retornam as predições realizadas pelos modelos carregados na memória. A quantidade de *endpoints* ativos no sistema pode ser escalonada horizontalmente para atender as altas demandas de solicitações dos usuários, melhorando assim o *throughput* do sistema.

Em arquiteturas de ML *Stream*, normalmente, quando um novo modelo está disponível para ser usado na inferência, é necessário excluir os *endpoints* existentes para então criar novos *endpoints* com os modelos já atualizados. Isso pode gerar indisponibilidade do sistema, além de desperdiçar recursos computacionais preciosos.

Em contrapartida, na arquitetura proposta, os *endpoints* continuam sempre ativos. Quando um novo modelo está disponível no repositório de modelos, os *endpoints* têm a capacidade de carregar esse novo modelo para memória sem a necessidade de reiniciar todo o serviço. Com isso, recursos computacionais são poupados e a disponibilidade do sistema permanece intacta.

De forma geral, ao iniciar, os *endpoints* de inferência buscam a última versão do modelo disponível no repositório de modelos e a carregam em memória, tornando-se assim, aptos a receber as solicitações para predição. As solicitações são feitas via requisições de rede e consistem em um vetor de *features* referentes a uma única instância. O *endpoint* processa a solicitação, submetendo as *features* ao modelo salvo na memória, que realiza a predição. Em seguida, o valor predito é retornado ao solicitante, completando o ciclo da requisição de inferência.

Quando um novo modelo é disponibilizado no repositório de modelos, todos os *endpoints* ativos recebem do módulo de atualização e versionamento, uma mensagem via rede com o ID do novo modelo disponível. Então, através de um processo paralelo ao de inferência, carregam o novo modelo na memória. Com isso, as novas solicitações de inferência começam a serem atendidas utilizando o modelo atualizado, sem a necessidade de reiniciar os *endpoints*.

Sendo assim, a capacidade do módulo de inferência de carregar o modelo atualizado diretamente na memória, sem a necessidade de reiniciar os endpoints, economiza recursos computacionais significativos. Além disso, a possibilidade de escalonamento horizontal dos endpoints contribui para a criação de um sistema de predição robusto e de alta performance.

De modo mais detalhado, os Algoritmos 1, 2 e 3, demonstram o funcionamento do módulo de Inferência.

Algorithm 1 - API Inferência - Aplicação

```
modelo ← carregaModelo()
while Aplicacao = True do
    funcaoPredicao(requisicaoHTTP)
    funcaoCarregaModeloAtualizado(requisicaoHTTP)
end while
```

Fonte: autor.

Inicialmente, carrega-se o modelo construído na etapa de desenvolvimento do modelo para a memória do *endpoint*, tornando-o assim apto a receber as requisições HTTP com a instância que se deseja realizar a predição ou as requisições com o ID do novo modelo. As requisições são tratadas pela função a qual foram encaminhadas, sendo assim, quando uma requisição HTTP com a instância que se deseja realizar a predição chega ao *endpoint*, invoca-se a função ***PREDICAO*** (Algoritmo 2), por outro lado, quando uma requisição HTTP com o ID do novo modelo chega ao *endpoint*, invoca-se a função ***CARREGAMODELOATUALIZADO*** (Algoritmo 3).

Algorithm 2 - API Inferêcnia - Função Predição

```
/*Entrada: Requisições HTTP com a instância para predição*/
/*Saída: Valor da predição para o solicitante*/

function PREDICAO(requisicaoHTTP)
    instancia ← requisicaoHTTP
    predicao ← modelo(instancia)
    return predicao
end function
```

Fonte: autor.

Quando uma requisição HTTP com a instância que se deseja realizar a predição é encaminhada ao *endpoint*, a função ***PREDICAO*** (Algoritmo 2) fica responsável por tratar essa mensagem, alocando as *features* da instância recebida em uma variável chamada *instancia*. Em seguida, essa variável é submetida ao modelo salvo na memória para que este realize a predição. Por fim, o valor da predição é retornado ao solicitante fechando o ciclo de função ***PREDICAO*** (Algoritmo 2).

Algorithm 3 - API Inferêcnia - Função Carrega Modelo Atualizado

```
/*Entrada: Requisições HTTP com o ID do novo modelo*/
/*Saída: Confirmação de substituição*/

function CarregaModeloAtualizado(requisicaoHTTP)
    idNovoModelo ← requisicaoHTTP
    modelo ← carregaModelo(idNovoModelo)
    return 'OK'
end function
```

Fonte: autor.

Quando um novo modelo está disponível no repositório de modelos, uma requisição HTTP com o ID desse novo modelo é enviada para a função ***CARREGAMODELOATUALIZADO*** (Algoritmo 3) do *endpoint* de inferência. O ID do novo modelo contido na mensagem recebida, então é salvo em uma variável, que em seguida, será usada para carregar o novo modelo do repositório de modelos. Esse novo modelo substitui o antigo e começa a ser usado pela função ***PREDICAO*** (Algoritmo 2) para realizar as predições subsequentes. Em seguida, uma confirmação de 'OK' é enviada ao *endpoint* de atualização e versionamento do modelo avisando que o carregamento do novo modelo foi realizado com sucesso.

4.3 Módulo de Atualização e Versionamento

O módulo de atualização e versionamento da arquitetura é responsável por atualizar de forma incremental o modelo original e disponibilizar as versões atualizadas do modelo em um repositório de modelos em uma frequência parametrizável. O módulo também é responsável por enviar aos *endpoints* de inferência, o ID do novo modelo salvo no repositório.

A principal vantagem do módulo de atualização e versionamento do modelo é sua capacidade de atualização incremental. Em sistemas tradicionais, a atualização do modelo é realizada utilizando o conceito de lote ou minilote, onde é necessário primeiro armazenar um conjunto de dados para, em seguida, retreinar o modelo com esse conjunto. Além de custoso computacionalmente, o processo de treinamento em lote ou mesmo minilote é mais demorado. Por consequência, o modelo atualizado leva mais tempo para estar disponível,

ocasionando uma resposta mais lenta do sistema, às mudanças de conceitos ocorridas nos dados. Outro fator relevante consiste no armazenamento de dados sensíveis necessários para atualizações que não são feitas de forma incremental. Esse armazenamento pode ser inviabilizado por questões contratuais e de LGPD[1] em cenários onde os processos de ML são realizados por terceiros ou mesmo em localidades não permitidas, como é o caso de armazenamento de dados bancários nacionais fora do país.

Na arquitetura proposta, a atualização do modelo ocorre a cada novo evento de dados, garantindo sua disponibilidade imediata para ser salvo no repositório, independentemente da frequência de disponibilização pré-estabelecida. Essa frequência é um parâmetro crucial para assegurar que a alocação de recursos computacionais não prejudique o desempenho do sistema e refere-se ao intervalo em que os modelos atualizados são salvos no repositório de modelos. Dessa forma, a arquitetura se destaca em comparação aos sistemas tradicionais, pois, graças à atualização incremental constante, o modelo está sempre preparado para a serialização. Além disso, a arquitetura não está sujeita a restrições legais e de segurança da informação relacionadas ao armazenamento de dados sensíveis.

De modo geral, ao iniciar o módulo de atualização e versionamento do modelo, o modelo originalmente criado na fase de desenvolvimento é carregado na memória. A partir desse ponto, o modelo é atualizado de forma incremental, à medida que instâncias rotuladas chegam ao módulo por meio de um serviço de mensageria que é alimentado pelo fluxo de dados gerado pelo sistema encarregado da rotulagem das instâncias.

É importante ressaltar que os rótulos das instâncias podem chegar com atraso, que pode variar de acordo com a aplicação da arquitetura. Por exemplo, em um sistema de detecção de fraude de cartão de créditos, os rótulos só estarão disponíveis depois da confirmação da fraude ou não por um agente externo (humano ou sistêmico), o que pode levar meses. Já em um sistema de recomendação de anúncios, os rótulos podem estar disponíveis em alguns minutos ou segundos, uma vez que a rotulação nesses casos, envolve o clique ou não no anúncio que foi sugerido. Ou seja, o módulo de atualização fica dependente do sistema de rotulagem para realizar o processo de atualização.

Assim que um novo evento já rotulado chega ao *endpoint*, ele é imediatamente usado para atualização do modelo e é contabilizado no contador de eventos. Quando o contador de eventos atingir um valor previamente determinado via parâmetro, inicia-se um processo paralelo para serialização do modelo atualizado no repositório de modelos. Esse parâmetro

[1] Lei Geral de Proteção de Dados

determina a frequência de serialização do modelo e pode ser configurado a qualquer momento. Ele tem forte impacto na performance do sistema como um todo, uma vez que períodos muito curtos de serialização desprendem muito recurso computacional.

Após a serialização do modelo, é enviada uma mensagem pela rede aos *endpoints* do módulo de inferência com o ID do novo modelo disponibilizado. Tanto a disponibilização do modelo, quanto o envio da mensagem com o ID do novo modelo, são processos paralelos ao de atualização do modelo, ou seja, o modelo principal continua sendo atualizado enquanto os demais processos estão sendo executados, evitando assim a interrupção do serviço.

Os Algoritmos 4, 5 e 6, demonstram o funcionamento detalhado do módulo de atualização e versionamento do modelo.

Algorithm 4 - API Atualização e Versionamento - Aplicação

```
contador ← 0
freqSerializacao ← 1000          ▷ Valor padrão para serialização do modelo
modelo ← carregaModel()
while Aplicacao = True do
    funcaoAtualizaModelo(consumerKafka)
    funcaoFrequenciaSerializacao(requisicaoHTTP)
end while
```

Fonte: autor.

Assim como no módulo de inferência, inicialmente, o módulo de atualização e versionamento carrega para a memória o modelo originalmente desenvolvido na etapa de criação do modelo. Quando um novo evento é publicado no serviço de mensageria com a instância rotulada, esses dados são consumidos pelo *endpoint* de atualização e versionamento, que invoca a função ***ATUALIZAMODELO*** (Algoritmo 5).

Por outro lado, quando uma requisição HTTP com o novo valor do parâmetro *freqSerializacao* chega ao *endpoint*, invoca-se a função ***FREQUENCIASERIALIZACAO*** (Algoritmo 6). Neste processo inicial, também é configurado o parâmetro *freqSerializacao* com um valor padrão que define qual será a frequência de disponibilização do modelo. Além disso, é iniciado um contador de eventos que será utilizado na condição que decide quando o modelo será serializado. Com isso, o módulo está pronto para consumir as instâncias rotuladas que são publicadas no serviço de mensageria pelo sistema de rotulação ou as requisições HTTP com o novo valor do parâmetro *freqSerializacao*.

De modo mais detalhado o Algoritmo 5 apresenta a função ***ATUALIZAMODELO.***

Algorithm 5 - API Atualização e Versionamento - Função Atualiza Modelo

```
/*Entrada: Mensagem consumida do tópico Kafka com a instância rotulada*/
/*Saída: Confirmação de atualização e serialização do modelo*/

function AtualizaModelo(consumerKafka)
    instanciaRotulada ← consumerKafka
    modeloAtualizado ← atualizaModelo(instanciaRotulada)
    contador ← contador + 1
    if contador%freqSerializacao = 0 then
        idModeloSerializado ← serializaModelo(modeloAtualizado)
        for endPoint ∈ moduloInferencia do
            enviaID(idModeloSerializado)
        end for
    end if
    return 'OK'
end function
```

Fonte: autor.

Quando consumida uma mensagem do tópico do serviço de mensageria contendo a instância já rotulada, a função ***AtualizaModelo*** (Algoritmo 5), aloca a *feature* e a *label* recebido, em uma variável chamada de *instanciaRotulada* e então utiliza essa variável para atualizar o modelo carregado na memória. Em seguida, é somado o valor 1 ao contador de evento, contabilizando assim mais processamento realizado. Então, verifica-se, se o resto da divisão entre o contador e a *freqSerializacao* é igual a zero. Caso a condição retorne um valor falso, o processo de atualização desta instância se encerra. Porém, caso a comparação retorne um valor verdadeiro, modelo atualizado é serializado no repositório de modelos. Em seguida envia-se o ID deste novo modelo a todos os *endpoints* ativos no módulo de inferência, para que estes possam carregar o novo modelo para a memória.

O Algoritmo 6 demonstra de forma mais detalhada o funcionamento da função ***FrequenciaSerializacao***.

Algorithm 6 - API Atualização e Versionamento - Função Freq. Serialização

```
/*Entrada: Requisições HTTP com o novo valor  do parâmetro*/
/*Saída: Confirmação de recebimento*/

function frequenciaSerializacao(requisicaoHTTP)
    freqSerializacao ← requisicaoHTTP
    return 'OK'
end function
```

Fonte: autor.

A função chamada ***FREQUENCIASERIALIZACAO*** (Algoritmo 6) é responsável por configurar o parâmetro que define a quantidade de eventos que deve ocorrer para que um novo modelo seja serializado no repositório de modelos. Como já mostrado anteriormente, inicialmente esse parâmetro é configurado com um valor padrão, porém sua configuração posterior pode ser realizada apenas enviando uma mensagem com o novo valor do parâmetro. Quando essa requisição é recebida pela função ***FREQUENCIASERIALIZACAO*** (Algoritmo 6), o novo valor é atribuído ao parâmetro *freqSerializacao* que será utilizado em serializações subsequentes.

A seção seguinte apresentará a aplicação responsável por gerar os fluxos de dados para inferência e atualização dos modelos, além de coletar os logs da estrutura e gerar os arquivos com os resultados.

4.4 Aplicação geradora dos fluxos de dados

Para conduzir os testes, foi desenvolvida uma aplicação *Python* responsável por simular as requisições dos usuários, calcular a métrica com base nos valores probabilísticos de predição retornados do módulo de inferência, realizar a rotulagem das instâncias e publicá-las no tópico *Kafka*. Além disso, a aplicação monitora a fila *Kafka*, mede o tempo total de consumo das instâncias por ambos os módulos, e gera os arquivos com os resultados dos testes.

A aplicação foi alocada na VM de gerência e, de forma mais detalhada, tem como função percorrer todas as linhas dos *datasets* e, utilizando técnicas de processamento paralelo (seis processos simultâneos), envia os dados como requisições HTTP para o módulo de inferência. Este, por sua vez, utiliza as *features* para realizar a predição (probabilidade de ser 0 ou 1) e retorna o valor predito à aplicação, que então calcula a métrica com base no valor retornado e na *label* verdadeira.

Neste trabalho, adotou-se como métrica de avaliação o *roc_auc_score* [99] da biblioteca *Scikit-Learn*, para a medição da performance preditiva dos modelos. Essa métrica calcula a área sob a curva ROC (*Receiver Operating Characteristic*), que é um gráfico que representa a relação entre a taxa de verdadeiros positivos e a taxa de falsos positivos para diferentes limiares de decisão. Um valor de *roc_auc_score* próximo a 1

indica uma excelente capacidade do modelo de discriminar entre as classes, enquanto um valor próximo a 0.5 sugere uma performance semelhante à de uma classificação aleatória. Essa métrica é comumente utilizada em ambiente de *Stream Learning*, pois fornece uma medida robusta da performance preditiva do modelo ao longo do tempo.

O cálculo da métrica *roc_auc_score* foi feito utilizando uma população de mil eventos. De forma mais detalhada, todo valor de predição retornado pelo sistema de inferência, juntamente com a *label* verdadeira, são armazenados em vetor. Esse vetor inicia-se vazio e vai acumulando valores conforme as predições são realizadas. Assim, quando o número de valores armazenados no vetor se torna divisível por mil, um processo paralelo é iniciado. Neste processo são selecionados os últimos mil valores armazenados para realização do cálculo da métrica *roc_auc_score*. O resultado, é então armazenado em um segundo vetor que será utilizado na geração dos gráficos da performance preditiva do modelo. Vale ressaltar que para os testes de *baseline* o processo é basicamente o mesmo, o que muda neste caso, é que a predição não é feita na arquiteta, mas sim localmente.

Nos testes que envolvem todo o *pipeline*, as mesmas *features* usadas na inferência, juntamente com a *label* verdadeira, são enviadas para um tópico no *Kafka*, que é consumido pelo módulo de atualização e versionamento. Esse módulo atualiza o modelo original com base nesses dados e, conforme a frequência de disponibilização definida, serializa o modelo atualizado para que o módulo de inferência o carregue na memória.

Simultaneamente, a aplicação monitora regularmente o tópico *Kafka* para verificar as instâncias na fila. Ao término da geração dos fluxos de dados, a aplicação calcula o tempo total de consumo das instâncias e coleta outras informações sobre a configuração da estrutura. Todos os dados coletados são então armazenados em arquivos para a avaliação dos resultados.

A seção seguinte apresenta de forma detalhada o protótipo desenvolvido para execução dos testes.

4.5 Protótipo

Para validar a arquitetura proposta neste trabalho, foi desenvolvido um protótipo para execução dos testes de aplicabilidade do sistema em ambiente de produção. Para isso, foi criado um ambiente que simula uma aplicação real, como mostra a Figura 4.

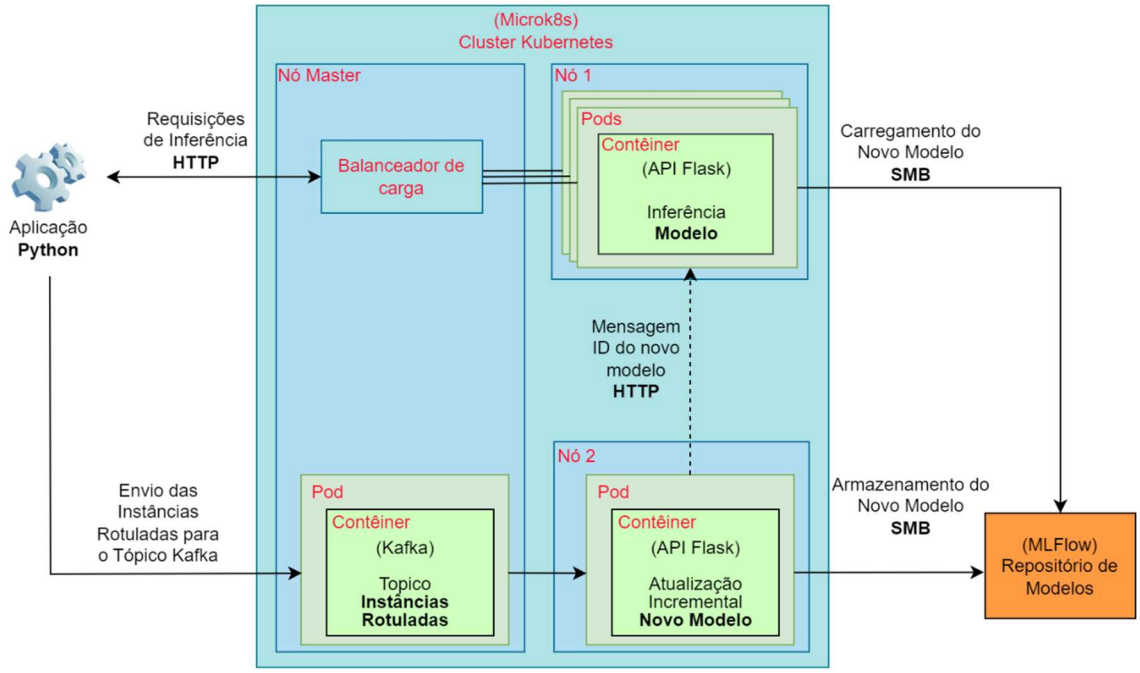

Figura 4 - Visão geral do protótipo. Fonte: autor.

Os fluxos de dados que alimentam os módulos de inferência e de atualização e versionamento do modelo mostrados na figura acima, foram gerados através da aplicação *Python* mencionada na subseção anterior, que serializa as instâncias dos conjuntos de dados, respeitando sua ordem cronológica.

Os *endpoints* de inferência e atualização e versionamento foram desenvolvidos com APIs *Flask Python* [100], encapsuladas em contêineres e orquestradas via *Kubernetes*. O *cluster Kubernetes* gerencia também um contêiner com o *Apache Kafka*, responsável por receber as mensagens com as instâncias rotuladas, que serão consumidas pelo *endpoint* de atualização e versionamento. O *Kafka* é uma plataforma distribuída de streaming de eventos que permite publicar, armazenar e processar fluxos de registros em tempo real. Ideal para sistemas que precisam lidar com grandes volumes de dados rapidamente. A

conteinerização e a orquestração desses serviços possibilitam a criação de uma arquitetura robusta e eficiente, garantindo também a portabilidade para diversos sistemas.

Para criação e gerenciamento do *cluster Kubernetes* foi utilizada a ferramenta *Microk8s* [101], que possibilitou a implantação de um *cluster Kubernetes on-premise*. O *Microk8s* disponibiliza ferramentas importantes, como o balanceador de cargas, responsável por distribuir as inúmeras requisições recebidas entre os *endpoints* de inferência, possibilitando assim o escalonamento horizontal do módulo de inferência. Outro aspecto importante do *Microk8s* é a facilidade de gerenciamento do *cluster*, permitindo uma fácil adição ou remoção de nós, possibilitando assim, o gerenciamento de recursos computacionais. Além disso, o *Microk8s* disponibiliza também, uma fácil migração para serviços de *Cloud*, caso exista a necessidade de migração para um ambiente com maior capacidade computacional.

Conforme ilustrado na Figura 4, o *cluster Kubernetes* é composto por três nós distintos: o nó *master*, o nó 1 e o nó 2, cada um com atribuições específicas. É importante destacar que essa configuração foi utilizada exclusivamente para os testes, podendo a quantidade de nós ser ajustada conforme a demanda. O nó *master* é responsável por balancear a carga das instâncias que chegam ao módulo de inferência, além de hospedar o contêiner com o *Kafka* e suas dependências. O nó 1 é dedicado à hospedagem das aplicações conteinerizadas voltadas para inferência (*endpoints* de inferência), enquanto o nó 2 acomoda a aplicação conteinerizada de atualização e versionamento do modelo (*endpoint* de atualização e versionamento).

Cada nó corresponde a uma VM (máquina virtual) dotada das seguintes configurações de hardware e software: 16 GB de memória RAM, 8 vCPUs, 100 GB de armazenamento em disco e o sistema operacional *Ubuntu Desktop* 20.04. Porém, é importante ressaltar que os contêineres, que são alocados nos nós conforme função, sofreram limitações de *hardware* durante a realização dos experimentos, visando assim, adequar suas respectivas performances às necessidades de cada teste, evidenciando assim, pontos importantes para o entendimento do funcionamento da arquitetura. Essas limitações serão mais bem detalhadas no decorrer do Capítulo 5.

Foi utilizada também, uma quarta VM (VM de gerenciamento) responsável pela geração dos fluxos de dados, gerenciamento do *cluster Kubernetes* e hospedagem do repositório de modelos. Essa VM é dotada das seguintes configurações: 24 GB de memória RAM, 16 vCPUs, 150 GB de armazenamento em disco e o sistema operacional *Ubuntu Desktop* 20.04.

Para o armazenamento e gerenciamento das versões dos modelos de ML no protótipo, foi utilizada a ferramenta *MLFlow*. Entre as principais vantagens do uso do *MLFlow*, destaca-se sua capacidade de gerenciamento de experimentos, onde um registro sistemático permite acompanhar o progresso e os resultados das diferentes iterações dos modelos, garantindo a reprodutibilidade dos resultados. Além disso, a implantação da ferramenta é significativamente simples, facilitando sua utilização. Contudo, para o problema em questão, a principal vantagem do uso do *MLFlow* é o gerenciamento das versões dos modelos. O *MLFlow* oferece uma maneira robusta de gerenciar o controle de versões de modelos, código-fonte e dados, permitindo o rastreamento das mudanças ao longo do tempo. Isso proporciona ao protótipo uma excelente capacidade de controle de versões, além de facilitar a disponibilização dos modelos retreinados para o módulo de inferência.

