INSTITUTO FEDERAL DE SANTA CATARINA

ANDRÉ LUIZ FARACO MAZUCHELI

Aplicação dos Conceitos de Engenharia de Aprendizado de Máquina em Produção em um Sistema de Detecção de Anomalias

São José - SC

fevereiro/2024

APLICAÇÃO DOS CONCEITOS DE ENGENHARIA DE APRENDIZADO DE MÁQUINA EM PRODUÇÃO EM UM SISTEMA DE DETECÇÃO DE ANOMALIAS

Projeto de Trabalho de conclusão de curso apresentado à Coordenadoria do Curso de Engenharia de Telecomunicações do campus São José do Instituto Federal de Santa Catarina.

Orientador: Prof. Mario de Noronha Neto, Dr.

São José - SC

fevereiro/2024

André Luiz Faraco Mazucheli

Aplicação dos Conceitos de Engenharia de Aprendizado de Máquina em Produção em um Sistema de Detecção de Anomalias

Este trabalho foi julgado adequado para obtenção do título de Engenheiro de Telecomunicações, pelo Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina, e aprovado na sua forma final pela comissão avaliadora abaixo indicada.

São José - SC, 22 de Fevereiro de 2024:

Prof. Mario de Noronha Neto, Dr.
Orientador
Instituto Federal de Santa Catarina

Professor Roberto Wanderley da Nóbrega, Dr.
Instituto Federal de Santa Catarina

Professor Cleber Jorge Amaral, Dr.
Instituto Federal de Santa Catarina

Este trabalho é dedicado a minha família.

AGRADECIMENTOS

Gostaria de expressar minha gratidão, em primeiro lugar, à minha família pelo apoio constante, especialmente ao meu padrasto, mãe e irmã, que estiveram ao meu lado desde o início da minha jornada acadêmica.

Sou imensamente grato ao professor Mário de Noronha por sua atenção dedicada e por desempenhar um papel fundamental em minha formação, colaborando ativamente em atividades de pesquisa que resultaram neste trabalho.

Expresso meu reconhecimento ao Instituto Federal de Santa Catarina - Câmpus São José e a todos os seus colaboradores pelo compromisso com a excelência na educação pública.

Por fim, agradeço a todos que de alguma forma fizeram parte dessa jornada desafiadora, com lembranças que serão guardadas sempre com muito carinho e aprendizados que levarei para a vida. Obrigado!

"A mente que se abre a uma nova ideia jamais voltará ao seu tamanho original."
(Albert Einstein)

RESUMO

O avanço da tecnologia, especialmente no campo de *Machine Learning*, tem impulsionado a aplicação de modelos de aprendizado de máquina em cenários reais. No entanto, a transição de um modelo funcional de um ambiente de desenvolvimento, como o Jupyter Notebook, para um ambiente de produção enfrenta desafios como escalabilidade, segurança, eficiência operacional e manutenção contínua. Essa discrepância entre o desenvolvimento e a produção destaca a necessidade de uma abordagem mais estruturada para a implantação e operação desses sistemas em produção. Este trabalho tem como objetivo aplicar conceitos de engenharia de aprendizado de máquina em produção, para orientar a elaboração, implantação e manutenção contínua desses sistemas em produção. O foco está em explorar cada etapa do ciclo de vida desses sistemas, definindo boas práticas e ferramentas para auxiliar no processo.

Palavras-chave: MLOps. Machine Learning. Engenharia de dados.

ABSTRACT

With the advancement of technology, especially in the field of *Machine Learning*, the application of machine learning models in real-world scenarios has become increasingly common. However, the transition from a functional model in a development environment, such as Jupyter Notebook, to a production environment faces challenges such as scalability, security, operational efficiency, and continuous maintenance. This discrepancy between development and production highlights the need for a more structured approach to deploying and operating these systems in production. This work aims to apply concepts of machine learning engineering in production, to guide the development, deployment, and continuous maintenance of these systems in production. The focus is on exploring each stage of the life cycle of these systems, defining best practices and tools to assist in the process.

Keywords: MLOps. Machine Learning. Data Engineering.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

ACID Atomicidade, Consistência, Isolamento e Durabilidade.

API Application Programming Interface.

CI Continuous Integration.

CI/CD Continuous Integration/Continuous Delivery.

ECG Eletrocardiograma.

HLP Human Level Performance.

IA Inteligência Artificial.

JSON JavaScript Object Notation.

LGPD Lei Geral de Proteção de Dados Pessoais.

ML Machine Learning.

MLOps Machine Learning Operations.

NoSQL Not Only Structured Query Language.

POC Proof of Concept.

POO Programação Orientada a Objetos.

REST Representational State Transfer.

SGBD Sistema Gerenciador de Banco de Dados.

SQL Structured Query Language.

TF TensorFlow.

VM Virtual Machine.

YAML Ain't Markup Language.

SUMÁRIO

1 INTRODUÇÃO

A Inteligência Artificial (IA) tem experimentado um crescimento substancial em diversos setores, impulsionado pelo avanço tecnológico e pela disponibilidade de grandes volumes de dados. No entanto, embora as ferramentas e técnicas de Machine Learning (ML) tenham avançado, ainda existe uma lacuna significativa entre o desenvolvimento de modelos em ambientes controlados, e sua implementação em ambientes de produção. Essa disparidade destaca a complexidade adicional envolvida na transição de um modelo funcional para um ambiente real, onde a complexidade e os desafios escalam, e fatores como escalabilidade, segurança e eficiência operacional desempenham um papel crucial (ASHMORE; CALINESCU; PATERSON, 2021).

Enquanto o ambiente de desenvolvimento permite explorar algoritmos, testar modelos e iterar rapidamente, a transição para a produção exige considerações adicionais e eficiência operacional. O simples fato de tornar um modelo de ML funcional em ambiente de desenvolvimento não garante sua viabilidade em um ambiente real, onde fatores como volume de dados, latência, monitoramento e manutenção contínua desempenham um papel fundamental.

Modelos de aprendizado de máquina, do inglês ML, quando implantados em sistemas do mundo real, necessitam de flexibilidade pois existe uma variação dos dados de entrada com a variação do tempo em ambiente de produção. O desempenho do modelo em produção se degrada devido a estas frequentes mudanças nos dados. Monitorar essa estrutura se torna relevante para definir se existe a necessidade de realizar novamente o treinamento e garantir que esteja funcionando conforme o esperado, porém, acompanhar seu desempenho traz diversos desafios ao longo do ciclo de vida do aprendizado de máquina (GARG et al., 2021).

Sistemas de aprendizado de máquina possuem um ciclo de vida que procuram a aprimorar gradualmente sua eficiência. O ciclo pode ser dividido em quatro estágios básicos: escopo, dados, modelagem e implantação podendo os três últimos ser iterativos. Cada etapa possui uma série de processos e análises que conforme são aperfeiçoados melhoram a precisão e o desempenho do sistema.

Dentro