A serialização e carregamento do modelo no *MLFlow* ocorre via protocolo SMB (*Server Message Block*). Como mostrado na Figura 7, o *MLFlow* foi hospedado na VM de gerenciamento e geração de fluxo de dados e embora não faça parte do *cluster Kubernetes*, seu repositório está compartilhado via rede entre todos os nós do *cluster*, permitindo assim, que qualquer contêiner dentro do *cluster* consiga ter acesso ao repositório de maneira compartilhada. Com isso, o modelo salvo pelo *endpoint* de atualização e versionamento, fica disponível para os *endpoints* de inferência, imediatamente após a finalização do processo de serialização.

Como premissa primordial, um sistema de *Stream Learning* deve trabalhar com algoritmos de ML que permitam serem atualizados de maneira incremental. Por isso, foi utilizada a biblioteca *River*, responsável pela criação e atualização dos modelos utilizados no protótipo. A biblioteca *River* é uma ferramenta que permite aprendizado de máquina incremental em *Python*. Ela se destina a lidar com fluxos de dados contínuos e não se encaixa no paradigma tradicional de aprendizado de máquina, que assume um conjunto de dados estáticos. Isso é fundamental para aplicações em que os dados estão evoluindo constantemente, adequando-se perfeitamente ao objetivo proposto neste trabalho.

O *River* oferece uma variedade de algoritmos de aprendizado de máquina, como regressão linear, classificação, agrupamento e detecção de anomalias, todos projetados para serem eficientes em termos de memória e processamento. Além disso, a biblioteca fornece uma interface simples e coesa para treinar, avaliar e usar modelos em cenários de aprendizado incremental.

A Figura 5 apresenta o diagrama de sequência com uma visão geral do funcionamento do protótipo criado. Nela é possível acompanhar a sequência dos eventos ocorridos no sistema como um todo.

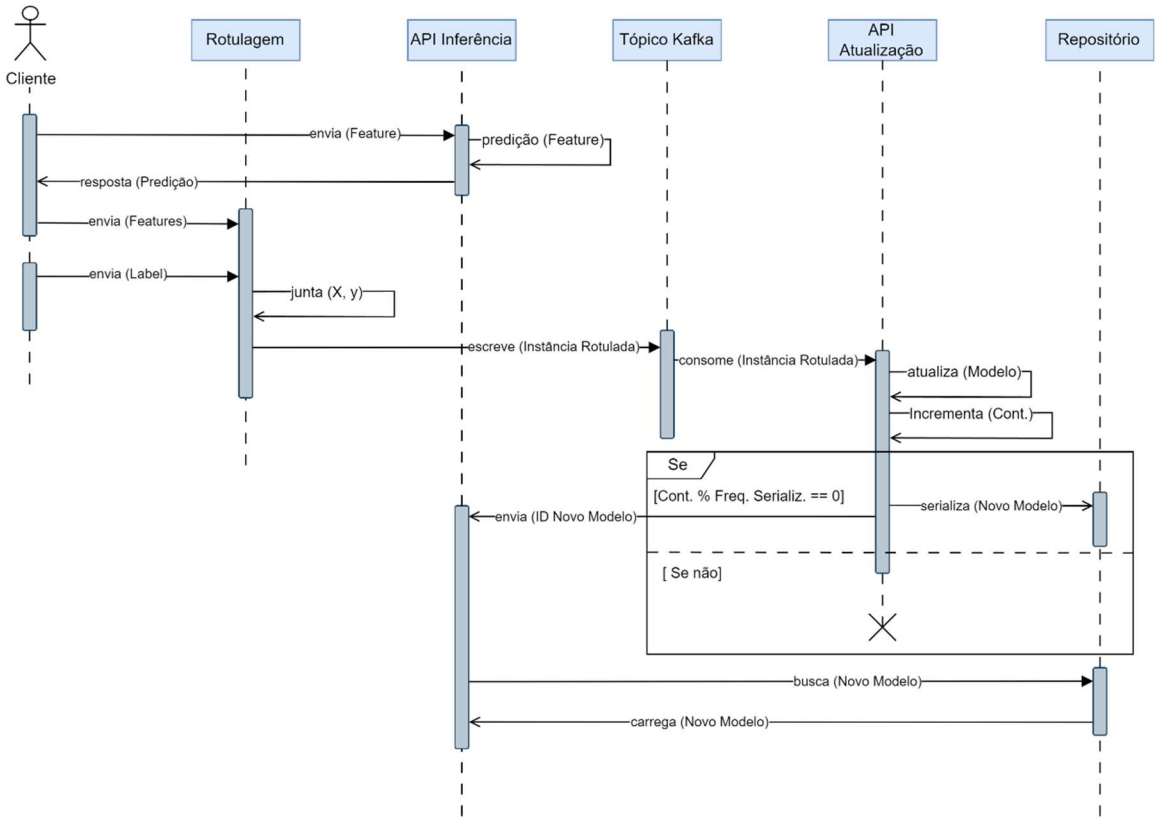

Figura 5 - Diagrama de Sequência. Fonte: autor.

Inicialmente, uma requisição HTTP com as *features* de uma instância, é enviada pelo Cliente à API de Inferência (evento "**envia(Feature)**" da Figura 5), a API recebe os dados da requisição, submete ao modelo para realização da predição e retorna ao Cliente o valor da predição (evento "**resposta(Predição)**" da Figura 5). Após o recebimento da predição, o cliente envia as *features* para a entidade de Rotulação (evento "**envia(Feature)**" da Figura 5). Quando o rótulo da instância fica disponível, esta também é enviada para a entidade Rotulação (evento "**envia(Label)**" da Figura 5), que fará a junção das *features* e da *label*. Em seguida, a instância rotulada é enviada ao tópico *Kafka* (evento "**escreve(Instância Rotulada)**" da Figura 5), então, o tópico *Kafka* é consumido pela API de Atualização (evento "**consome(Instância Rotulada)**" da Figura 5) que utiliza a instância rotulada para atualizar o modelo em memória. Após a atualização incrementa-se o contador que será utilizado na condição "Se". Caso o resto da divisão do contador pelo parâmetro "frequência de Serialização" seja diferente de zero, o processo todo é encerrado.

Entretanto, se o resto da divisão do contador pelo parâmetro “frequência de Serialização” for igual a zero, a API de atualização e versionamento do modelo, serializa o modelo atualizado no *MLFlow* (repositório de modelo) (evento “**serializa(Novo Modelo)**” da Figura 5) e em seguida, envia uma mensagem com o ID do modelo recém serializado, para as APIs de Inferência (evento “**envia(ID do Modelo)**” da Figura 5), que ao receber a mensagem utiliza o ID para carregar o modelo salvo no *MLFlow* em sua memória.

4.6 Discussão da Proposta

A arquitetura proposta apresenta solução para problemas intrínsecos do *Stream Learning* como: (i) escalonamento horizontal dos *endpoints* de inferência, (ii) carregamento dos modelos sem a necessidade de reinicialização das aplicações, (iii) atualização incrementais dos modelos, (iv) frequência de serialização dos modelos quase ótimas e (v) gerenciamento e disponibilização das versões dos modelos.

A proposta utiliza microsserviços conteinerizados e é orquestrada pelo *Kubernetes*, o que gera uma arquitetura ágil e flexível, viabilizando assim, sua implementação em diversos ambientes, desde *on-premises* até sistemas de nuvem. Isso porque, a adoção de um sistema de microsserviços conteinerizados proporciona módulos leves computacionalmente e de simples implantação, conferindo à proposta uma notável flexibilidade e portabilidade.

A estrutura é dividida em dois módulos: módulo de inferência e módulo de atualização e versionamento do modelo. No que diz respeito ao módulo de inferência proposto, destaca-se o método de carregamento dos modelos atualizados para memória do *endpoint*, diretamente do repositório de modelos, eliminando a necessidade de recriar todo o *endpoint*. Além disso, o módulo de inferência possui a capacidade de escalonamento horizontal, permitindo a instanciação de várias réplicas do modelo simplesmente aumentando o número de *endpoints* ativos, resultando em um sistema mais estável e robusto.

Em paralelo, o módulo de atualização e versionamento do modelo permite atualização incremental do modelo. Com isso, é possível obter modelos atualizados com um menor custo computacional do que utilizando treinamento em lote ou minilotes.

Também é possível configurar uma janela de frequência de serialização do modelo, permitindo assim que modelos atualizados estejam à disposição dos *endpoints* de inferência em períodos muito menores de tempo do que em sistemas de atualizações tradicionais. Consequentemente, o sistema torna-se mais adaptado às mudanças ocorridas nos dados.

Destaca-se também a capacidade da estrutura proposta em gerenciar as diversas versões dos modelos gerados no processo de atualização e versionamento do modelo. Isso possibilita o carregamento dos modelos atualizados para o módulo de inferência de modo eficaz, garantindo assim, uma rápida substituição do modelo antigo pelo novo em todos os *endpoints* ativos no sistema.

Outro aspecto importante, é que a arquitetura proposta possui um baixo nível de dependência entre as APIs de ML. Isso significa que, embora a escolha das ferramentas utilizadas na criação do protótipo seja considerada a mais adequada no momento de sua elaboração, nada impede que outras ferramentas, linguagens de programação e *learners* possam ser integrados à arquitetura.

Portanto, a estrutura proposta introduz de maneira satisfatória as premissas de funcionamento e desempenho inerentes a ambientes de *Stream Learning.*

4.7 Considerações Finais

Neste capítulo, foi apresentada de forma detalhada a proposta desenvolvida nesta pesquisa, que tem como objetivo propor uma estrutura para implantação de modelos de *Stream Learning* em produção, com capacidade de atualizar os modelos de forma incremental e disponibilizá-los para inferência e uma frequência parametrizável.

Inicialmente, foi oferecida uma visão geral da arquitetura proposta, seguida por uma explicação detalhada dos módulos que a compõem e a apresentação do protótipo desenvolvido para validação da proposta. Além disso, foi promovida uma discussão abordando os pontos relevantes introduzidos na estrutura proposta, que visam mitigar os desafios impostos pelo *Stream Learning*.

No próximo capítulo, serão apresentados os testes realizados para validar a estrutura proposta. Inicialmente, serão apresentados os *datasets* utilizados e a abordagem dos

experimentos realizados. Em seguida, serão apresentados os resultados preliminares, seguidos por uma discussão dos resultados encontrados.

Capítulo 5

Resultados

Neste capítulo, serão apresentados os resultados obtidos no âmbito do projeto desenvolvido.

As avaliações realizadas foram elaboradas visando responder às seguintes questões de avaliação (QA):

- QA1 - Qual é a performance preditiva do modelo em um cenário de atualização incremental?
- QA2 - Qual é a escalabilidade de inferência da abordagem proposta sem a execução de atualizações incrementais?
- QA3 - Qual é o impacto da frequência de versionamento do modelo no *throughput* e na performance preditiva do sistema?
- QA4 - Qual é o impacto do número de *endpoints* no *throughput* e na performance preditiva do sistema?

É importante destacar que, embora tenham sido identificados trabalhos semelhantes ao proposto, para utilizá-los como referência nos resultados obtidos, seriam necessárias diversas adaptações em suas estruturas para que se adequassem à metodologia adotada nos testes desta pesquisa. Essas adaptações, no entanto, inviabilizaram o uso desses trabalhos como comparativo direto. Diante disto, optou-se por criar um *baseline* que simula um ambiente ideal, servindo como referência para a avaliação dos resultados.

Sendo assim, esta seção aborda as características dos *datasets* e dos *learners* utilizados nos experimentos, a descrição da abordagem dos testes, a descrição da aplicação geradora dos fluxos de dados, a criação dos modelos e *baseline*, a avaliação da proposta e uma análise detalhada dos resultados alcançados.

5.1 Datasets

Para a realização dos experimentos, foram empregados três *datasets* distintos, cada um contendo desvios de conceito nos dados. Esses *datasets* foram escolhidos com o intuito de testar a robustez e a eficácia dos métodos propostos em diferentes cenários. Dois dos *datasets* são sintéticos e buscam simular condições controladas de variação de conceito, enquanto o terceiro é extraído do mundo real, fornecendo um contexto mais complexo e dinâmico para os testes.

Os *datasets* sintéticos utilizados nos experimentos são o AGR_a [102] e AGR_g [102], ambos simulam o problema de determinar se um empréstimo deve ou não ser concedido a um cliente bancário. Estes *datasets* foram gerados usando o gerador de dados *Agrawal* do MOA e contém seis atributos nominais e três contínuos, mapeados em dois rótulos, totalizando assim 9 atributos que descrevem as características das instâncias (*features*) e um atributo alvo (*label*). Para simular os desvios de conceito foram atribuídos fatores de perturbação extras para cada atributo gerando assim três desvios de conceitos abruptos para o *dataset* AGR_a e três desvios de conceito graduais para o *dataset* AGR_g.

Já o *dataset* extraído do mundo real, trata-se do *Youchoose* [103]. O *dataset Youchoose* compreende uma série de sessões capturadas por um varejista *online*, nas quais são registrados os cliques feitos pelos usuários durante suas interações. Em algumas dessas sessões, ocorrem também eventos de compra, mostrando que o usuário concluiu a sessão comprando algum item na loja virtual. Esses dados foram coletados ao longo de vários meses em 2014 e correspondem a 16 atributos que descrevem os eventos ocorridos durante as sessões (*features*), além de um atributo alvo (*label*) que indica se a compra foi realizada ou não.

As características dos desvios de conceito de cada *dataset* será mostrado na seção 5.3 deste capítulo, logo após a apresentação dos *learner* escolhidos para esse trabalho.

5.2 *Learners* utilizados nos experimentos

Modelos de ML *stream* são algoritmos de aprendizado de máquina desenvolvidos para processar e analisar dados em tempo real, conforme esses dados são gerados ou recebidos. Há diversas ferramentas que oferecem suporte para esses modelos, sendo que uma das mais reconhecidas é o *River*. Como já explicado anteriormente, o *River* é uma biblioteca *Python* destinada a aprendizado de máquina em fluxos de dados (*stream learning*). Ela permite a construção de modelos que aprendem e fazem previsões em tempo real, um dado de cada vez, sem a necessidade de processar todo o conjunto de dados de uma só vez. Para os testes em questão foram selecionados três *learners* do *River* com características distintas, descritos nos parágrafos a seguir.

Naive Bayes (NB) [104]: este é um classificador probabilístico que se baseia no Teorema de *Bayes*, assumindo que os atributos são condicionalmente independentes dados à classe. Essa suposição simplifica os cálculos e permite que o modelo seja eficiente tanto no treinamento quanto na inferência. A atualização do modelo é incremental, ajustando as estimativas de probabilidade com cada novo exemplo de dados recebido. Devido à sua simplicidade, o *Naive Bayes* é robusto a ruídos nos dados, mas seu desempenho em ambientes com grandes desvios de conceito nos dados pode ser comprometido devido sua baixa plasticidade. Este *learner* não possui parâmetros a serem configurados.

Adaptive Random Forest Classifier (ARF) [105]: este é um modelo de *ensemble* composto por múltiplas árvores de decisão, projetado para se adaptar rapidamente a mudanças no conceito dos dados (alta plasticidade). Este modelo não apenas treina as árvores de forma incremental, mas também monitora o desempenho de cada árvore, substituindo ou ajustando aquelas que apresentam queda de desempenho devido a alterações nos dados. A diversidade das árvores é mantida por meio de técnicas de amostragem ponderada e substituição de árvores, o que aumenta a robustez e a precisão do modelo. No entanto, essa complexidade adicional implica em maior consumo de memória e processamento comparado a modelos mais simples, tornando-os mais pesados computacionalmente.

A configuração dos parâmetros utilizados no ARF é a seguinte:

- n_models = 10
- drift_detector = ADWIN

- warning_detector = ADWIN
- split_criterion = info_gain
- Demais parâmetros = defaults

Hoeffding Tree Classifier (HT) [106]: este *learner* utiliza o teorema de *Hoeffding* para construir árvores de decisão de maneira incremental. Este modelo é eficiente em termos de memória, armazenando apenas estatísticas necessárias para tomar decisões de divisão nos nós da árvore. As divisões são realizadas de forma assíncrona, conforme novos dados são recebidos e são garantidas estatisticamente pelo *bound* de *Hoeffding*. O *Hoeffding Tree Classifier* é particularmente adequado para fluxos de dados contínuos e grandes volumes de dados. Além disso, o *Hoeffding Tree Classifier* possui a melhor relação elasticidade/plasticidade dos três, se adequando bem aos desvios de conceito dos dados sem esquecer totalmente o conhecimento adquirido anteriormente.

A configurações dos parâmetros utilizados no HT é a seguinte:

- grace_period = 200
- split_criterion = info_gain
- delta = 1e-07
- max_size = 100.0
- Demais parâmetros = defaults

Em suma, cada um desses modelos oferece um conjunto distinto de vantagens: o *Naive Bayes* se destaca pela simplicidade e eficiência, o *Adaptive Random Forest* pela adaptabilidade e robustez, e o *Hoeffding Tree* pela eficiência de memória e atualizações incrementais significativas.

5.3 Curva característica dos desvios de conceito dos *datasets*

Como já mencionado anteriormente, os três *datasets* utilizados nos experimentos possuem desvios de conceitos em seus dados. No caso dos *datasets* sintéticos, é conhecido que existem três desvios abruptos no *dataset* AGR_a e três desvios graduais no *dataset* AGR_g. Já o *dataset Youchoose*, extraído do mundo real, não possui um padrão de desvio de conceito conhecido e, portanto, suas características não são bem definidas como nos sintéticos.

A fim de facilitar a visualização dos desvios de conceitos dos *datasets* foram gerados gráficos com a evolução da métrica, mostrando as curvas características destes desvios para cada *learner* e *dataset* escolhidos para este trabalho. Para a criação desses gráficos, foi treinado um modelo de cada *learner* utilizando 0,1% das instâncias iniciais de cada *datasets* (500 instâncias para os sintéticos e 1000 instâncias para o real). Assim, é possível observar a variação da métrica quando todas as instâncias dos conjuntos de dados são submetidas aos seus respectivos modelos.

Como já explicado anteriormente, cada ponto no gráfico corresponde à medida da métrica *roc_auc_score* utilizando uma população de mil eventos. Além disso, é importante ressaltar que as curvas características mostradas a seguir, sofrem pequenas variações conforme o *learner* utilizado. Essas variações se devem às particularidades intrínsecas de cada *learner,* fazendo com que a métrica responda de forma diferente no mesmo *dataset*. Sendo assim, as Figuras 6, 7 e 8, apresentam as características dos desvios de conceito contido no *dataset* AGR_a para os *learners*: *Naive Bayes*, *Adaptive Random Forest* e *Hoeffding Tree*, respectivamente.

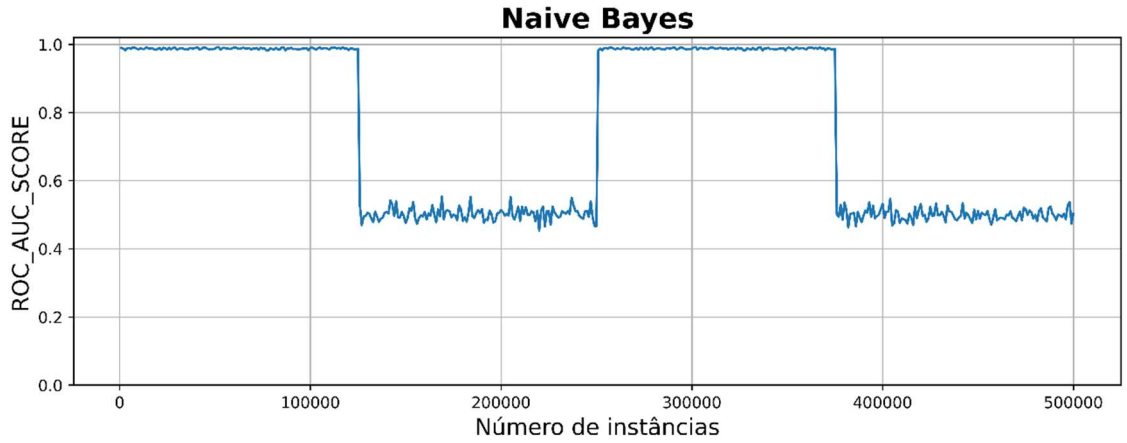

Figura 6 - Evolução da métrica roc_auc_score, mostrando os desvios de conceito do dataset AGR_a, para o learner NB. Fonte: autor.

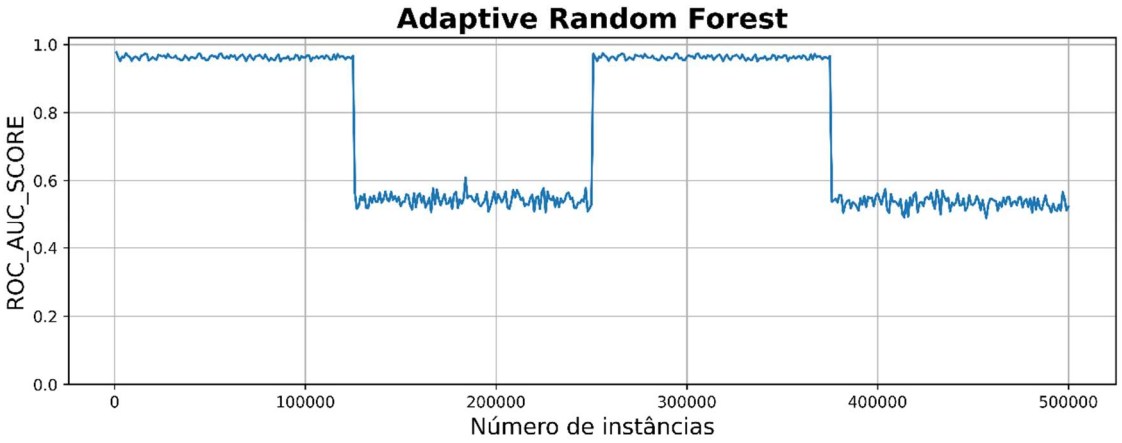

Figura 7 - Evolução da métrica roc_auc_score, mostrando os desvios de conceito do dataset AGR_a, para o learner ARF. Fonte: autor.

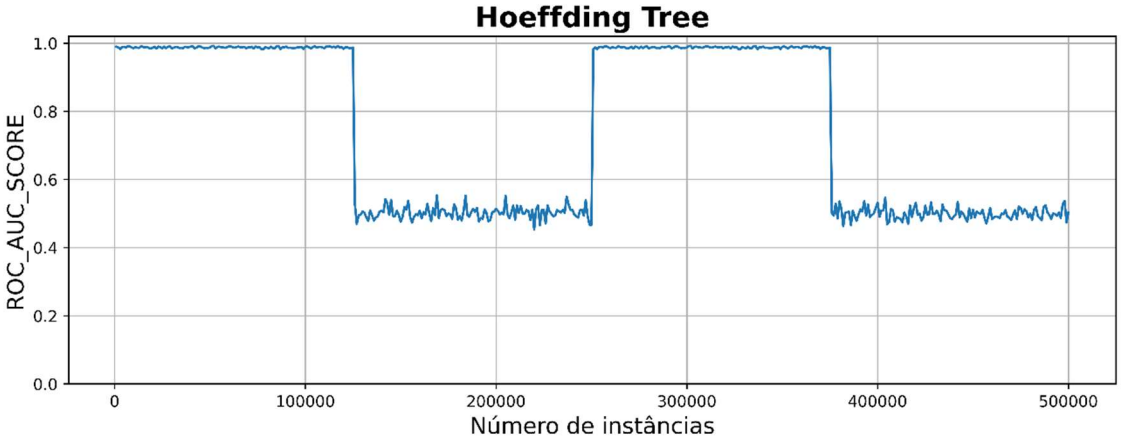

Figura 8 - Evolução da métrica roc_auc_score, mostrando os desvios de conceito do dataset AGR_a, para o learner HT. Fonte: autor.

Seguindo o mesmo princípio as Figuras 9, 10 e 11, mostram as características dos desvios de conceito contido no *dataset* AGR_g para os *learners*: *Naive Bayes*, *Adaptive Random Forest* e *Hoeffding Tree*, respectivamente.

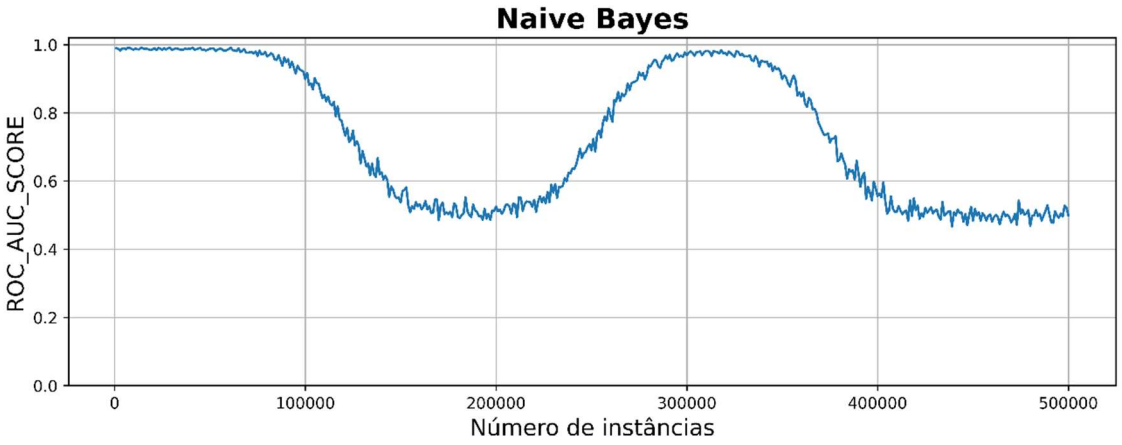

Figura 9 - Evolução da métrica roc_auc_score, mostrando os desvios de conceito do dataset AGR_g, para o learner NB. Fonte: autor.

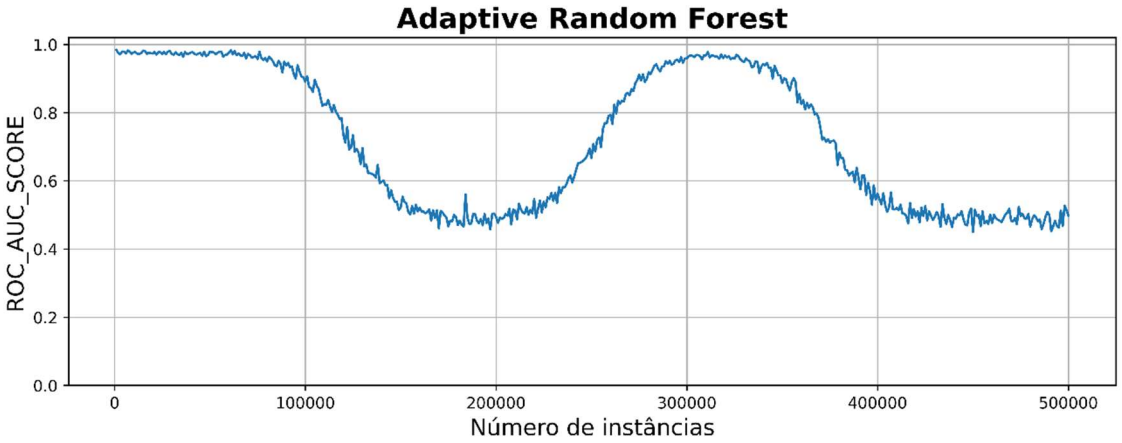

Figura 10 - Evolução da métrica roc_auc_score, mostrando os desvios de conceito do dataset AGR_g, para o learner ARF. Fonte: autor.

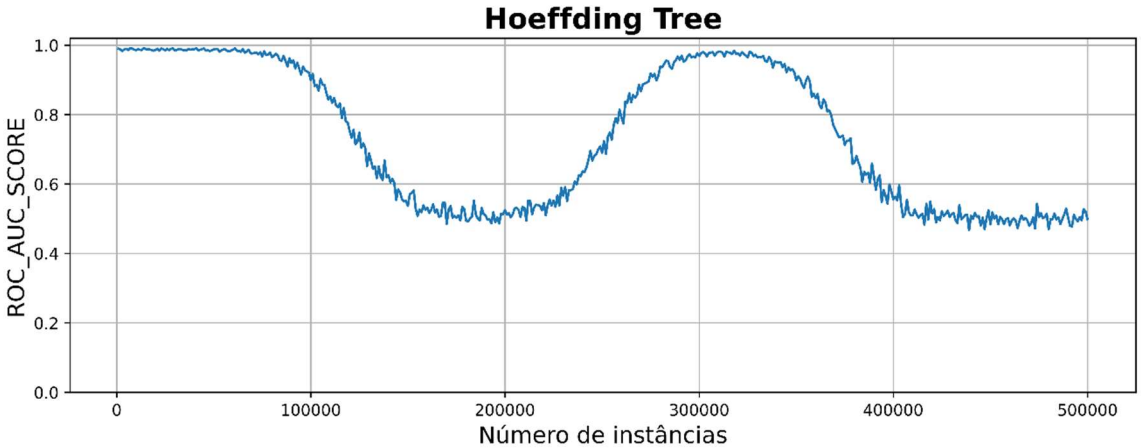

Figura 11 - Evolução da métrica roc_auc_score, mostrando os desvios de conceito do dataset AGR_g, para o learner HT. Fonte: autor.

Por fim, as Figuras 12, 13 e 14, mostram as características dos desvios de conceito contido no *dataset* Youchoose para os *learners*: *Naive Bayes*, *Adaptive Random Forest* e *Hoeffding Tree*, respectivamente.

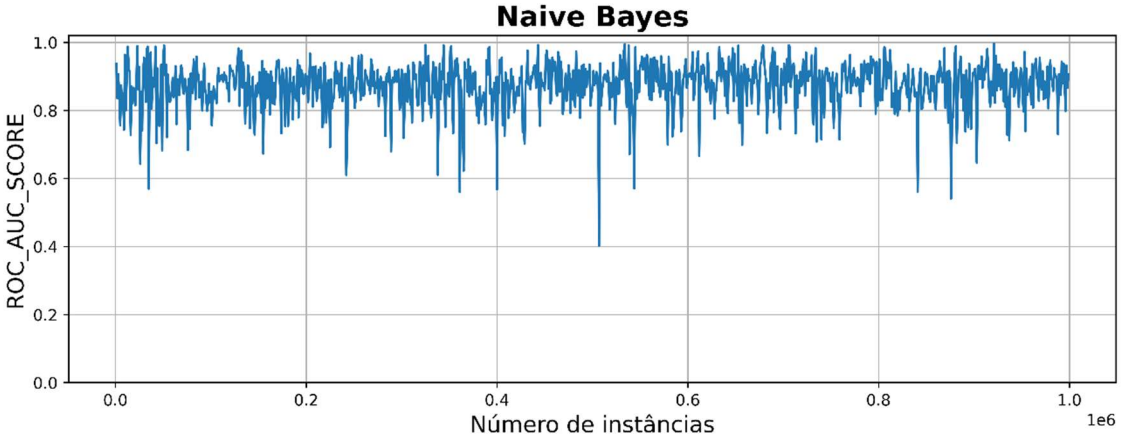

Figura 12 - Evolução da métrica roc_auc_score, mostrando os desvios de conceito do dataset Youchoose, para o learner NB. Fonte: autor.

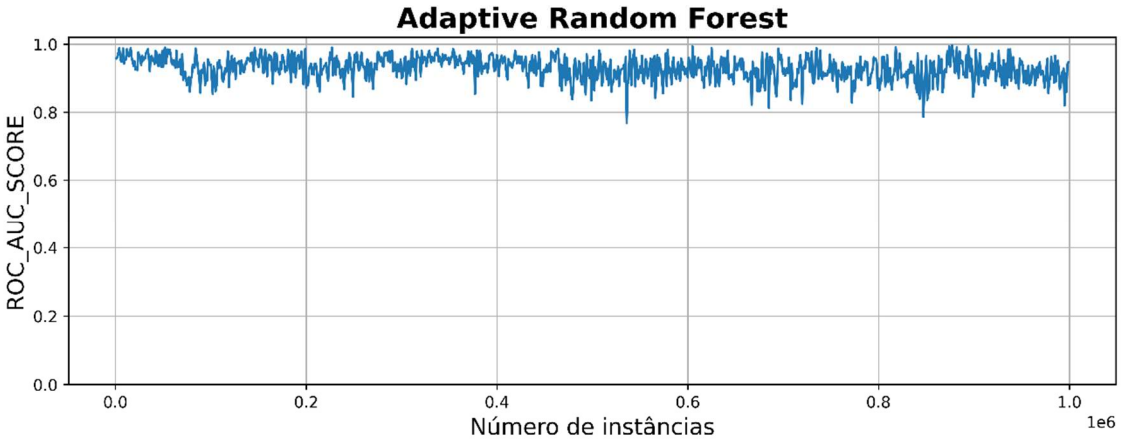

Figura 13 - Evolução da métrica roc_auc_score, mostrando os desvios de conceito do dataset Youchoose, para o learner ARF. Fonte: autor.

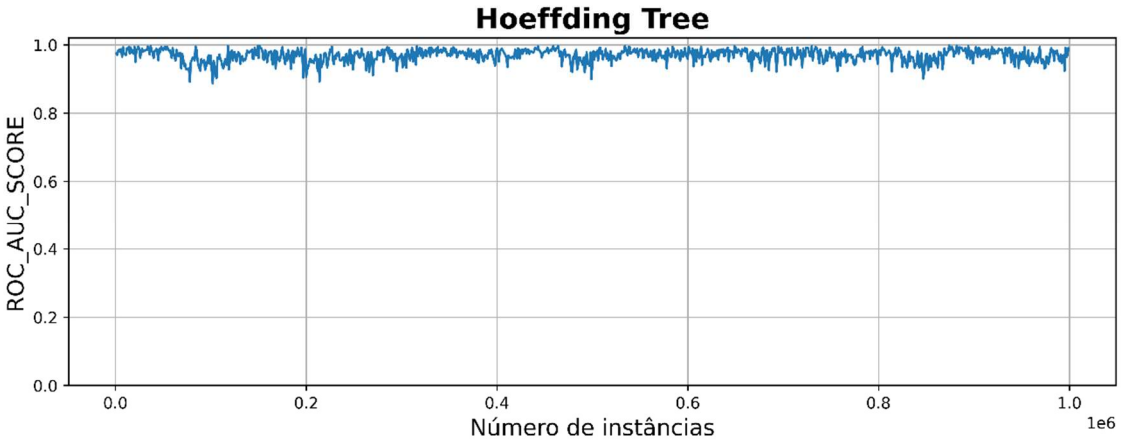

Figura 14 - Evolução da métrica roc_auc_score, mostrando os desvios de conceito do dataset Youchoose, para o learner HT. Fonte: autor.

Nas primeiras instâncias de cada *dataset* é possível observar que, como esperado, a performance dos modelos é alta, uma vez que o padrão desses dados já é conhecido. O modelo continua performando bem até o momento em que ocorrem os desvios de conceito nos dados. Para os *datasets* sintéticos, esse momento fica muito bem definido e pode-se notar claramente os três desvios de conceito que ocorrem nestes dados. Entretanto, para o *dataset* real, embora sejam observadas pequenas mudanças causadas pelas diferenças de características de cada *learner*, essas variações não estão bem definidas e o que se observa é apenas picos de variações na performance dos modelos.

5.4 Abordagem de testes

A fim de validar a aplicabilidade e a performance da estrutura proposta, foi elaborado um protocolo de testes que visa mostrar a capacidade de escalonamento horizontal do módulo de inferência, a capacidade de atualização incremental dos modelos pelo módulo de atualização, além da capacidade de serialização dos modelos atualizados em diversos valores de frequências.

Em suma, os testes foram divididos em quatro fases distintas, cada uma com um objetivo específico que buscam responder às questões mencionadas no início deste capítulo:

Primeira Fase: Esta fase busca estabelecer uma abordagem inicial que sirva como referência comparativa para as fases subsequentes. O foco é criar uma base de comparação para os testes posteriores levando em consideração um cenário ideal, onda a atualização ocorre de forma incremental, ou seja, a cada novo evento o modelo é atualizado e a próxima predição é feita já utilizando este modelo. É importante destacar que, nesta fase, os testes serão executados de forma centralizada em um ambiente controlado, e não dentro da estrutura proposta, visando gerar uma base de testes consistente e sem qualquer influência da arquitetura apresentada.

Segunda Fase: O objetivo principal desta fase é avaliar a capacidade da arquitetura de escalar horizontalmente os *endpoints* de inferência, com o propósito de atender a demandas elevadas de solicitações. A análise concentra-se em verificar a eficiência apenas do módulo de inferência em lidar com uma alta quantidade de requisições simultâneas.

Terceira Fase: Nesta fase, a meta é avaliar o impacto da frequência de disponibilização dos modelos atualizados no *throughput* e na performance preditiva. O foco está em entender como a regularidade das atualizações de modelos influencia a capacidade do sistema em atender as solicitações e em manter a performance preditiva dos modelos.

Quarta Fase: A fase final visa avaliar a capacidade preditiva do sistema com diversos níveis de paralelismo. A análise é realizada considerando diferentes quantidades de *endpoints* ativos no sistema, utilizando uma frequência de atualização definida com base nos resultados do teste anterior. Esta etapa busca entender se o sistema pode se adaptar eficazmente às mudanças nos padrões dos dados, mantendo uma alta qualidade preditiva.

Em resumo, cada fase foi projetada para abordar aspectos cruciais da performance e escalabilidade do sistema concentrando-se em avaliar o seu comportamento levando-se em consideração a performance preditiva dos modelos (precisão com que o modelo realiza as predições, inferidas através da métrica *roc_auc_score*), *throughput* do sistema (tempo que o sistema leva para processar todo o fluxo de dados) e fila *Kafka* (número de instâncias enviadas ao tópico *Kafka* que ainda não foram consumidas pelo módulo de atualização e versionamento).

Também é importante destacar que todos os testes realizados foram realizados sob limitações de *hardware* planejadas, com o objetivo de simular ambientes desafiadores para a estrutura. Dessa forma, é possível observar seu comportamento em situações críticas e identificar possíveis gargalos no *pipeline*.

5.5 Criação dos modelos e teste *baseline*

Utilizando os *learners* e *datasets* mencionados acima, foram criados nove modelos distintos que foram treinados utilizando todas as instâncias de seus respectivos *datasets*. Esses modelos serão inicialmente carregados pelos os *endpoints* e servem como base para criação das versões geradas através da atualização incremental. Para facilitar o reconhecimento de cada modelo, eles foram nomeados conforme seus *datasets* e *learners*. Estes nomes serão utilizados nos demais testes para diferenciação dos modelos.

A Tabela 2 apresenta os nomes e respectivos *dataset* e *learners*:

Tabela 2 - Nomeação dos modelos gerados levando em conta o dataset e o learner utilizados na criação dos modelos.

Nome do Modelo	**Dataset**	**Learner**
AGR_a_NB	AGR_a	Naive Bayes
AGR_a_ARF	AGR_a	Adaptive Random Forest
AGR_a_HT	AGR_a	Hoeffding Tree
AGR_g_NB	AGR_g	Naive Bayes
AGR_g_ARF	AGR_g	Adaptive Random Forest
AGR_g_HT	AGR_g	Hoeffding Tree
Youchoose_NB	Youchoose	Naive Bayes
Youchoose_ARF	Youchoose	Adaptive Random Forest
Youchoose_HT	Youchoose	Hoeffding Tree

Para melhor entendimento dos testes subsequentes, é importante ressaltar que os modelos gerados, devido às características próprias e configuração de cada *learner*, apresentam tamanhos distintos. A Figura 15 mostra os tamanhos dos modelos criados em *quilobytes.*

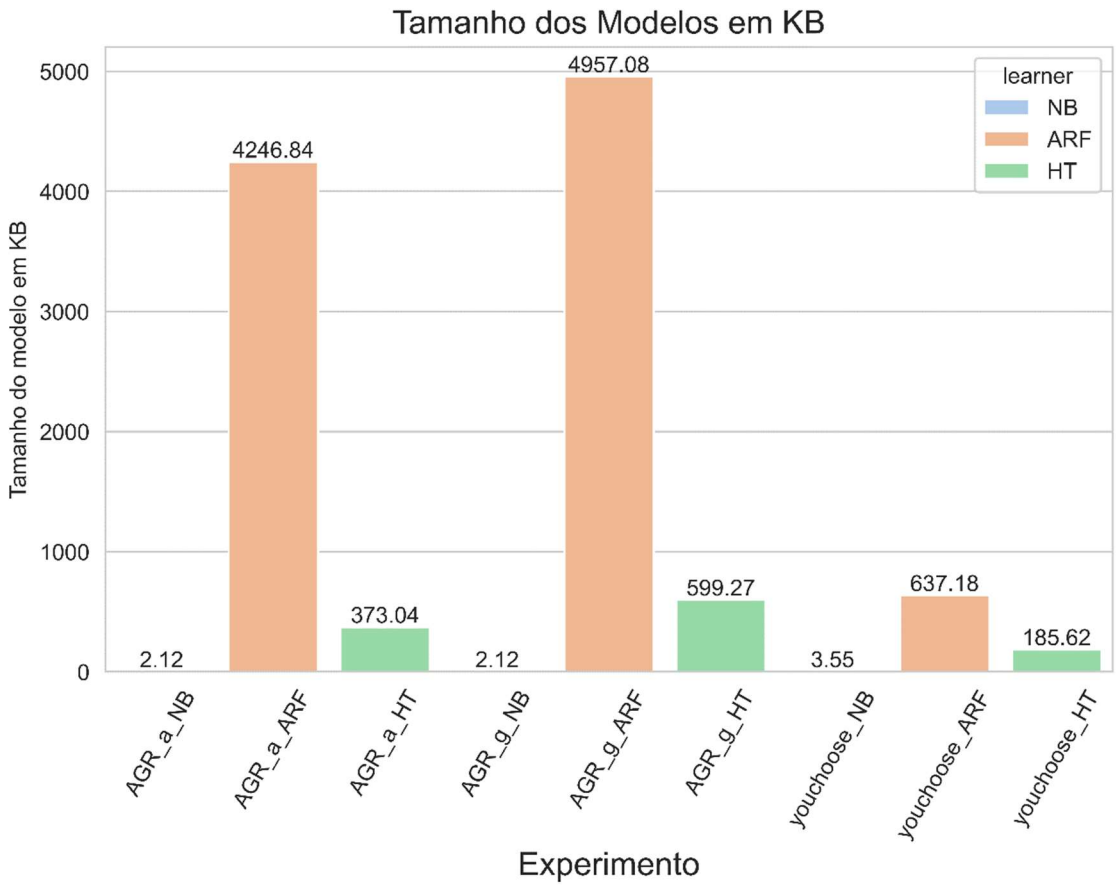

Figura 15 - Tamanho dos modelos em kB. Fonte: autor.

Analisando o gráfico acima, nota-se que os modelos baseados no *learner* NB são muito leves computacionalmente, com seu tamanho ficando em torno de pouco mais de 2 kB, na média. Em contrapartida, os modelos baseados no *learner* ARF são grandes, podendo chegar a tamanhos próximos a 5 MB. Já os modelos baseados no HT, possuem tamanhos relativamente pequenos, com seus tamanhos variando em torno de 400 kB na média.

Os tamanhos dos modelos têm grande relevância nos resultados dos testes a seguir, pois eles vão impactar diretamente na fila *Kafka*, *throughput* e performance preditiva do sistema.

O teste de *baseline*, ou linha de base, servirá como uma referência fundamental e ponto de comparação em relação a performance preditiva para os testes subsequentes. Ele foi realizado com o objetivo de demonstrar como o modelo se comporta às mudanças de conceito presentes nos *datasets*, quando recebe atualizações incrementais. Dessa forma, o teste oferece um parâmetro de validação para a arquitetura desenvolvida, permitindo uma análise do desempenho e da capacidade de adaptação do modelo em um cenário ideal.

Este teste consiste em processar integralmente o fluxo de dados de cada *dataset* utilizando todos os modelos, fora do ambiente do cluster *Kubernetes*. Assim como nos demais testes, a atualização do modelo é realizada de forma incremental. No entanto, neste caso específico, não há necessidade de disponibilizar o modelo atualizado para o módulo de inferência, visto que o teste é conduzido em uma única máquina, de maneira centralizada.

Embora esse método não permita o escalonamento horizontal do sistema, ele assegura que o modelo recém-atualizado possa ser utilizado na próxima inferência, possibilitando assim, a atualização em tempo real. Vale destacar que, em um ambiente de produção real, onde pode haver milhares de usuários fazendo requisições ao sistema, esse método centralizado de inferência torna-se impraticável.

A seguir são apresentados os resultados do teste de *baseline*. Os gráficos a seguir mostram a evolução da métrica *roc_auc_score* com as atualizações incrementais para todos os modelos desenvolvidos e *datasets* (linhas vermelhas). Para facilitar a compreensão dos resultados, os gráficos mostram também as curvas características de cada *dataset* sem atualizações (linhas tracejadas azuis). As Figuras 16, 17 e 18, apresentam os resultados dos três *learners*: *Naive Bayes*, *Adaptive Random Forest* e *Hoeffding Tree*, respectivamente, ao processarem os dados do *dataset* AGR_a com atualizações incrementais em cada interação.

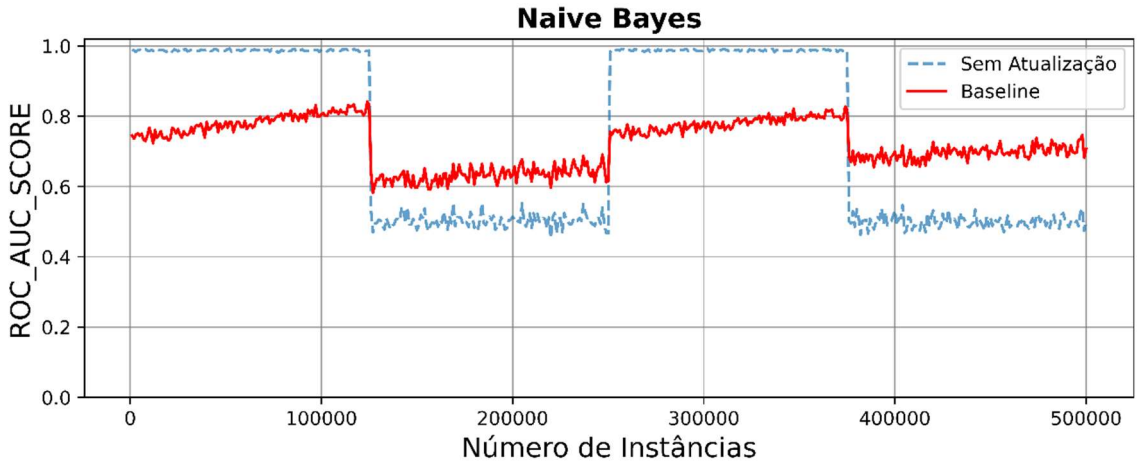

Figura 16 - Resultado do teste de baseline para o dataset AGR_a e learner NB. Fonte: autor.

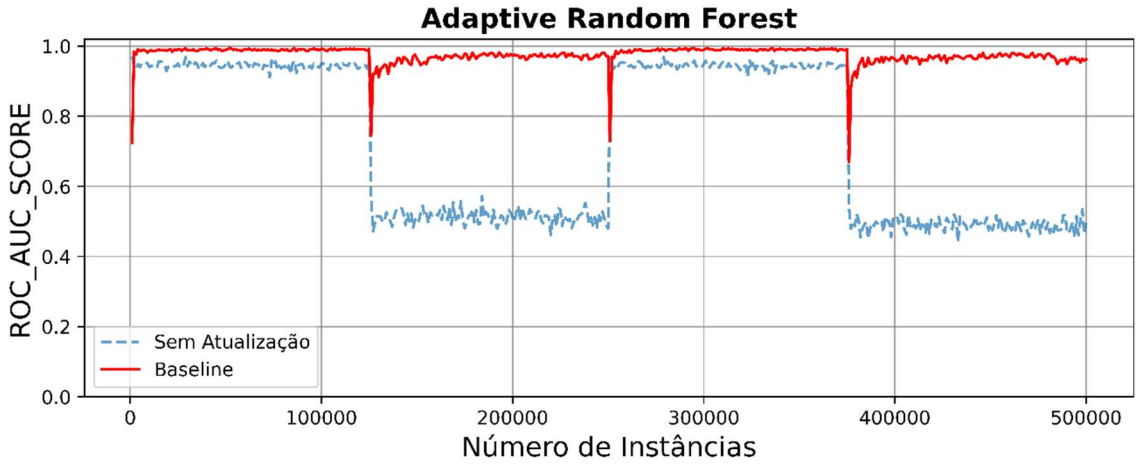

Figura 17 - Resultado do teste de baseline para o dataset AGR_a e learner ARF. Fonte: autor.

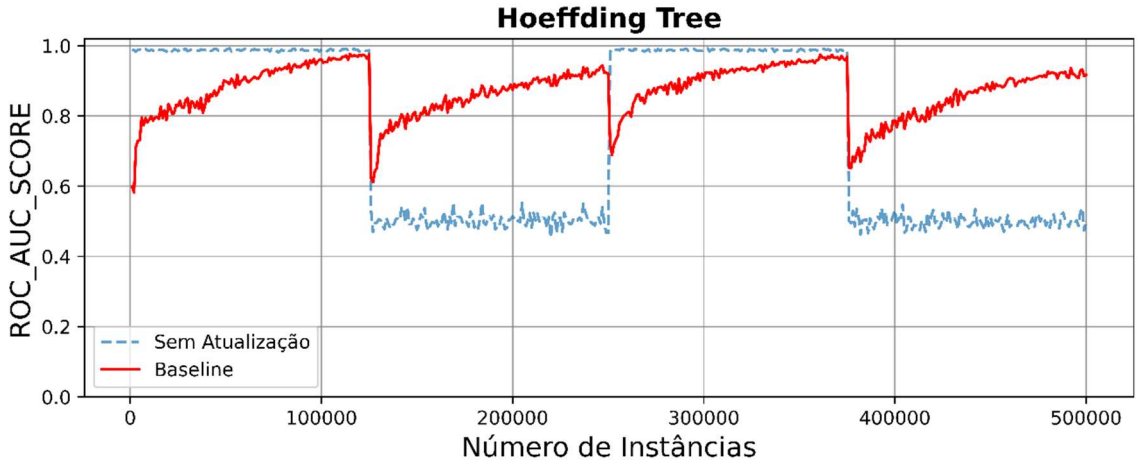

Figura 18 - Resultado do teste de baseline para o dataset AGR_a e learner HT. Fonte: autor.

Já Figura 19, Figura 20 e Figura 21, apresentam os resultados dos três *learners*: *Naive Bayes*, *Adaptive Random Forest* e *Hoeffding Tree*, respectivamente, ao processarem os dados do *dataset* AGR_g com atualizações incrementais em cada interação.

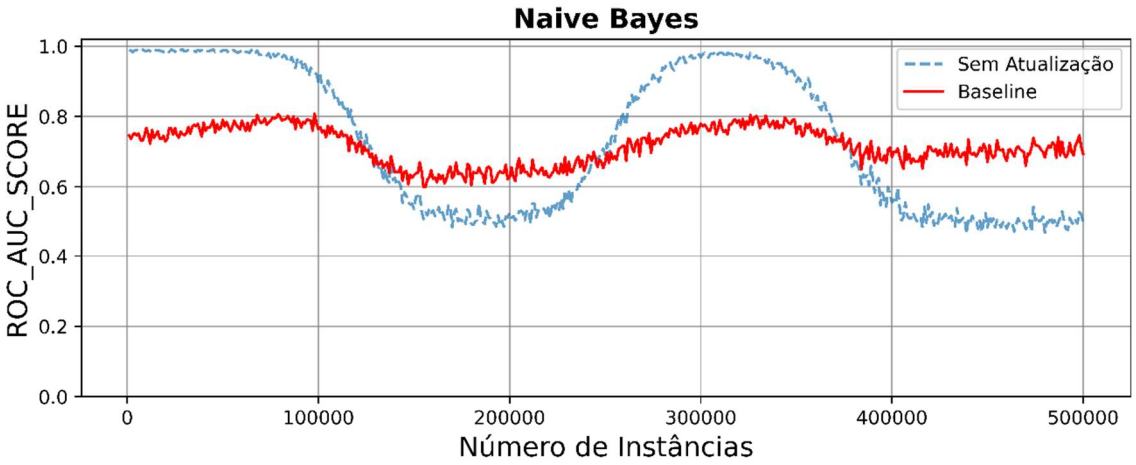

Figura 19 - Resultado do teste de baseline para o dataset AGR_g e learner NB. Fonte: autor.

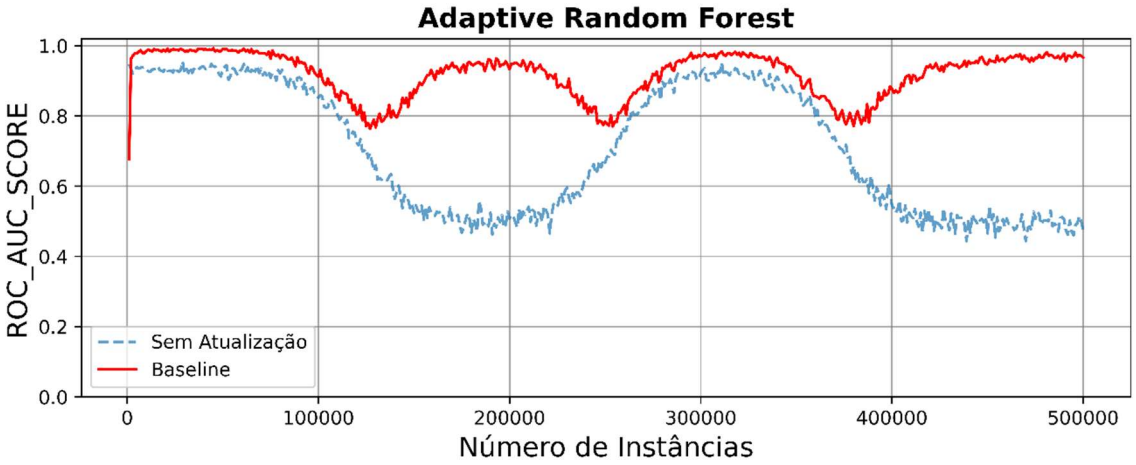

Figura 20 - Resultado do teste de baseline para o dataset AGR_g e learner ARF. Fonte: autor.

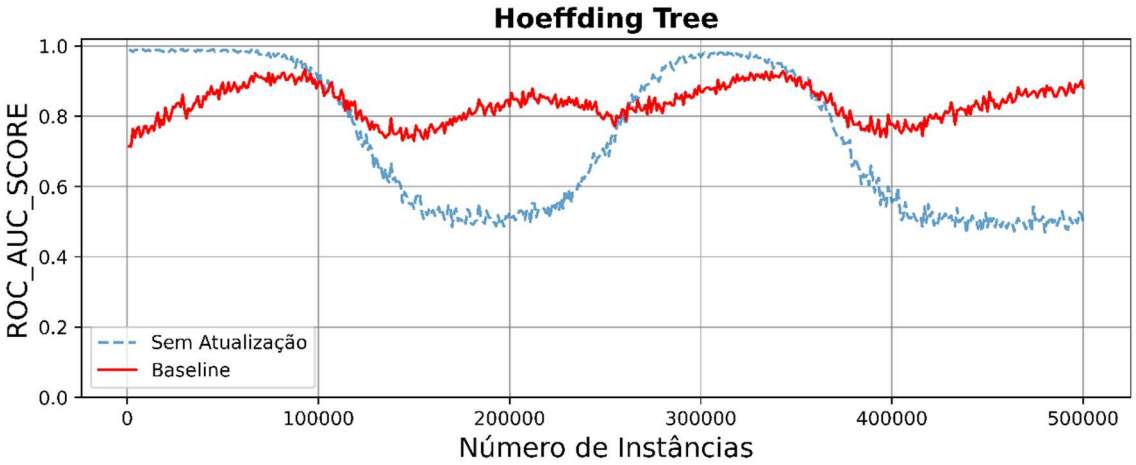

Figura 21 - Resultado do teste de baseline para o dataset AGR_g e learner HT. Fonte: autor.

E por fim, a Figura 22, Figura 23 e Figura 24, apresentam os resultados dos três *learners*: *Naive Bayes*, *Adaptive Random Forest* e *Hoeffding Tree*, respectivamente, ao processarem os dados do *dataset Youchoose* com atualizações incrementais em cada interação.

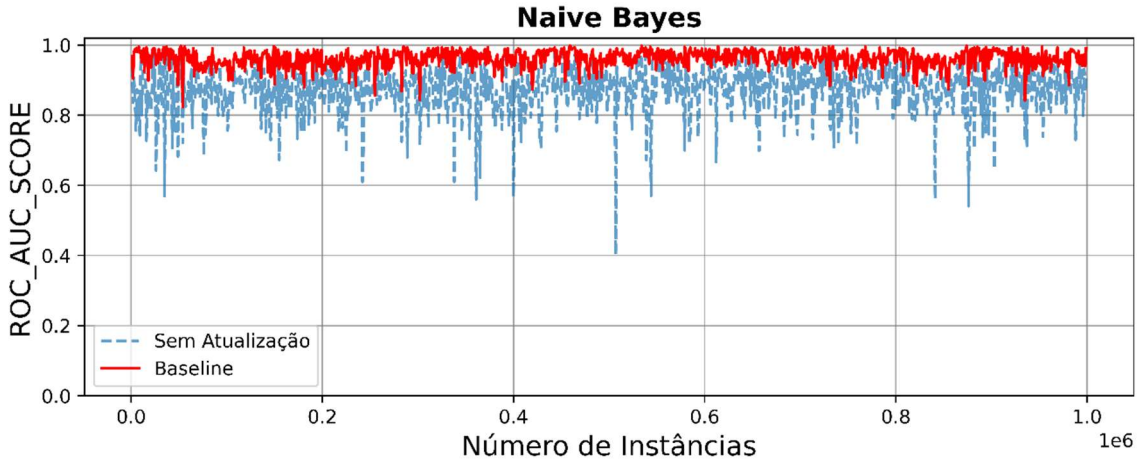

Figura 22 - Resultado do teste de baseline para o dataset Youchoose e learner NB. Fonte: autor.

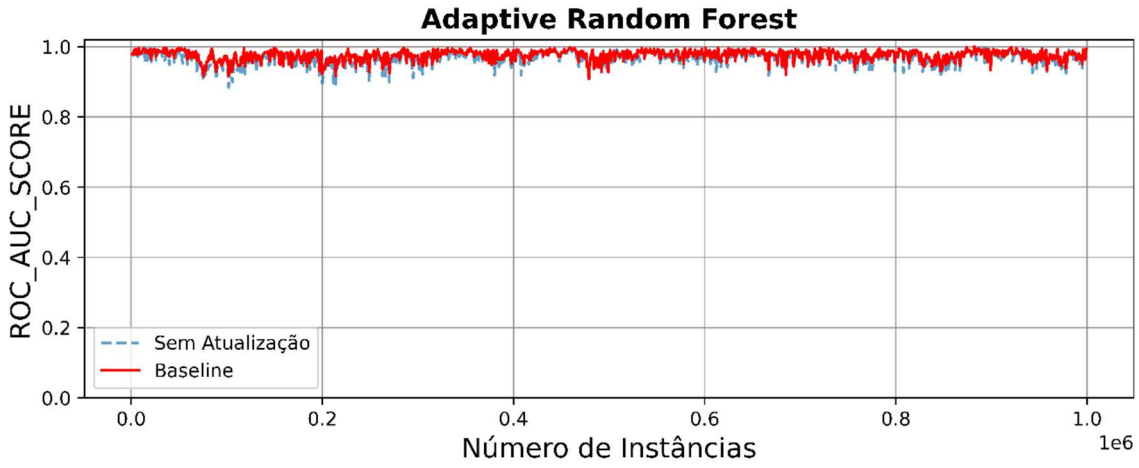

Figura 23 - Resultado do teste de baseline para o dataset Youchoose e learner ARF. Fonte: autor.

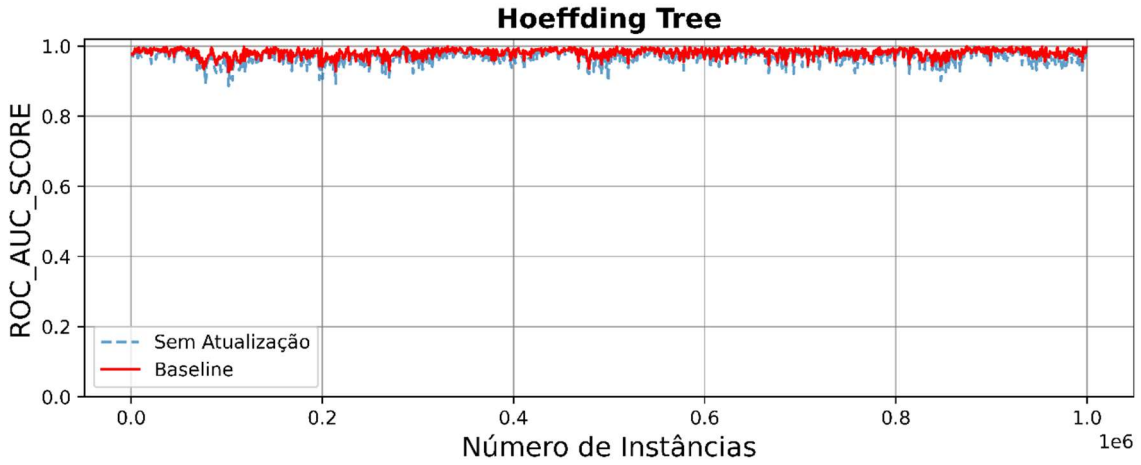

Figura 24 - Resultado do teste de baseline para o dataset Youchoose e learner HT. Fonte: autor.

Os gráficos acima mostram que, com a atualização incremental (na qual o modelo é ajustado a cada nova ocorrência e a próxima predição já utiliza o modelo atualizado), a adaptação dos modelos baseados nos *learners* ARF e HT aos desvios de conceito de cada *dataset* ocorre de forma rápida. No entanto, devido à sua alta elasticidade, os modelos baseados no *learner* NB tiveram sua performance um tanto comprometida, apresentando um achatamento na curva da métrica, que variou em torno de um valor médio.

Assim, embora existam pequenas variações na métrica *roc_auc_score* entre os diferentes *learners* para o mesmo *dataset*, devido às características distintas de cada um, a performance preditiva se manteve satisfatória ao longo de todo o fluxo de dados para todos os testes.

5.6 Teste do escalonamento do módulo de inferência

Este teste visa mostrar a capacidade do módulo de inferência em paralelizar os *endpoints* conforme a demanda de solicitações dos usuários. Teoricamente, o consumo de instância por segundo deve aumentar linearmente conforme aumenta-se o número de *endpoints* ativos no sistema. Ou seja, se o consumo de instância por segundo com um *endpoint* ativo sistema for igual a X (100%), espera-se que com dois *endpoints*, o consumo

seja X*2 (200%), já com 3 *endpoints* o consumo será X*3 (300%) e assim por diante. Porém, é preciso considerar o *overhead* do sistema, ou seja, os custos computacionais indiretos dos processos.

Para execução destes testes, os *endpoints* do módulo de inferência tiveram seus *hardwares* limitados a ¼ de CPU por container sem limitações de memória. Além disso, os *datasets* utilizados para geração dos fluxos tiveram seu tamanho reduzido para 150 mil instâncias, igualando assim o número de instâncias utilizadas no teste. Outro aspecto importante é que, como já mencionado anteriormente, os modelos utilizados na inferência foram criados utilizando todas as instâncias dos *datasets* originais, proporcionando assim, modelos com tamanhos condizentes com os encontrados em ambientes reais.

A dinâmica do teste consiste em consumir um fluxo de dados com solicitações de inferência, utilizando diferentes números de *endpoints* ativos no sistema (paralelismo horizontal), abrangendo todos os *learners* e *datasets*. Neste teste, as *features* de cada instância dos *datasets* são enviadas apenas ao módulo de inferência, sem passar pelo módulo de atualização e versionamento.

Inicialmente, mede-se o consumo de instâncias por segundo para cada *learner* e *dataset*, com apenas um *endpoint* ativo no sistema. Em seguida, aumenta-se o número de *endpoints* ativos para dois e repete-se a medição do consumo de instância por segundo para todos os *learners* e *datasets*. Esse procedimento é repetido até que o número de *endpoints* ativos alcance oito.

Com este teste, buscou-se investigar o comportamento do sistema em relação ao escalonamento horizontal do número de *endpoints* de inferência. A expectativa inicial era que o consumo de instâncias aumentasse linearmente com o aumento do número de *endpoints*. Entretanto, os resultados demonstraram uma discrepância entre o consumo ideal esperado e o consumo real observado.

Vale ressaltar que para esse teste, foi considerado apenas o consumo de instâncias por segundo, não considerando a performance preditiva dos modelos, uma vez que não houve atualizações dos modelos.

A Figura 25 mostra o resultado dos testes de escalabilidade do módulo de inferência. A linha azul mostra o valor médio do consumo de instância por segundo real (com *overhead*), levando em consideração todos os *learners* e *dataset*. Enquanto a linha vermelha mostra qual seria o consumo de instância por segundo ideal (sem *overhead*), levando em consideração todos os *learners* e *dataset*.

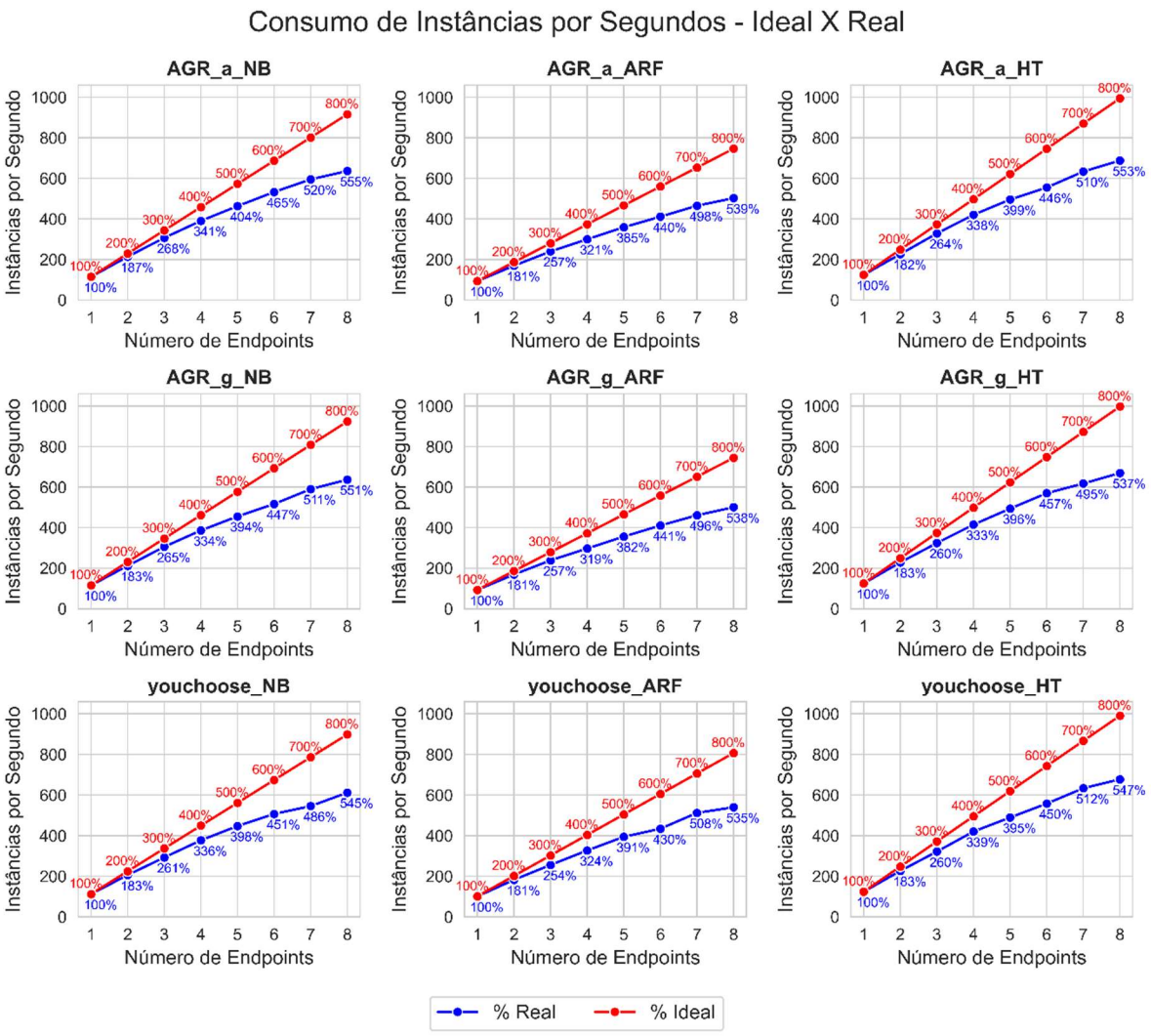

Figura 25 - Resultado dos testes de escalabilidade do módulo de inferência. Fonte: autor.

No gráfico acima pode-se notar que a linha vermelha, que representa o cenário ideal hipotético, cresce linearmente. Entretanto, devido ao *overhead* do sistema, não é isso o que acontece na prática. Em todos os testes, a linha azul, que representa o cenário real, começa a se afastar da linha vermelha conforme aumenta-se o número de *endpoints* ativos no sistema, tendendo a uma curva logarítmica (salvo pequenas variações).

Destaca-se também, a diferença no consumo de instâncias por segundo para cada *learner* utilizado. Embora a curva do consumo real tenha características semelhantes às dos demais, os modelos baseados no *learner* ARF apresentam um consumo consideravelmente menor. Isso ocorre devido ao tamanho destes modelos, como mostra a Figura 15, o que os torna computacionalmente mais pesados.

Sendo assim, embora seja claro que o escalonamento do módulo de inferência resulte em um aumento significativo no consumo de instâncias por segundo, esse aumento não é

linear como se esperava, devido ao *overhead* intrínseco à arquitetura. Entretanto, o paralelismo do sistema se mostra bastante eficiente, principalmente quando o número de *endpoints* ativos não atinge valores muito altos (valores entre 2 e 4 *endpoints*), onde nota-se que o valor real não fica tão distante do ideal.

5.7 Teste da frequência de disponibilização dos modelos

A terceira fase dos testes tem como objetivo avaliar o comportamento da arquitetura proposta comparando diversos valores do parâmetro frequência de disponibilização dos modelos. Esse parâmetro define a periodicidade com que uma versão do modelo, que está sendo atualizada pelo módulo de atualização e versionamento, será serializada no repositório de modelos e consequentemente, carregado para o módulo de inferência. Valores mais altos para este parâmetro implicam em disponibilizações de modelos em intervalos de tempo mais espaçados, por outro lado, valores menores, implicam em modelos disponibilizados em intervalos menores de tempo.

É importante destacar, mais uma vez, que na proposta desenvolvida, diferentemente do que se encontra na literatura, o módulo de inferência tem a capacidade de carregar o modelo atualizado para a memória dos *endpoints* sem precisar recriá-lo novamente. Além disso, os *endpoints* de inferência e de atualização e versionamento carregam e disponibilizam os modelos em processos paralelos, conferindo a estes, um menor custo computacional na substituição dos modelos. Entretanto, mesmo com essa vantagem computacional, a disponibilização do modelo em períodos muito curtos de tempo, pode impactar significativamente no sistema, resultando em perda de performance preditiva e aumento do *throughput*.

Para a realização deste experimento, os contêineres de inferência tiveram seus recursos computacionais limitados a ½ CPU por contêiner, sem restrições de memória RAM. Já o contêiner responsável pela atualização e versionamento foi limitado a 2 CPUs e 12 GB de memória RAM.

Com base em testes anteriores, foi definida uma frequência de disponibilização inicial de 2000 e final de 1000 eventos. Para cada frequência de disponibilização, o teste

foi executado com diferentes números de *endpoints* ativos no sistema, sendo definidos quatro valores: 2, 4, 6 e 8 *endpoints*.

A dinâmica do teste envolveu variações sucessivas na frequência de disponibilização. Após completar o teste com a frequência inicial e todos os números de *endpoints*, a frequência de disponibilização foi reduzida em 250 eventos, passando de 2000 para 1750 eventos. O teste foi então repetido para todos os números de *endpoints* ativos novamente. Esse processo continuou sendo repetido, com reduções sucessivas de 250 eventos na frequência de disponibilização, até que a frequência alcançasse 1000 eventos. Dessa forma, cada combinação de frequência de disponibilização e número de *endpoints* foi testada para avaliar como esses diferentes valores de frequência afetam a arquitetura proposta.

A seguir serão apresentados os resultados do teste de frequência de disponibilização dos modelos, onde serão apresentadas as performances preditivas dos modelos, o comportamento da fila *Kafka* e o *throughput* do sistema, para cada valor do parâmetro. Neste experimento, o *throughput* será analisado em duas partes: o consumo de instâncias por segundo e o número de modelos atualizados disponibilizados por segundo. Este último, é o resultado da multiplicação do tempo total para consumir todo o fluxo, pela frequência de disponibilização dos modelos, dividido pelo número total de instâncias do *dataset* em questão.

É importante destacar que, embora o número de *endpoints* utilizados neste teste também tenha impacto nos resultados, o foco da avaliação está apenas na frequência de disponibilização dos modelos.

Esta análise será iniciada pelo comportamento da fila *Kafka* em relação às diferentes frequências de disponibilização. Isso porque, o número de instâncias na fila *Kafka*, esperando para serem consumidas pelo módulo de atualização e versionamento, tem relação direta com a performance preditiva e *throughput* do sistema. Sendo assim, é essencial entender o seu comportamento em cada valor dos parâmetros, para então entender como ela impacta nos demais resultados.

Explicando de forma resumida, a formação das filas no tópico *Kafka* ocorre quando há um acúmulo de instâncias aguardando para serem processadas pelo módulo de atualização e versionamento. Esse acúmulo se deve ao fato de que o módulo de inferência processa as requisições mais rapidamente do que o módulo de atualização e versionamento consegue consumi-las. Como as instâncias rotuladas são enviadas ao tópico *Kafka* logo após a predição, o volume de dados pode exceder a capacidade de processamento do módulo de atualização e versionamento quando há muitos *endpoints* ativos na inferência

ou quando o parâmetro de frequência de serialização está configurado em valores baixos, resultando assim na formação das filas.

A Figura 26 mostra a média de instâncias na fila do *Kafka* para todos os modelos, conforme varia-se a frequência de disponibilização e número de *endpoints* ativos no sistema.

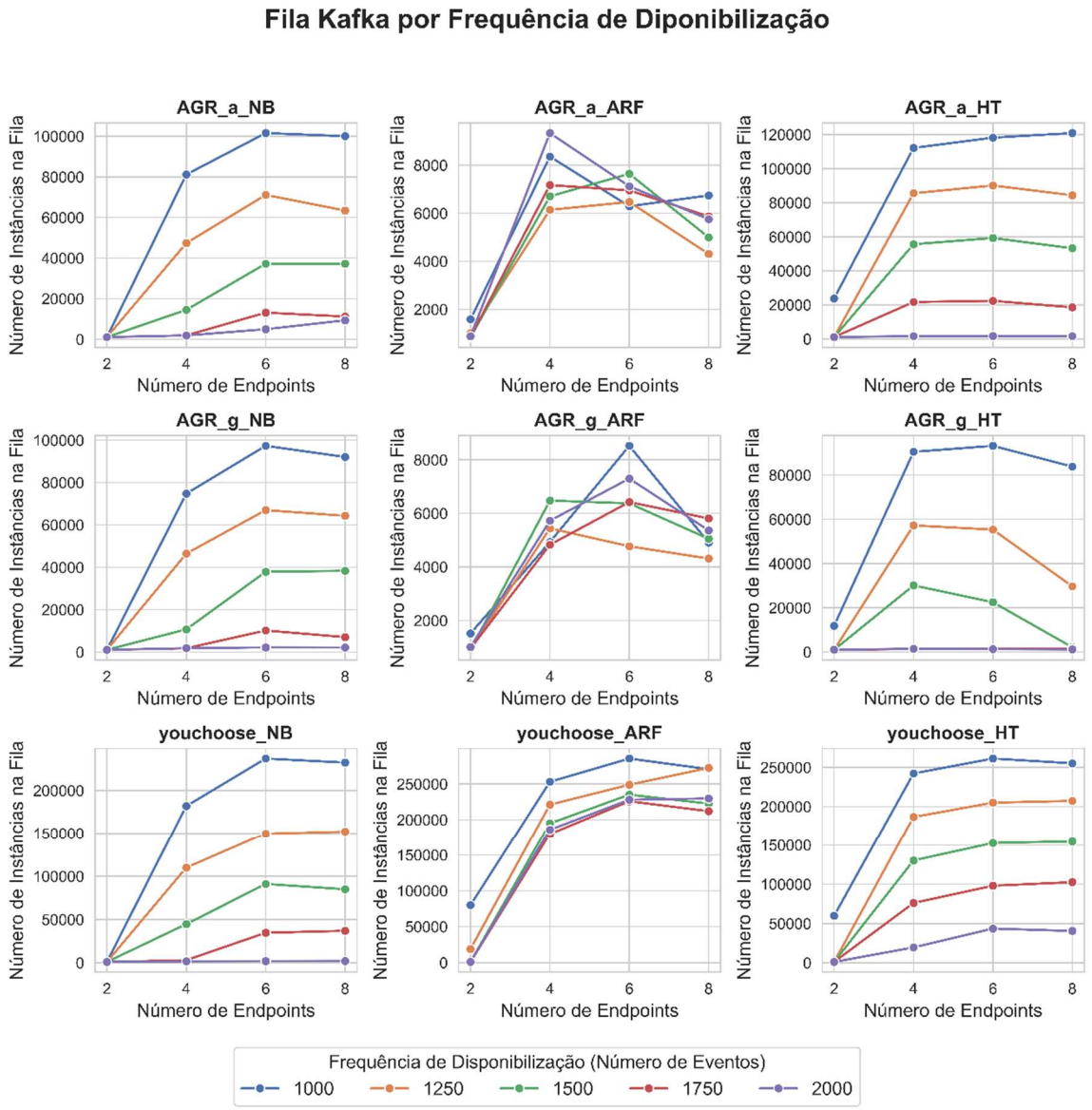

Figura 26 - Média de instâncias na fila do Kafka por frequência de disponibilização dos modelos e número de endpoints ativos no sistema. Fonte: autor.

Nos gráficos apresentados na Figura 26, observa-se o aumento significativo de instâncias na fila *Kafka* conforme variam os parâmetros de frequência de disponibilização e o número de *endpoints* ativos no sistema. No entanto, quando o número de *endpoints* atinge valores mais altos, geralmente acima de seis, ocorre uma estabilização ou até uma

diminuição na fila *Kafka* para determinadas frequências. Esse comportamento pode ser atribuído ao aumento do *overhead* gerado com escalonamento horizontal do sistema de inferência, que também pôde ser observado nos demais testes.

Analisando os resultados mais detalhadamente, é possível observar que para os modelos baseados em NB e HT, valores maiores de frequências de disponibilização (2000 eventos ou mais próximo disto) tem menor impacto na fila *Kafka.* Onde observa-se em alguns casos, que a curva permanece próximo de zero, mesmo com valores elevados de *endpoints* ativos. Então, conforme diminui-se o valor do parâmetro (tendendo a 1000 eventos), nota-se um aumento consistente no número de instâncias na fila. Podendo em alguns casos, ultrapassar o valor de 250 mil instâncias na fila esperando para serem consumidas.

Entretanto, nos modelos baseados no *learner* ARF, essas grandes mudanças não são observadas. Isso ocorre porque os modelos gerados com o *learner* ARF são computacionalmente pesados. Consequentemente, o consumo de instâncias por segundo na inferência é mais lento, o que resulta em um menor número de instâncias enviadas ao módulo de atualização e, assim, em filas menores.

Assim, fica evidente que para modelos que não sofrem com o tamanho excessivo, com os casos dos modelos baseados nos *learners* NB e HT, a frequência de disponibilização tem um grande impacto na fila *Kafka*. Para valores muito baixos deste parâmetro, a quantidade de instâncias na fila pode chegar a ¼ do tamanho do *dataset*. Como por exemplo no caso no modelo Youchoose_HT, onde foi utilizada uma amostra de 1 milhão de instâncias e a fila *Kafka* chegou a ter 250 mil instâncias esperando para serem consumidas.

Em seguida, será analisado o impacto da frequência de disponibilização no *throughput* do sistema. Para isso, foi medido o número de instâncias consumidas por segundo e o número de modelos disponibilizados por segundo, para cada caso. A Figura 27 apresenta o consumo de instâncias por segundo dos modelos (considerando o processamento de todo o *pipeline*) para todos os valores de frequências de disponibilização e número de *endpoints* ativos.

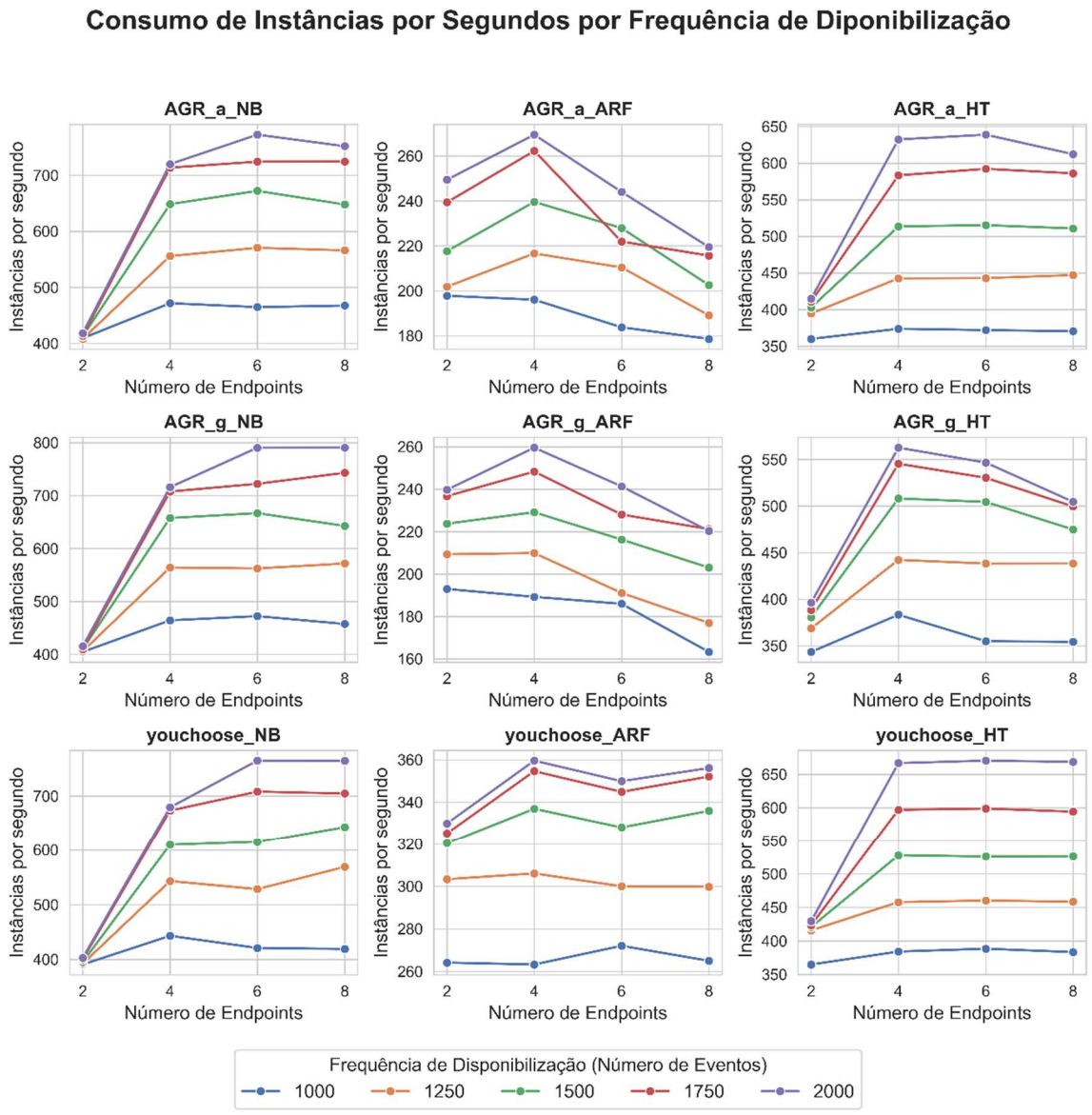

Figura 27 - Consumo de instâncias por segundo por frequências de disponibilização dos modelos e número de endpoints ativos. Fonte: autor.

Analisando os gráficos acima, nota-se que valores mais altos do parâmetro de frequência de disponibilização (2000 eventos ou próximo disto) apresentam um maior consumo de instâncias por segundo em comparação com valores mais baixos para esse parâmetro (1000 eventos ou próximo disto). Isso se deve ao fato de que, para valores menores do parâmetro, o sistema gasta mais recurso computacional nos processos de serialização e carregamentos dos modelos. Por consequência, os módulos levam mais tempo para consumir os fluxos de dados, impactando assim no consumo de instância por segundo do sistema.

Também se observa, assim como nos resultados da fila do *Kafka*, uma estabilização no consumo de instâncias para valores de *endpoints* ativos maiores que seis. Como já mencionado, esse fenômeno ocorre devido ao aumento do *overhead* gerado com escalonamento horizontal do sistema de inferência, uma vez que, mais *endpoints* exigem maior coordenação e comunicação, o que pode afetar o desempenho e levar à estabilização ou diminuição no consumo das instâncias.

Nos modelos baseados nos *learners* NB e HT, o consumo passa de 650 instâncias por segundo (chegando perto de 800 instâncias por segundo para o caso do modelo AGR_a_NB), quando o parâmetro está configurado com 2000 eventos e o número de *endpoints* ativos é igual ou maior a quatro. Conforme diminui-se o valor do parâmetro frequência de disponibilização (tendendo a 1000 eventos), observa-se uma diminuição gradual do consumo, onde o consumo máximo observado fica em torno de 480 instâncias por segundo. Entretanto, para os modelos baseados no *learner* ARF, o consumo de instâncias por segundo, de modo geral, é relativamente menor. Isso ocorre porque estes modelos são computacionalmente mais pesados, como já explicado anteriormente.

Sendo assim, observa-se que a frequência de disponibilização de modelos tem grande influência no *throughput* do sistema. Onde, para o parâmetro configurado em 2000 eventos, especialmente com mais de 4 *endpoints* ativos, o sistema consumiu quase o dobro de instâncias por segundo em comparação à configuração com 1000 eventos.

Em seguida, a Figura 28 apresenta o número de modelos atualizados disponibilizados por segundo, para cada valor de frequência de disponibilização e número de *endpoints* selecionados para este teste.

Um ponto importante a destacar, é que a variação no parâmetro de frequência de disponibilização dos modelos pode não impactar diretamente o número de modelos disponibilizados por segundo. Isso ocorre devido ao *overhead* do sistema, que em ambientes com processamentos de fila de modo distribuídos pode ser significativo. Assim, as mudanças neste parâmetro nem sempre se traduzem em um aumento no número de modelos disponibilizados por segundo.

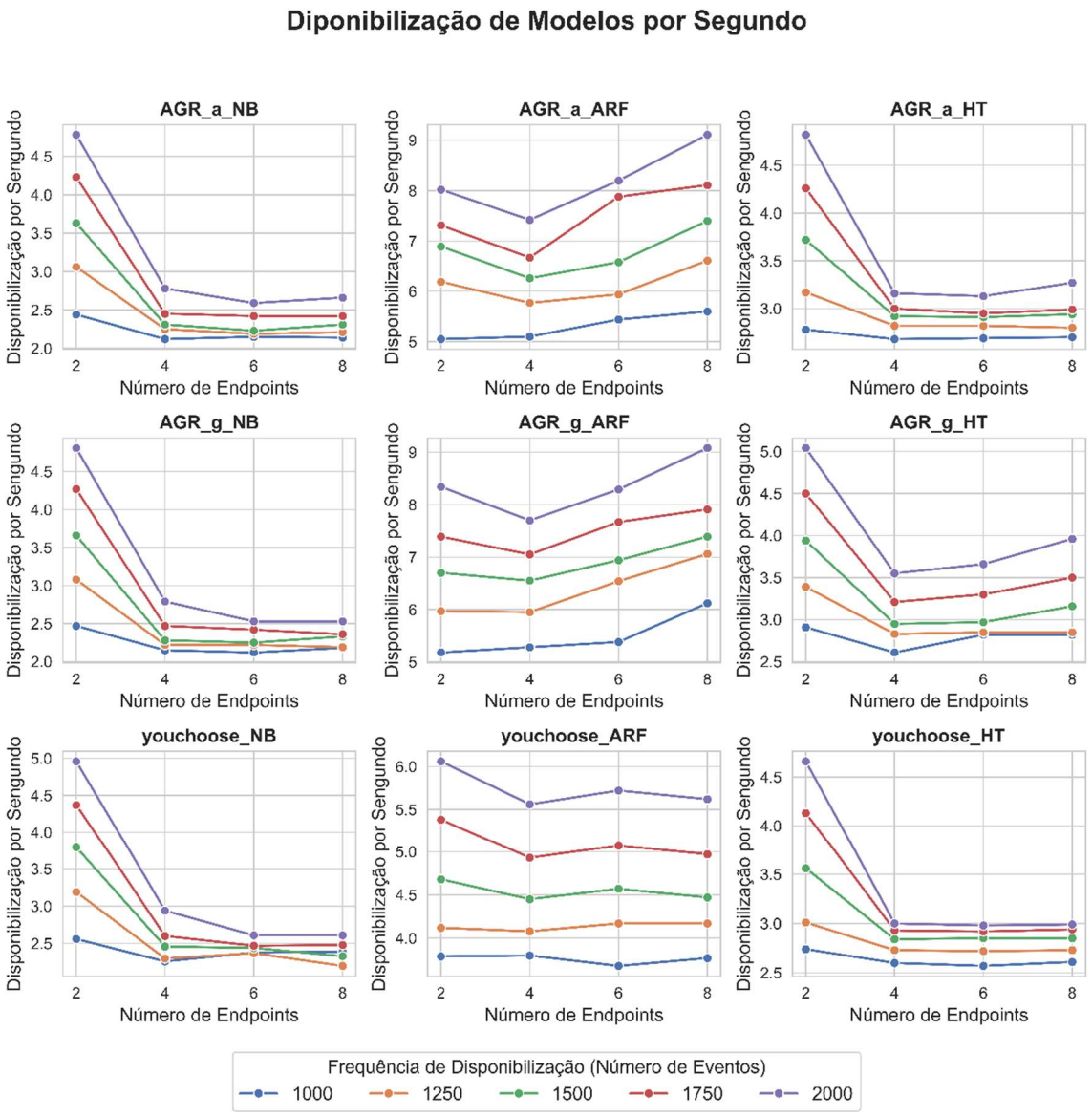

Figura 28 - Número de modelos atualizados disponibilizados por segundo por frequência de disponibilização dos modelos e número de endpoints ativos. Fonte: autor.

Uma análise mais detalhada dos gráficos revela que os modelos baseados no *learner* ARF são mais impactados pela frequência de disponibilização, pois o *overhead* do sistema é atenuado pelo tamanho dos modelos gerados com esses *learners*.

Em contraste, os modelos baseados nos *learners* NB e HT, que são computacionalmente mais leves, mostram uma menor sensibilidade à variação da frequência de disponibilização. Para esses modelos, especialmente em cenários com um maior número de *endpoints* ativos, o número de modelos disponibilizados por segundo se mantém relativamente constante, independentemente do valor do parâmetro de frequência, devido ao *overhead* do sistema.

De modo geral, mesmo nos cenários em que a frequência de disponibilização foi configurada para valores mais altos (2000 eventos) e o número de *endpoints* ativos foi mantido baixo (2 *endpoints*), o que teoricamente aumentaria os intervalos de disponibilização, o sistema ainda conseguiu disponibilizar modelos atualizados em menos de 9 segundos, até mesmo para os *learners* mais pesados.

Já para os *learners* mais leves, em cenários favoráveis (frequência de disponibilização próxima de 1000 eventos e número de *endpoints* acima de 4), o tempo de disponibilização de um modelo atualizado ficou frequentemente abaixo de 2,5 segundos.

Após entender o comportamento da fila do *Kafka* e o *throughput* do sistema em relação às variações no parâmetro frequência de disponibilização, é necessário analisar seu impacto na performance preditiva dos modelos.

Devido à grande quantidade de experimentos realizados e à complexidade dos resultados, optou-se por utilizar nesta análise apenas os resultados obtidos com os modelos gerados com o *dataset* AGR_a, pois estes, apresentaram uma melhor visualização do impacto da frequência de disponibilização na performance preditiva do modelo devido às características de seus desvios de conceito.

Sendo assim, inicialmente serão apresentados os gráficos que comparam os resultados das performances preditivas dos modelos com a performance preditiva do *baseline* para os dois extremos dos valores de frequência previamente definidos (2000 e 1000 eventos). Em seguida, serão exibidos os gráficos comparativos da métrica *roc_auc_score* em relação ao *baseline*, juntamente com a variação da fila *Kafka*, para alguns cenários específicos, a fim de mostrar melhor a relação entre esses dois fatores.

Os gráficos apresentados na Figura 29 comparam a evolução da métrica *roc_auc_score* para os modelos baseados nos *learners* NB, ARF e HT, utilizando o *dataset* AGR_a e uma frequência de disponibilização de 2000 eventos, em relação à métrica medida no teste *baseline*.

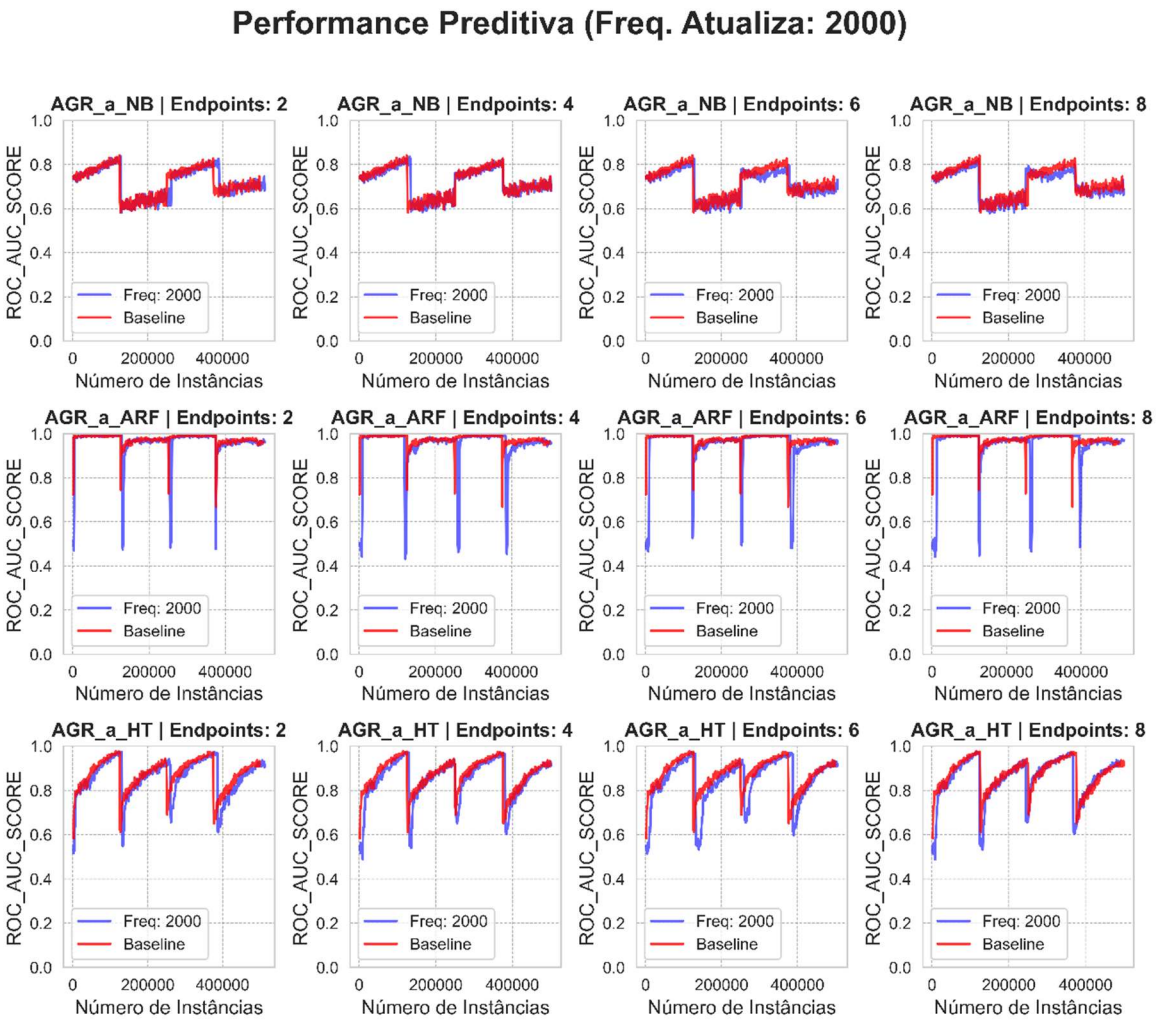

Figura 29 - Evolução da métrica roc_auc_score para os learners NB, ARF e HT e dataset AGR_a, com frequência de disponibilização de 2000 eventos. Fonte: autor.

Já gráficos apresentados na Figura 30 comparam a evolução da métrica *roc_auc_score* para os modelos baseados nos *learners* NB, ARF e HT, utilizando o *dataset* AGR_a e uma frequência de disponibilização de 1000 eventos, em relação à métrica medida no teste *baseline*.

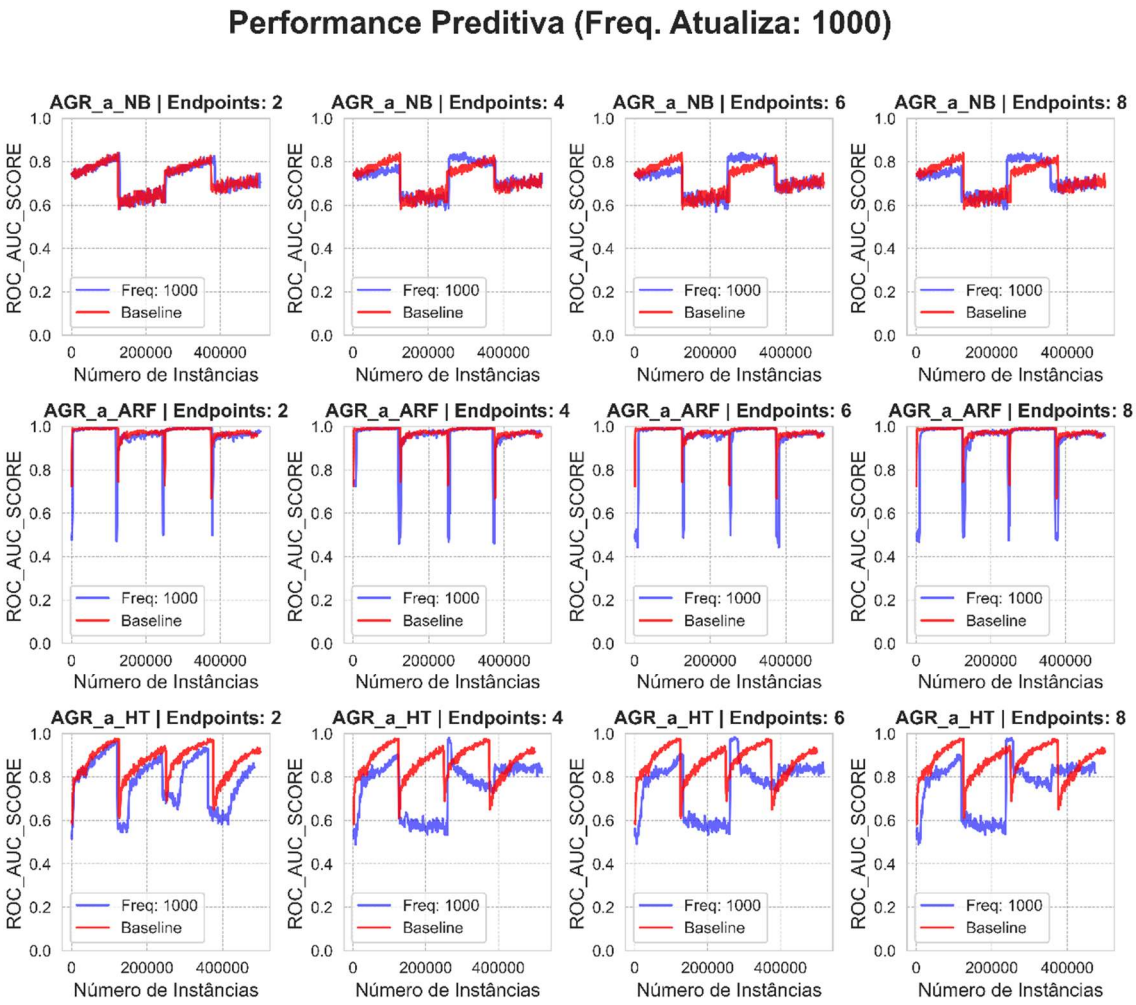

Figura 30 - Evolução da métrica roc_auc_score para os learners NB, ARF e HT e dataset AGR_a, com frequência de disponibilização de 1000 eventos. Fonte: autor.

Como já mencionado, devido às particularidades de cada *learner*, os modelos reagem de forma distinta às atualizações realizadas. Por isso, as análises dos resultados serão feitas levando em consideração essas diferentes características.

Os modelos baseados no *learner* NB possuem grande elasticidade, conforme explicado na seção 5.2 deste capítulo. Isso significa que esses modelos têm a capacidade de reter conceitos antigos, ajustando-se minimamente para incorporar novos dados sem grandes mudanças estruturais. Para um melhor entendimento de como as características dos *learners* NB prevalecem em relação às configurações dos parâmetros, a Figura 31 mostra em mais detalhe a variação da métrica *roc_auc_score* em relação à métrica *baseline* e ao número de instâncias na fila *Kafka*, no pior cenário para esses experimentos (frequência = 1000 e número de *endpoints* = 8).

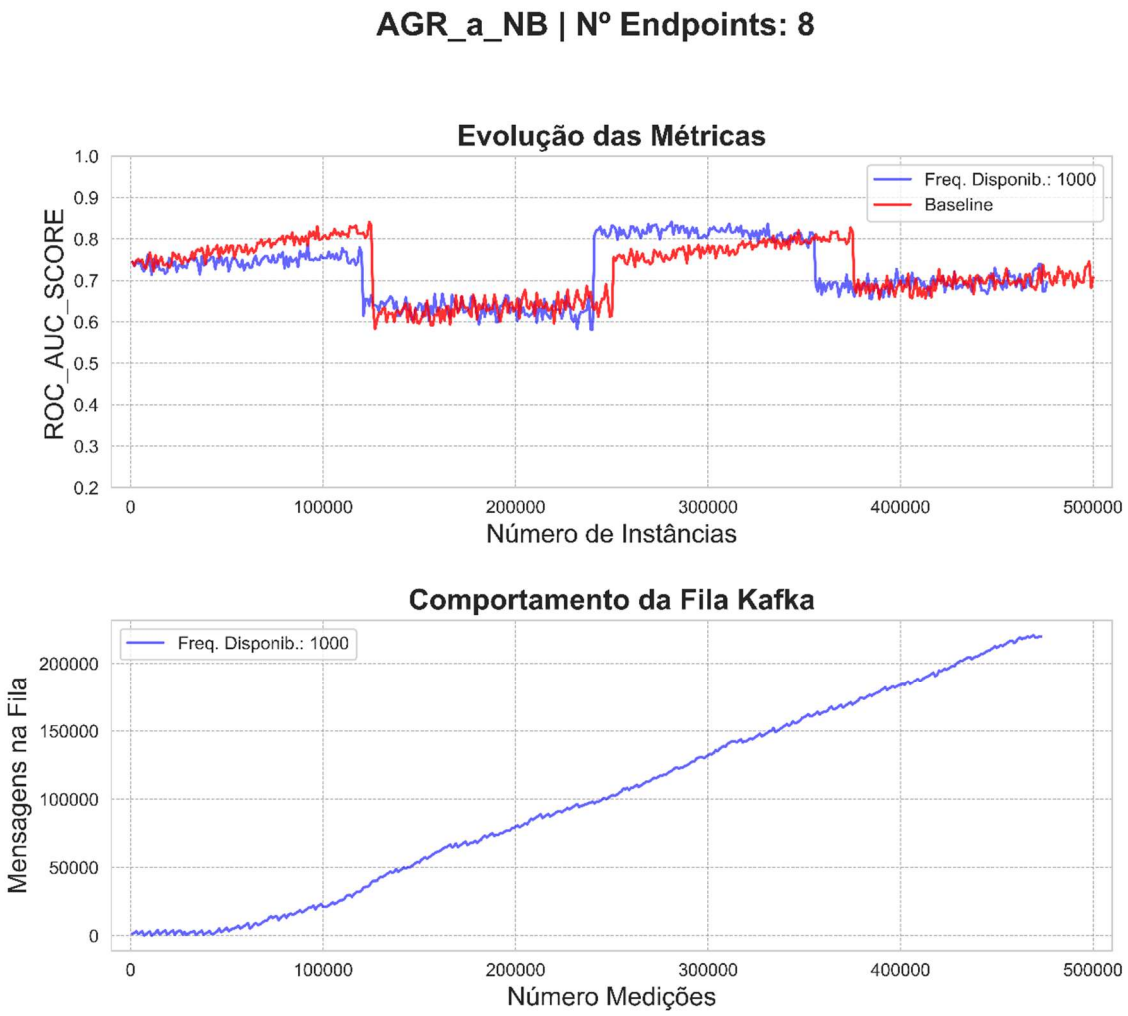

Figura 31 - Métrica roc_auc_score em relação à métrica baseline e o número de instâncias na fila Kafka para o modelo AGR_a_NB, com 8 endpoints ativos e frequência de disponibilização em 1000 eventos. Fonte: autor.

Na Figura 31 observa-se que, mesmo com um valor de frequência de disponibilização menor e o número máximo de *endpoints* ativos, o que incide em uma grande quantidade de instâncias na fila *Kafka*, a variação da métrica *roc_auc_score* permanece muito pequena. Assim, é possível concluir que, independentemente das frequências de disponibilização ou do tamanho das filas *Kafka*, devido às características do *learner* NB, a adaptação do modelo não apresenta grandes variações, mantendo-se próxima ao valor do *baseline*.

Por outro lado, os modelos baseados no *learner* ARF possuem alta plasticidade, o que significa que eles se adaptam rapidamente aos dados atuais, podendo esquecer ou não priorizar conceitos antigos. Isso facilita sua adaptação a mudanças de conceito no *dataset*. No entanto, a performance observada nesses experimentos não se deve apenas a essa característica. O grande tamanho dos modelos ARF gerados neste trabalho impacta diretamente o *throughput* de inferência, tornando o processo mais lento e reduzindo o fluxo

de instâncias rotuladas publicadas no tópico *Kafka*. Como resultado, o tamanho da fila *Kafka* diminui, permitindo que esses modelos sejam atualizados com instâncias mais recentes, o que os torna mais adequados para lidar com desvios de conceito nos dados.

Esse fenômeno pode ser mais bem observado na Figura 32, que mostra o resultado do experimento no pior cenário (frequência = 1000 e número de *endpoints* = 8).

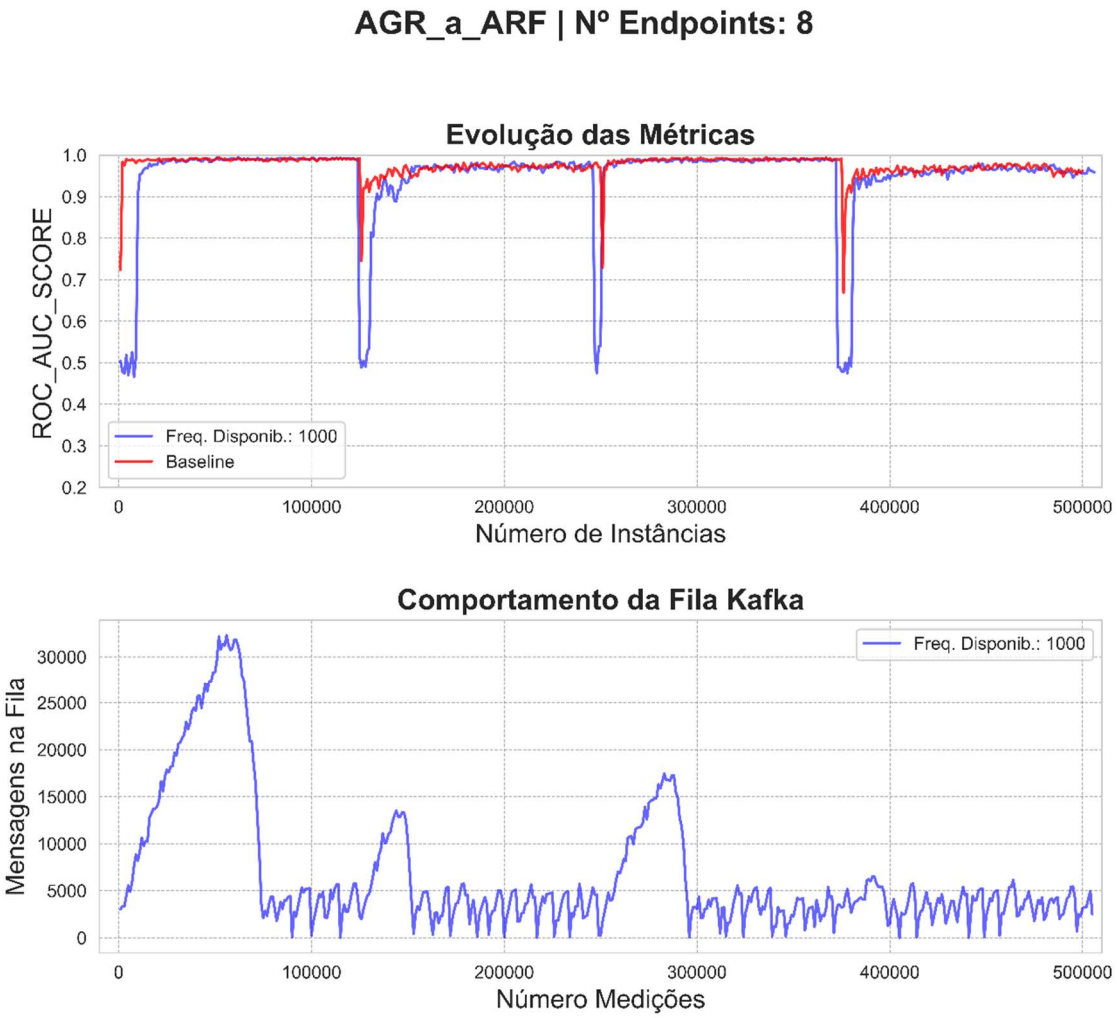

Figura 32 - Métrica roc_auc_score em relação à métrica baseline e o número de instâncias na fila Kafka para o modelo AGR_a_ARF, com 8 endpoints ativos e frequência de disponibilização em 1000 eventos. Fonte: autor.

Os resultados mostram que, mesmo no pior cenário, não há grandes variações na fila *Kafka*, apenas alguns picos de aumento são observados quando ocorrem desvios de conceito no *dataset*. Esses picos resultam do mecanismo de subdivisão dos nós da árvore do *learner*, que, ao ser acionado, consome mais recursos computacionais e, consequentemente, impacta a capacidade de consumo de instâncias pelo módulo de atualização e versionamento. No entanto, quando o conceito dos dados se estabiliza, o módulo retoma sua capacidade normal de consumo de instâncias por segundo, diminuindo

rapidamente a fila *Kafka* e mantendo a performance preditiva do modelo próxima à do *baseline*.

Por fim, o modelo baseado no *learner* HT, que apresenta uma relação intermediária entre elasticidade e plasticidade, foi o mais impactado pelas variações na frequência de disponibilização dos modelos. Neste caso específico, vale demonstrar de forma mais detalhada, o impacto da frequência de disponibilização na fila *Kafka* e na performance preditiva do modelo, em dois cenários. Primeiro com a frequência em 2000 eventos e em seguida com 1000 eventos, ambos os cenários com número de *endpoints* igual a 8.

Sendo assim, a Figura 33 mostra de modo mais detalhado o resultado do experimento para o cenário onde a frequência de disponibilização dos modelos é igual a 2000 eventos.

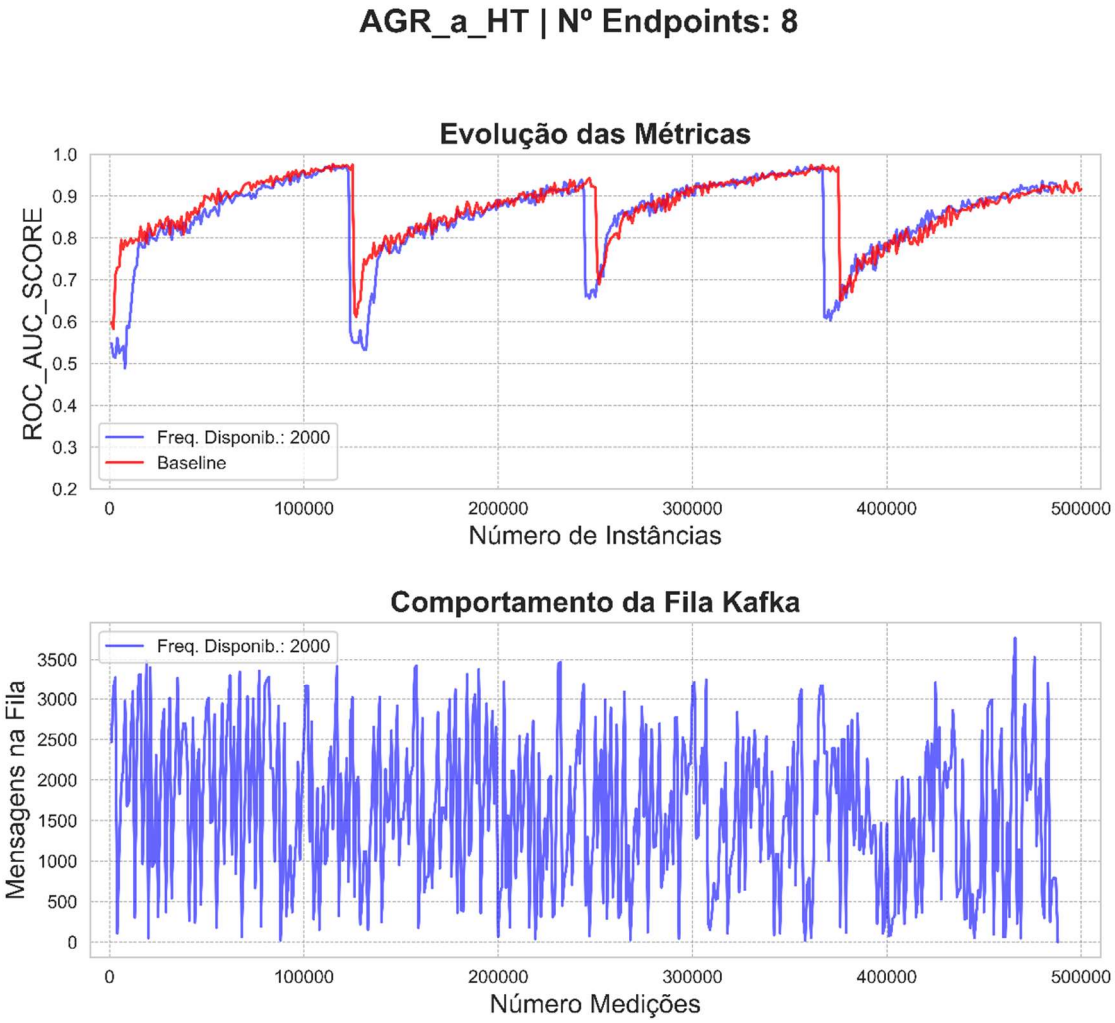

Figura 33 - Métrica roc_auc_score em relação à métrica baseline e o número de instâncias na fila Kafka para o modelo AGR_a_HT, com 8 endpoints ativos e frequência de disponibilização em 2000 eventos. Fonte: autor.

Já a Figura 34 mostra de modo mais detalhado o resultado do experimento para o cenário onde a frequência de disponibilização dos modelos é igual a 1000 eventos.

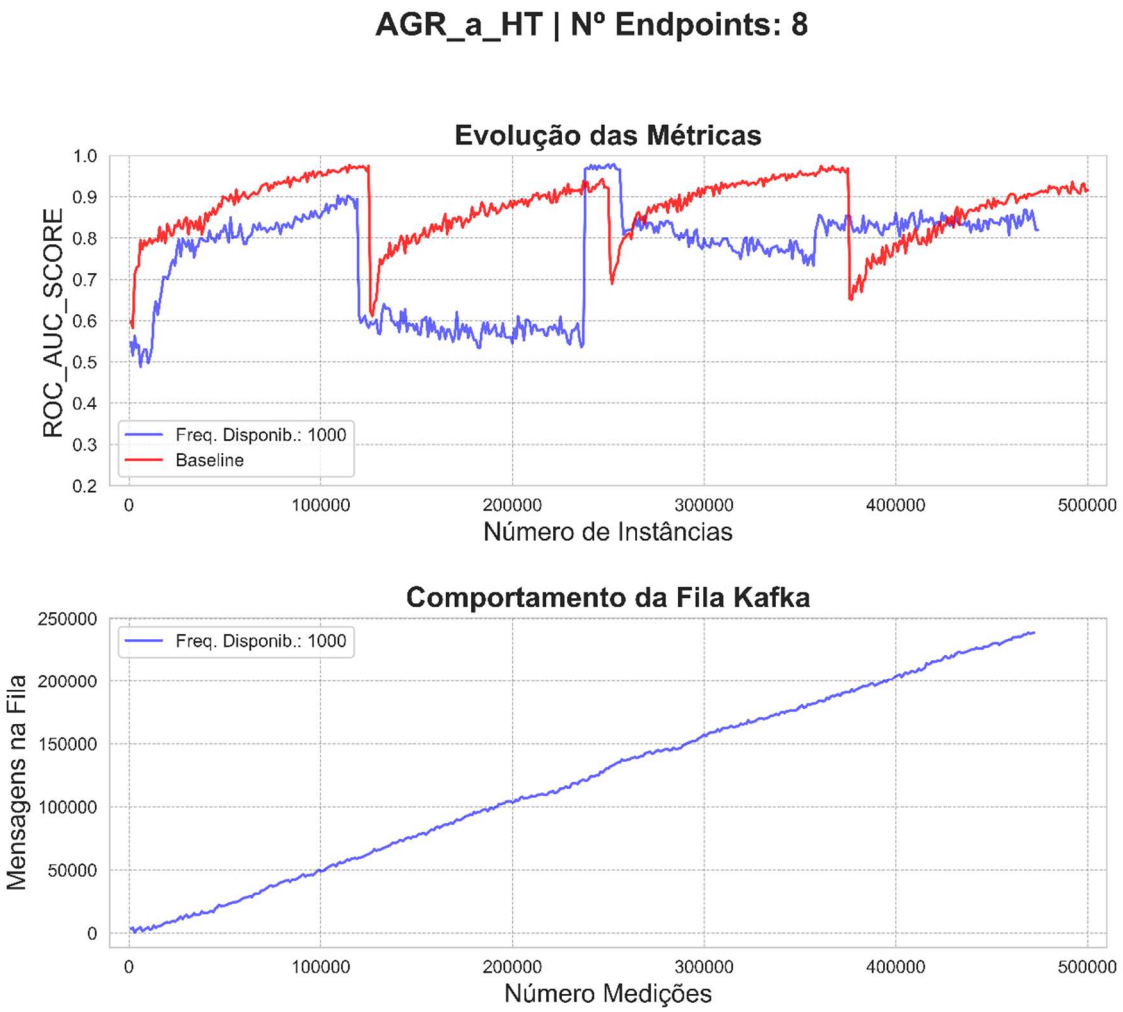

Figura 34 - Métrica roc_auc_score em relação à métrica baseline e o número de instâncias na fila Kafka para o modelo AGR_a_HT, com 8 endpoints ativos e frequência de disponibilização em 1000 eventos. Fonte: autor.

Comparando as figuras acima, pode-se notar que com o parâmetro da frequência configurado em 2000 eventos, a variação no tamanho da fila *Kafka* é relativamente pequena, não ultrapassando 4000 instâncias esperando para serem consumidas. Com isso, a performance preditiva se manteve muito próxima do *baseline*. No entanto, com o parâmetro configurado em 1000 eventos, observa-se um grande aumento da fila *Kafka* e por consequência, uma perda significativa na capacidade preditiva dos modelos. Assim, conclui-se que, para os modelos baseados no *learner* HT, a frequência de disponibilização dos modelos exerce um grande impacto na performance preditiva do sistema.

Os resultados ressaltam a importância de otimizar o parâmetro de frequência de disponibilização conforme as necessidades do cenário, buscando equilibrar *throughput* e performance preditiva. Essa otimização visa evitar o acúmulo excessivo de instâncias na fila *Kafka*, garantindo que os modelos permaneçam atualizados e reflitam com precisão os

dados atuais, sem comprometer a capacidade de atender às solicitações de inferência dos usuários.

5.8 Teste performance preditiva e *throughput* do sistema em relação ao número de *endpoints*

Por fim, o último teste busca avaliar o impacto do número de *endpoints* ativos sobre o sistema. De modo mais detalhado, o experimento monitorou comportamento do *throughput*, a fila *Kafka* e a performance preditiva do sistema, considerando todo o pipeline da arquitetura (Inferência, Atualização e Versionamento do Modelo), para diferentes valores do parâmetro que define o número de *endpoints* ativos no sistema.

Nestes experimentos, os contêineres de inferência tiveram seus recursos computacionais limitados a ½ CPU por contêiner, sem restrições de memória RAM. Em contrapartida, o contêiner responsável pela atualização e versionamento foi limitado a 2 CPUs e 12 GB de memória RAM.

A quantidade de *endpoints* ativos na inferência variou de um a oito. Com isso, a alocação de recursos computacionais começou com o módulo de inferência utilizando ¼ de CPU em comparação com o módulo de atualização e versionamento quando havia apenas um *endpoint* ativo. À medida que o número de *endpoints* aumentou para oito, a CPU alocada para a inferência passou a ser o dobro da alocada para o módulo de atualização e versionamento.

Quanto à frequência de disponibilização dos modelos, esta foi fixada em 1500 eventos. Esse valor foi determinado com base nos resultados dos testes anteriores (Frequência de Disponibilização dos Modelos) e corresponde ao ponto intermediário, onde o sistema não está completamente saturado e nem trabalhando com folga. Assim, embora o sistema responda adequadamente em termos de performance preditiva com poucos *endpoints* ativos na inferência, observa-se um aumento da fila no *Kafka* e, consequentemente, uma degradação na métrica *roc_auc_score* à medida que o número de *endpoints* aumenta.

A análise será iniciada pelo impacto no *throughput* do sistema, que será avaliado para todos os modelos e *datasets*. Entretanto, devido à complexidade dos resultados e ao elevado

número de experimentos, a análise da fila *Kafka* e da performance preditiva será restrita ao *learner* HT e ao *dataset* AGR_a.

Sendo assim, o gráfico apresentado na Figura 35 mostra o consumo de instâncias por segundo para todos os modelos e *datasets* conforme varia-se o número de *endpoints* ativos, considerando o processamento de todo o *pipeline*.

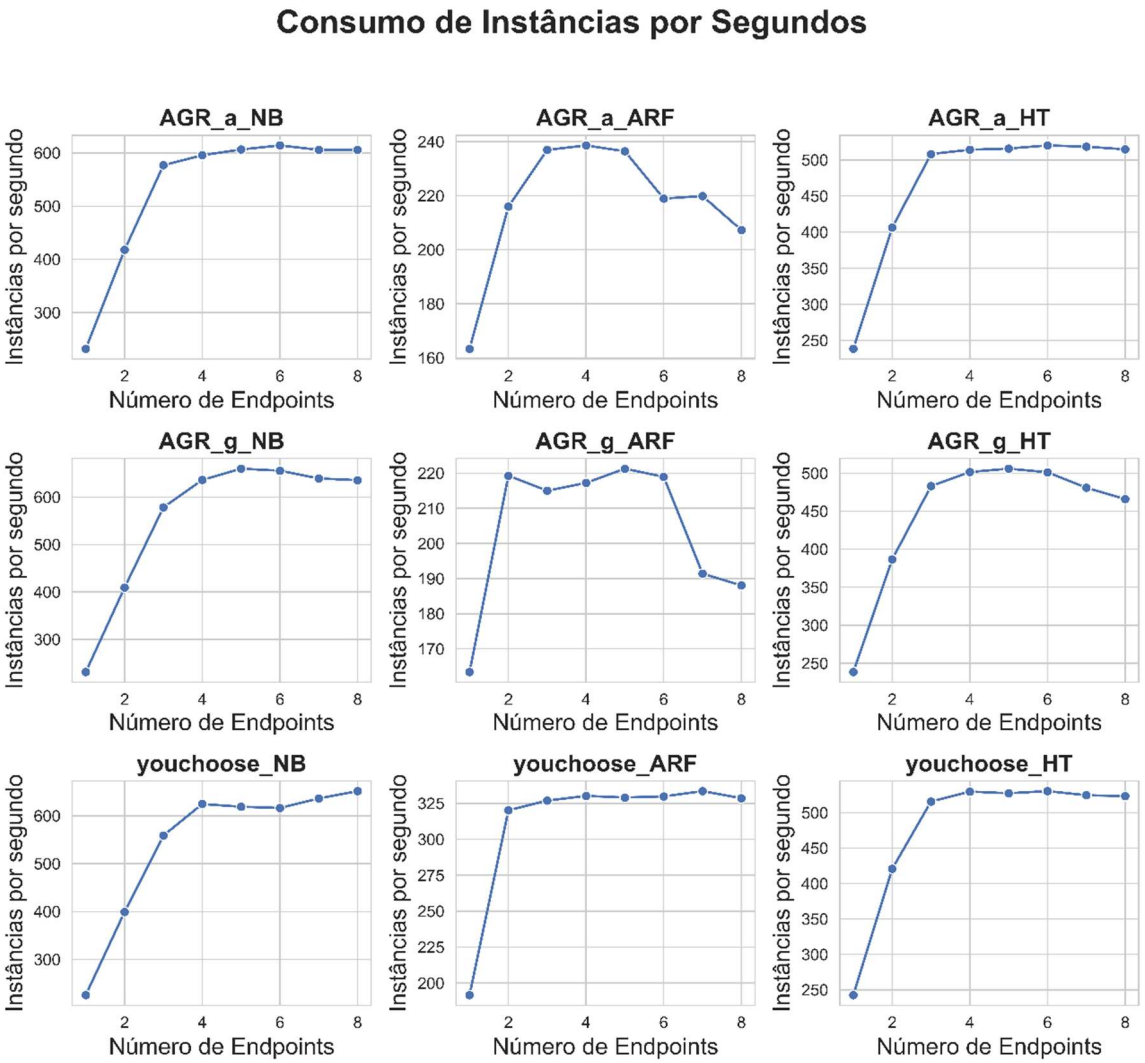

Figura 35 - Consumo de instâncias por segundo para todos os modelos e datasets, para cada valor do parâmetro número de endpoints. Fonte: autor.

O gráfico acima mostra um crescimento significativo no número de instâncias consumidas pelo sistema ao se aumentar o número de *endpoints* de um para dois e de dois para três, para praticamente todos os modelos (com exceção apenas para o modelo AGR_g_ARF). Porém, quando se aumenta o número de *endpoints* de três para quatro, observa-se apenas um pequeno incremento no consumo de instâncias. Já para valores maiores que quatro *endpoints* ativos, o que se observa é uma estabilização ou mesmo uma

diminuição no consumo das instâncias por parte de alguns modelos. Assim como já explicado anteriormente, esse fenômeno ocorre devido ao aumento do *overhead* gerado com escalonamento horizontal do sistema de inferência.

Conclui-se, portanto, que o escalonamento horizontal dos *endpoints* impacta positivamente no consumo de instâncias por segundo na arquitetura proposta. No entanto, para números mais altos de *endpoints* ativos, esse impacto diminui, levando a uma estabilização no consumo de instâncias por segundo e, em alguns casos, até mesmo a uma redução desse consumo.

Os gráficos a seguir ilustram o comportamento da fila *Kafka* e a evolução da métrica *roc_auc_score* do modelo AGR_a_HT, para diferentes números de *endpoints* ativos no sistema e com uma frequência de disponibilização fixada em 1500 eventos. Para facilitar a visualização, os resultados são apresentados em oito gráficos, uma para cada número de *endpoints* ativos, além de um gráfico que mostra o comportamento da fila *Kafka* para todos os gráficos anteriores.

Assim, a Figura 36, exibe os resultados dos testes para todos os valores de *endpoints*, juntamente com o comportamento da fila *Kafka*. A sequência de gráficos mostra a evolução da métrica *roc_auc_score* e o aumento da fila *Kafka* conforme o número de *endpoints* ativos no sistema aumenta. Nos gráficos que representam os resultados dos testes para um e dois *endpoints* ativos (gráficos *endpoints*=1 e *endpoints*=2), observa-se que a variação da fila *Kafka* é pequena, variando entre zero e duas mil instâncias esperando para serem consumidas. Como consequência, há uma boa performance preditiva do modelo para ambos os gráficos, com uma recuperação imediata da performance preditiva após a queda abrupta da métrica causada pelo desvio de conceito nos dados. Nestes casos, a curva da métrica *roc_auc_score* se assemelha à observada no teste de *baseline*.

À medida que o número de *endpoints* aumenta (gráfico: *endpoints*=3), há um aumento significativo no número de instâncias na fila *Kafka* e uma pequena degradação na performance preditiva do modelo. Essa diminuição de performance pode ser mais bem observada quando se analisa os intervalos que surgem entre a queda abrupta da métrica, causada pelos desvios de conceito nos dados, e o início da recuperação da performance do modelo. Nota-se também que esses intervalos se ampliam e afasta-se da métrica do *baseline*, à medida que o número de *endpoints* aumenta, até o ponto em que a curva da métrica perde suas características iniciais (*endpoints*=4 até a *endpoints*=8), onde não é mais possível visualizar as quedas e recuperações na métrica *roc_auc_score*. Para estes casos, observa-se um grande salto no número de instâncias na fila *Kafka*.

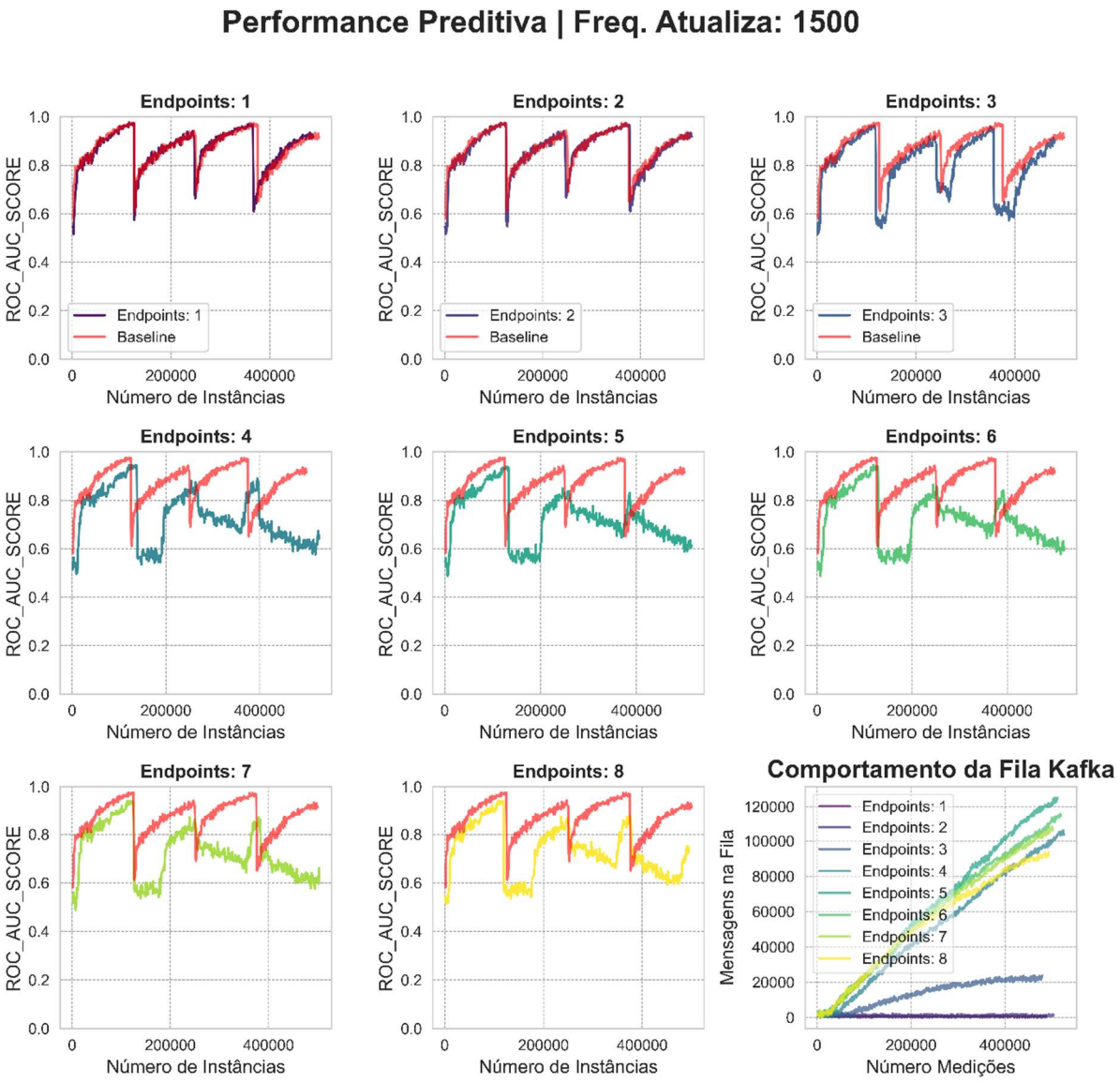

Figura 36 - Resultados dos testes para todos os valores de endpoint, juntamente com o comportamento da fila Kafka. Fonte: autor.

Assim, fica evidente que a queda na performance preditiva está diretamente relacionada ao aumento de instâncias na fila *Kafka*, o que, por sua vez, tem alta relação com o aumento de *endpoints* ativos no sistema. É importante lembrar que, quanto maior o número de *endpoints* ativos, maior é o fluxo de instâncias rotuladas publicadas no tópico *Kafka*. Quando as filas se tornam muito grandes, o tempo entre a chegada de uma instância no tópico e seu consumo pelo módulo de atualização e versionamento aumenta. Esse atraso no consumo das instâncias leva à criação de modelos desatualizados em relação à realidade atual dos dados, resultando em uma recuperação lenta da performance preditiva. Números

mais elevados de *endpoints* ativos, dependendo do modelo que está se utilizando, pode até levar a um colapso geral da capacidade preditiva.

Conclui-se, portanto, que o número de *endpoints* deve ser cuidadosamente ajustado para equilibrar o fluxo de instâncias e a capacidade de processamento do sistema. Um número excessivo de *endpoints* não apenas leva a filas longas e a uma queda significativa na performance preditiva, mas também pode não melhorar de forma satisfatória o *throughput* do sistema. Otimizar esse parâmetro é crucial para manter a precisão dos modelos e garantir uma resposta ágil às mudanças nos dados.

5.9 Discussão dos resultados dos testes

Nesta seção, serão discutidos os resultados obtidos nas avaliações realizadas, com foco nas questões de avaliação previamente estabelecidas. As questões abordam aspectos do sistema proposto, como a performance preditiva dos modelos em cenários de atualização incremental, a escalabilidade do módulo de inferência, e os impactos da frequência de versionamento e do número de *endpoints* ativos no sistema.

QA1 – Qual é a performance preditiva do modelo em um cenário de atualização incremental?

Os resultados demonstram que a atualização incremental dos modelos, em um cenário onde o modelo é atualizado e imediatamente utilizado para a predição da próxima instância (atualização 1 para 1), permite aos modelos baseados nos *learners* ARF e HT uma rápida adaptação às mudanças nos dados. No entanto, os modelos baseados no *learner* NB, devido à sua alta elasticidade, apresentaram uma performance menos significativa, com a métrica se estabilizando em torno de um valor médio. Apesar dessas variações, a performance preditiva geral se manteve satisfatória em todos os testes.

QA2 – Qual é a escalabilidade de inferência da abordagem proposta sem a execução de atualizações incrementais?

Os resultados mostram que, embora o escalonamento do módulo de inferência aumente significativamente o consumo de instâncias por segundo, esse aumento não é linear devido ao *overhead* do sistema. O desempenho real se distancia do ideal à medida que o número de *endpoints* ativos cresce (real em média 30% menor que o ideal para 8

endpoints ativos). Modelos baseados no *learner* ARF têm um consumo menor em comparação aos demais *learners*, devido ao seu tamanho e complexidade. Assim, o paralelismo do sistema é eficaz, mas principalmente quando o número de *endpoints* ativos é moderado.

QA3 – Qual é o impacto da frequência de versionamento do modelo no *throughput* e na performance preditiva do sistema?

Os resultados mostram que a frequência de disponibilização dos modelos impacta o número de instâncias na fila *Kafka*, onde, em diversos cenários, passou de 100 mil instâncias na fila para valores do parâmetro próximos ou igual a 1000 eventos, o que refletiu significativamente nos demais resultados. Em relação ao *throughput*, valores mais elevados para o parâmetro (2000 eventos) tendem a aumentar o consumo de instâncias (em diversos cenários ficou próximo de 800 instâncias por segundo) e o número de modelos atualizados por segundo (em diversos cenários se disponibilizou um modelo atualizado em menos de 2,5 segundos), porém, esses valores podem ser impactados pelo *overhead* do sistema. A frequência também afeta a performance preditiva dos modelos de maneiras distintas, com cada tipo de *learner* respondendo de forma diferente às alterações na frequência. De modo geral, valores mais altos para o parâmetro tendem a proporcionar uma melhor performance preditiva em comparação com valores mais baixos. Isso ocorre porque, com valores mais baixos, a serialização dos modelos é mais frequente, o que aumenta o consumo de recursos computacionais, resultando em uma queda na performance preditiva e no *throughput* do sistema.

QA4 – Qual é o impacto do número de *endpoints* no *throughput* e na performance preditiva do sistema?

O aumento no número de *endpoints* ativos no sistema impacta diretamente na fila *Kafka*, no *throughput* e na performance preditiva dos modelos. Com um maior número de *endpoints*, a fila *Kafka* tende a crescer significativamente (para a maioria dos cenários já se observa esse aumento a partir de 3 *endpoints* ativos), o que leva a uma queda no *throughput* do sistema e uma degradação da performance preditiva. Isso ocorre porque o alto número de instâncias na fila, aumenta o tempo entre a chegada e o consumo das instâncias, resultando em modelos desatualizados e com recuperação mais lenta de sua performance preditiva.

A análise dos resultados evidencia que a configuração cuidadosa dos parâmetros de frequência de disponibilização e do número de *endpoints* é essencial para equilibrar eficientemente o *throughput* do sistema, o tamanho da fila *Kafka* e a performance preditiva

dos modelos. Ajustar esses parâmetros de forma apropriada permite otimizar o consumo de instâncias por segundo e minimizar os atrasos no processamento, evitando o acúmulo excessivo de dados na fila *Kafka* e mantendo a precisão dos modelos.

Além disso, as características intrínsecas de cada *learner* e o tamanho de cada modelo têm um impacto significativo no comportamento do sistema. A relação entre elasticidade e plasticidade de cada modelo interfere diretamente na sua performance preditiva. Por exemplo, modelos baseados no *learner* NB, que são pouco impactados pelas diferentes configurações dos parâmetros, apresentam uma performance mais estável. De modo semelhante, modelos mais pesados computacionalmente, como os baseados no *learner* ARF, também tiveram sua performance menos afetada pelas configurações dos parâmetros. Por outro lado, os modelos gerados com base no *learner* HT, que possuem uma relação intermediária entre elasticidade e plasticidade e tamanhos consideravelmente pequenos, sofreram grandes impactos com as diferentes configurações dos parâmetros.

É importante destacar também que esses resultados foram obtidos com os módulos trabalhando sob limitação de *hardware* significativas (*endpoints* de inferência = ½ cores e *endpoint* de atualização = 2 cores). Essa limitação foi importante para evidenciar os gargalos do sistema. Portanto, ajustar as limitações de *hardware* de forma correta ajudam não somente a mitigar os problemas evidenciados, mas também a melhorar os valores dos obtidos.

Dessa forma, pode-se concluir que a escolha dos parâmetros de configuração é fundamental para a performance geral do sistema. Os resultados mostram que, embora a arquitetura proposta seja robusta e capaz de operar com eficiência em uma variedade de cenários, a performance ótima é alcançada através de uma configuração detalhada e personalizada dos parâmetros, que deve levar em consideração tanto as características dos *learners* quanto às necessidades específicas do ambiente de produção.

5.10 Considerações finais

Esse capítulo discorreu sobre os resultados dos experimentos executados com o intuito de validar a arquitetura proposta.

Inicialmente foram apresentados os *datasets* e *learners* utilizados nos testes. Em seguida, foi discutida a abordagem escolhida para a condução dos experimentos, onde criou-se um baseline para referência. Em seguida buscou-se validar o escalonamento do módulo de inferência, o impacto que o parâmetro frequência de disponibilização e o parâmetro número de *endpoints* ativos exercem no sistema. Por fim, foi realizada uma discussão sobre os resultados alcançados com os experimentos.

No capítulo a seguir, será apresentada a conclusão deste trabalho, elencando as vantagens e pontos críticos da arquitetura proposta, além de uma breve descrição das próximas atividades a serem desenvolvidas.

Capítulo 6

Conclusão

Este trabalho apresentou uma proposta de arquitetura para servir modelos de ML *online* em produção, com capacidade de atualização incremental e frequência de versionamento do modelo. A arquitetura foi desenvolvida com base no conceito de microsserviços conteinerizados, garantindo portabilidade e flexibilidade para diferentes sistemas, incluindo ambientes de nuvem. Essa adaptabilidade é crucial para que a arquitetura possa ser aplicada de maneira eficaz em diversos cenários.

A revisão da literatura revelou que há poucos trabalhos que abordam soluções para os desafios de atualização incremental de modelos em produção. Embora as abordagens tradicionais de *Stream Learning* realizem inferências *online*, elas geralmente atualizam seus modelos periodicamente através de processamento em lotes, limitando a capacidade de adaptação do sistema às mudanças nos dados. Em contraste, a arquitetura proposta busca superar esses desafios ao combinar atualização incremental contínua com uma operação eficiente em cenários de alta demanda seguindo as práticas de MLOps.

Implementar a atualização incremental em modelos de produção apresenta desafios como a necessidade de disponibilizar rapidamente modelos atualizados para inferência, gerenciar múltiplos modelos gerados e lidar com grandes fluxos de dados. A arquitetura desenvolvida foi projetada para mitigar esses problemas, oferecendo uma solução geral para *Stream Learning*, que permite a atualização contínua dos modelos e sua disponibilização para inferência em uma frequência configurável. Além disso, ela suporta um serviço de inferência escalável, capaz de atender grandes volumes de solicitações sem comprometer a eficiência do sistema.

Testes foram realizados com o objetivo de validar a arquitetura proposta. O protocolo desenvolvido para a execução dos experimentos incluiu a geração de um *baseline* que serviu como referência para os demais testes, uma avaliação da capacidade de escalonamento horizontal do módulo de inferência e uma análise de como os parâmetros

de frequência de disponibilização e número de *endpoints* impactam o *throughput* e a performance preditiva do sistema.

Os resultados dos experimentos enfatizaram a importância de uma configuração cuidadosa dos parâmetros para equilibrar fatores como *throughput* do sistema, tamanho da fila *Kafka* e performance preditiva dos modelos. Mesmo com cenários extremos e limitações de *hardwares* consideráveis, a arquitetura demonstrou sua capacidade de disponibilizar modelos atualizados em tempos baixos, atender às solicitações dos usuários e manter a performance preditiva dos modelos, o que reforça sua robustez.

Em resumo, a arquitetura proposta não só demonstrou ser eficiente em termos de recursos computacionais, como também se mostrou robusta e flexível para atender às diversas demandas de inferência em tempo real. A capacidade de disponibilizar modelos em tempos quase reais, mesmo em cenários complexos, reforça seu potencial para aplicações que exigem alta reatividade e precisão em ambientes de produção. A flexibilidade proporcionada pela parametrização permite uma adaptação precisa às necessidades específicas de cada contexto, garantindo que a solução se mantenha eficaz e ajustável em diferentes situações operacionais. Assim, a arquitetura representa uma contribuição significativa para o campo de ML *online*, oferecendo uma solução que equilibra eficiência, reatividade e precisão.

Como trabalhos futuros, pretende-se explorar o comportamento da arquitetura ao lidar com múltiplos modelos operando simultaneamente. Isso inclui a análise de como a estrutura gerenciará diferentes fluxos de dados para inferência e atualização, bem como o versionamento de diversos modelos em paralelo. Além disso, será investigada a possibilidade de acoplar um detector à arquitetura para automatizar a frequência de disponibilização e atualização de modelos, possibilitando assim ajustá-la conforme as necessidades do ambiente.

Outro ponto a ser desenvolvido no futuro, envolve a adaptação da arquitetura para trabalhar com minilotes de dados, permitindo suporte a algoritmos que não operam de forma *full-stream*. Pretende-se também avaliar o desempenho da arquitetura em aplicações reais que demandam alta frequência de atualização, verificando sua capacidade de atender a requisitos rigorosos em cenários práticos.

E por fim, conduzir testes em diferentes configurações de hardware, destacando a flexibilidade da arquitetura quanto à parametrização e sua capacidade de se adaptar a ambientes variados. A meta é compreender e otimizar o desempenho da arquitetura em

cenários mais complexos e diversificados, assegurando sua robustez, escalabilidade e eficiência mesmo diante de demandas extremas.

Referências

[1] S. A. Oke, “A literature review on artificial intelligence,” *International Journal of Information and Management Sciences*, vol. 19, no. 4. 2008. doi: 10.32350/air.11.01.

[2] P. P. Shinde and S. Shah, “A Review of Machine Learning and Deep Learning Applications,” in *Proceedings - 2018 4th International Conference on Computing, Communication Control and Automation, ICCUBEA 2018*, 2018. doi: 10.1109/ICCUBEA.2018.8697857.

[3] J. A. Pruneski *et al.*, “The development and deployment of machine learning models,” *Knee Surgery, Sports Traumatology, Arthroscopy*, vol. 30, no. 12. 2022. doi: 10.1007/s00167-022-07155-4.

[4] V. Chaoji, R. Rastogi, and G. Roy, “Machine learning in the real world,” *Proc. VLDB Endow.*, vol. 9, no. 13, pp. 1597–1600, 2015, doi: 10.14778/3007263.3007318.

[5] H. M. Gomes, J. P. Barddal, A. F. Enembreck, and A. Bifet, “A survey on ensemble learning for data stream classification,” *ACM Computing Surveys*, vol. 50, no. 2. 2017. doi: 10.1145/3054925.

[6] S. Agrahari and A. K. Singh, “Concept Drift Detection in Data Stream Mining: A literature review,” *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 10. 2022. doi: 10.1016/j.jksuci.2021.11.006.

[7] R. Ashmore, R. Calinescu, and C. Paterson, “Assuring the Machine Learning Lifecycle: Desiderata, Methods, and Challenges,” *ACM Computing Surveys*, vol. 54, no. 5. 2021. doi: 10.1145/3453444.

[8] A. M. Burgueño-Romero, C. Barba-González, and J. F. Aldana-Montes, “Big Data-driven MLOps workflow for annual high-resolution land cover classification models,” *Futur. Gener. Comput. Syst.*, vol. 163, no. August 2024, p. 107499, 2025, doi: 10.1016/j.future.2024.107499.

[9] D. Kreuzberger, N. Kuhl, and S. Hirschl, “Machine Learning Operations (MLOps): Overview, Definition, and Architecture,” *IEEE Access*, vol. 11, 2023, doi: 10.1109/ACCESS.2023.3262138.

[10] A. Paleyes, R. G. Urma, and N. D. Lawrence, “Challenges in Deploying Machine Learning: A Survey of Case Studies,” *ACM Comput. Surv.*, vol. 55, no. 6, 2022, doi: 10.1145/3533378.

[11] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, “Batch: Machine learning inference serving on serverless platforms with adaptive batching,” in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2020. doi: 10.1109/SC41405.2020.00073.

[12] S. Benson Edwin Raj and A. Annie Portia, “Analysis on credit card fraud detection methods,” in *2011 International Conference on Computer, Communication and Electrical Technology, ICCCET 2011*, 2011. doi: 10.1109/ICCCET.2011.5762457.

[13] G. Theocharous and P. S. Thomas, “Ad Recommendation Systems for Life-Time Value Optimization Categories and Subject Descriptors,” *WWW 2015 Companion Proc. 24th Int. Conf. World Wide Web*, 2015.

[14] M. Zaharia *et al.*, “Accelerating the Machine Learning Lifecycle with MLflow,” *Bull. IEEE Comput. Soc. Tech. Comm. Data Eng.*, 2018.

[15] AWS, “Amazon SageMaker.” [Online]. Available: https://aws.amazon.com/pt/sagemaker

[16] Valliappa Lakshmanan, *Data Science on the Google Cloud Platform*. 2022.

[17] N. Alshuqayran, N. Ali, and R. Evans, “A systematic mapping study in microservice architecture,” in *Proceedings - 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications, SOCA 2016*, 2016. doi: 10.1109/SOCA.2016.15.

[18] O. Bentaleb, A. S. Z. Belloum, A. Sebaa, and A. El-Maouhab, “Containerization technologies: taxonomies, applications and challenges,” *J. Supercomput.*, vol. 78, no. 1, 2022, doi: 10.1007/s11227-021-03914-1.

[19] S. Horchidan, E. Kritharakis, V. Kalavri, and P. Carbone, “Evaluating model serving strategies over streaming data,” in *Proceedings of the 6th Workshop on Data Management for End-To-End Machine Learning, DEEM 2022 - In conjunction with the 2022 ACM SIGMOD/PODS Conference*, 2022. doi: 10.1145/3533028.3533308.

[20] M. Zufle, F. Erhard, and S. Kounev, “Machine Learning Model Update Strategies for Hard Disk Drive Failure Prediction,” in *Proceedings - 20th IEEE International Conference on Machine Learning and Applications, ICMLA 2021*, 2021. doi: 10.1109/ICMLA52953.2021.00223.

[21] M. Schlegel and K. U. Sattler, “Management of Machine Learning Lifecycle Artifacts: A Survey,” *SIGMOD Rec.*, vol. 51, no. 4, 2023, doi: 10.1145/3582302.3582306.

[22] T. Schröder and M. Schulz, “Monitoring machine learning models: a categorization of challenges and methods,” *Data Sci. Manag.*, vol. 5, no. 3, pp. 105–116, Sep. 2022, doi: 10.1016/j.dsm.2022.07.004.

[23] S. R. Upadhyaya, “Parallel approaches to machine learning - A comprehensive survey,” *J. Parallel Distrib. Comput.*, vol. 73, no. 3, 2013, doi: 10.1016/j.jpdc.2012.11.001.

[24] S. Garg, P. Pundir, G. Rathee, P. K. Gupta, S. Garg, and S. Ahlawat, “On Continuous Integration / Continuous Delivery for Automated Deployment of Machine Learning Models using MLOps,” in *Proceedings - 2021 IEEE 4th International Conference on Artificial Intelligence and Knowledge Engineering, AIKE 2021*, 2021. doi: 10.1109/AIKE52691.2021.00010.

[25] N. Baumann, E. Kusmenko, J. Ritz, B. Rumpe, and M. B. Weber, “Dynamic data management for continuous retraining,” in *Proceedings - ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems, MODELS 2022: Companion Proceedings*, 2022. doi: 10.1145/3550356.3561568.

[26] M. Barry *et al.*, “StreamMLOps: Operationalizing Online Learning for Big Data Streaming & Real-Time Applications,” in *Proceedings - International Conference on Data Engineering*, 2023. doi: 10.1109/ICDE55515.2023.00272.

[27] N. J. Yadwadkar, F. Romero, Q. Li, and C. Kozyrakis, “A Case for Managed and Model-less Inference Serving,” in *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019*, 2019. doi: 10.1145/3317550.3321443.

[28] M. Komisarek, M. Choraś, R. Kozik, and M. Pawlicki, “Real-time stream processing tool for detecting suspicious network patterns using machine learning,” in *ACM International Conference Proceeding Series*, 2020. doi: 10.1145/3407023.3409189.

[29] I. L. Markov *et al.*, “Looper: An End-to-End ML Platform for Product Decisions,” in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2022. doi: 10.1145/3534678.3539059.

[30] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang, “Learning under Concept Drift: A Review,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 12. 2019. doi: 10.1109/TKDE.2018.2876857.

[31] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, “A design science research methodology for information systems research,” *J. Manag. Inf. Syst.*, vol. 24, no. 3, 2007, doi: 10.2753/MIS0742-1222240302.

[32] S. Agarwal, “Data mining: Data mining concepts and techniques,” in *Proceedings - 2013 International Conference on Machine Intelligence Research and Advancement, ICMIRA 2013*, 2014. doi: 10.1109/ICMIRA.2013.45.

[33] S. H. Liao, P. H. Chu, and P. Y. Hsiao, “Data mining techniques and applications - A decade review from 2000 to 2011,” *Expert Systems with Applications*, vol. 39, no. 12. 2012. doi: 10.1016/j.eswa.2012.02.063.

[34] L. Siguenza-Guzman, V. Saquicela, E. Avila-Ordóñez, J. Vandewalle, and D. Cattrysse, “Literature Review of Data Mining Applications in Academic Libraries,” *J. Acad. Librariansh.*, vol. 41, no. 4, 2015, doi: 10.1016/j.acalib.2015.06.007.

[35] P. Ristoski and H. Paulheim, “Semantic Web in data mining and knowledge discovery: A comprehensive survey,” *Journal of Web Semantics*, vol. 36. 2016. doi: 10.1016/j.websem.2016.01.001.

[36] J. P. Barddal, H. M. Gomes, F. Enembreck, and B. Pfahringer, “A survey on feature drift adaptation: Definition, benchmark, challenges and future directions,” *J. Syst. Softw.*, vol. 127, 2017, doi: 10.1016/j.jss.2016.07.005.

[37] V. M. A. Souza, D. M. dos Reis, A. G. Maletzke, and G. E. A. P. A. Batista, “Challenges in benchmarking stream learning algorithms with real-world data,” *Data Min. Knowl. Discov.*, vol. 34, no. 6, 2020, doi: 10.1007/s10618-020-00698-5.

[38] P. M. Gonçalves, S. G. T. De Carvalho Santos, R. S. M. Barros, and D. C. L. Vieira, “A comparative study on concept drift detectors,” *Expert Systems with Applications*, vol. 41, no. 18. 2014. doi: 10.1016/j.eswa.2014.07.019.

[39] B. Krawczyk, L. L. Minku, J. Gama, J. Stefanowski, and M. Woźniak, “Ensemble learning for data stream analysis: A survey,” *Inf. Fusion*, vol. 37, 2017, doi: 10.1016/j.inffus.2017.02.004.

[40] G. Heitor Murilo, R. Jesse, A. Bifet, J. P. Barddal, and J. Gama, “Machine learning for

streaming data: state of the art, challenges, and opportunities," *ACM SIGKDD Explor. Newsl.*, vol. 21, no. 2, 2019.

[41] C. Lehmann, L. Goren Huber, T. Horisberger, G. Scheiba, A. C. Sima, and K. Stockinger, "Big Data architecture for intelligent maintenance: a focus on query processing and machine learning algorithms," *J. Big Data*, vol. 7, no. 1, 2020, doi: 10.1186/s40537-020-00340-7.

[42] P. S. Janardhanan, "Project repositories for machine learning with TensorFlow," in *Procedia Computer Science*, 2020. doi: 10.1016/j.procs.2020.04.020.

[43] The Apache Software Foundation 2023, "Apache Airflow." [Online]. Available: https://airflow.apache.org/

[44] The Kubeflow Authors, "Kubeflow." [Online]. Available: https://www.kubeflow.org/

[45] Seldon Technologies Limited, "Seldon." [Online]. Available: https://www.seldon.io/

[46] L. MLflow Project, a Series of LF Projects, "Mlflow." [Online]. Available: https://mlflow.org/

[47] Microsoft, "Azure Machine Learning." [Online]. Available: https://azure.microsoft.com/en-us/products/machine-learning

[48] Google, "Google Cloud AI Plataform." [Online]. Available: https://console.cloud.google.com/marketplace/product/google-cloud-platform/cloud-machine-learning-engine?project=curso-403115

[49] The University of Waikato, "MOA - Massive Online Analysis." [Online]. Available: https://moa.cms.waikato.ac.nz/

[50] Oracle, "Java." [Online]. Available: https://www.java.com/pt-BR/

[51] A. Bifet *et al.*, "MOA: Massive Online Analysis, a Framework for Stream Classification and Clustering," *J. Mach. Learn. Res. Work. Conf. Proc.*, vol. 11, 2010.

[52] Apache Software Foundation, "SAMOA." [Online]. Available: http://samoa.incubator.apache.org

[53] C. Gambella, B. Ghaddar, and J. Naoum-Sawaya, "Optimization problems for machine learning: A survey," *European Journal of Operational Research*, vol. 290, no. 3. 2021. doi: 10.1016/j.ejor.2020.08.045.

[54] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. Zulkernine, and S. Khan, "A Survey of Distributed Data Stream Processing Frameworks," *IEEE Access*, vol. 7, pp. 154300–154316, 2019, doi: 10.1109/ACCESS.2019.2946884.

[55] Apache Software Foundation, "Apache Storm." [Online]. Available: https://storm.apache.org/

[56] Apache Software Foundation, "Apache Samza." [Online]. Available: https://samza.apache.org/

[57] Lightbend, "Scala." [Online]. Available: https://www.scala-lang.org/

[58] Python Software Foundation, "Python." [Online]. Available: https://www.python.org/

[59] “Scikit-Multiflow.” [Online]. Available: https://scikit-multiflow.github.io/

[60] Python Software Foundation, “Creme.” [Online]. Available: https://pypi.org/project/creme/

[61] Dunder Data, “River.” [Online]. Available: https://riverml.xyz/

[62] “Scikit-Learn.” [Online]. Available: https://scikit-learn.org/

[63] “NumPy.” [Online]. Available: https://numpy.org/

[64] “SciPy.” [Online]. Available: https://scipy.org/

[65] “Juputer.” [Online]. Available: https://jupyter.org/

[66] A. Bifet, S. Maniu, J. Qian, G. Tian, C. He, and W. Fan, “StreamDM: Advanced Data Mining in Spark Streaming,” in *Proceedings - 15th IEEE International Conference on Data Mining Workshop, ICDMW 2015*, 2016. doi: 10.1109/ICDMW.2015.140.

[67] Apache Software Foundation, “Spark Streaming.” [Online]. Available: https://spark.apache.org/docs/latest/streaming-programming-guide.html

[68] Huawei, “Ark Lab.” [Online]. Available: http://dev3.noahlab.com.hk/

[69] Apache Software Foundation, “Hadoop.” [Online]. Available: https://hadoop.apache.org/

[70] W. Jamil, N. C. Duong, W. Wang, C. Mansouri, S. Mohamad, and A. Bouchachia, “Scalable online learning for flink: SOLMA library,” in *ACM International Conference Proceeding Series*, 2018. doi: 10.1145/3241403.3241438.

[71] Apache Software Foundation, “Apache Flink.” [Online]. Available: https://flink.apache.org/

[72] D. Xu, D. Wu, X. Xu, L. Zhu, and L. Bass, “Making real time data analytics available as a service,” in *QoSA 2015 - Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures, Part of CompArch 2015*, 2015. doi: 10.1145/2737182.2737186.

[73] Apache Software Foundation, “Apache Spark.” [Online]. Available: https://spark.apache.org/

[74] Apache Software Foundation, “MLlib.” [Online]. Available: https://spark.apache.org/mllib/

[75] R. Minon, J. Diaz-De-Arcaya, A. I. Torre-Bastida, G. Zarate, and A. Moreno-Fernandez-De-Leceta, “MLPacker: A Unified Software Tool for Packaging and Deploying Atomic and Distributed Analytic Pipelines,” in *2022 7th International Conference on Smart and Sustainable Technologies, SpliTech 2022*, 2022. doi: 10.23919/SpliTech55088.2022.9854211.

[76] R. Ramanath *et al.*, “Lambda Learner: Fast Incremental Learning on Data Streams,” in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2021. doi: 10.1145/3447548.3467172.

[77] F. Carcillo, A. Dal Pozzolo, Y. A. Le Borgne, O. Caelen, Y. Mazzer, and G. Bontempi, “SCARFF:

A scalable framework for streaming credit card fraud detection with spark," *Inf. Fusion*, vol. 41, 2018, doi: 10.1016/j.inffus.2017.09.005.

[78] Apache Software Foundation, "Apache Kafka." [Online]. Available: https://kafka.apache.org/

[79] Apache Software Foundation, "Apache Cassandra." [Online]. Available: https://cassandra.apache.org/

[80] J. Kranjc, R. Orač, V. Podpečan, N. Lavrač, and M. Robnik-Šikonja, "ClowdFlows: Online workflows for distributed big data mining," *Futur. Gener. Comput. Syst.*, vol. 68, 2017, doi: 10.1016/j.future.2016.07.018.

[81] Ecma International, "Javascript." [Online]. Available: https://www.javascript.com/

[82] Django Software Foundation, "Django." [Online]. Available: https://www.djangoproject.com/

[83] The University of Waikato, "Weka." [Online]. Available: https://www.cs.waikato.ac.nz/ml/weka/

[84] "Orange." [Online]. Available: https://orangedatamining.com/

[85] Apache Software Foundation, "Elasticsearch." [Online]. Available: https://www.elastic.co/pt/elasticsearch

[86] Apache Software Foundation, "Kibana." [Online]. Available: https://www.elastic.co/pt/elasticsearch

[87] S. Anandan, M. Bogoevici, G. Renfro, I. Gopinathan, and P. Peralta, "Industry paper: Spring XD - A modular distributed stream and batch processing system," in *DEBS 2015 - Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, 2015. doi: 10.1145/2675743.2771879.

[88] Vm. Inc, "Rabbitmq." [Online]. Available: https://www.rabbitmq.com/

[89] Spreing, "Redis." [Online]. Available: https://docs.spring.io/spring-data/data-redis/docs/current/reference/html/

[90] Apache Software Foundation, "Apache Zookeeper." [Online]. Available: https://zookeeper.apache.org/

[91] S. Garcia-Rodriguez, M. Alshaer, and C. Gouy-Pailler, "STREAMER: A Powerful Framework for Continuous Learning in Data Streams," in *International Conference on Information and Knowledge Management, Proceedings*, 2020. doi: 10.1145/3340531.3417427.

[92] Influxdata, "InfluxDB." [Online]. Available: https://www.influxdata.com/

[93] C. Martín, P. Langendoerfer, P. S. Zarrin, M. Díaz, and B. Rubio, "Kafka-ML: Connecting the data stream with ML/AI frameworks," *Futur. Gener. Comput. Syst.*, vol. 126, 2022, doi: 10.1016/j.future.2021.07.037.

[94] "Tensorflow." [Online]. Available: https://www.tensorflow.org/

[95] Docker Inc, “Docker.” [Online]. Available: https://www.docker.com/

[96] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, “Clipper: A low-latency online prediction serving system,” in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017*, 2017.

[97] D. Baylor *et al.*, “TFX: A TensorFlow-based production-scale machine learning platform,” in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017. doi: 10.1145/3097983.3098021.

[98] A. Pareek, B. Zhang, and B. Khaladkar, “A demonstration of striim a streaming integration and intelligence platform,” *DEBS 2019 - Proc. 13th ACM Int. Conf. Distrib. Event-Based Syst.*, pp. 236–239, 2019, doi: 10.1145/3328905.3332519.

[99] S.-L. Developers, “Metrics ROC_AUC_SCORE.” [Online]. Available: https://scikit-learn.org/0.16/modules/generated/sklearn.metrics.roc_auc_score.html

[100] C. Pallets, “API Flask.” [Online]. Available: https://flask.palletsprojects.com/en/3.0.x/api/

[101] Canonical Ltd, “Microk8s.” [Online]. Available: https://microk8s.io/

[102] D. Marrón Vida, “Improving decision tree and neural network learning for evolving data-streams,” 2019, [Online]. Available: https://widgets.ebscohost.com/prod/customerspecific/ns000545/customproxy.php?url=https://search.ebscohost.com/login.aspx?direct=true&db=edstdx&AN=edstdx.10803.668371&amp%0Alang=pt-pt&site=eds-live&scope=site

[103] D. Ben-Shimon, B. Shapira, A. Tsikinovsky, L. Rokach, M. Friedmann, and J. Hoerle, “RecSys challenge 2015 and the YOOCHOOSE dataset,” in *RecSys 2015 - Proceedings of the 9th ACM Conference on Recommender Systems*, 2015. doi: 10.1145/2792838.2798723.

[104] F. J. Yang, “An implementation of naive bayes classifier,” in *Proceedings - 2018 International Conference on Computational Science and Computational Intelligence, CSCI 2018*, 2018. doi: 10.1109/CSCI46756.2018.00065.

[105] H. M. Gomes *et al.*, “Adaptive random forests for evolving data stream classification,” *Mach. Learn.*, vol. 106, no. 9–10, 2017, doi: 10.1007/s10994-017-5642-8.

[106] F. Banar, A. Tabatabaei, and M. Saleh, “Stream Data Classification with Hoeffding Tree : An Ensemble Learning Approach,” in *2023 9th International Conference on Web Research, ICWR 2023*, 2023. doi: 10.1109/ICWR57742.2023.10139228.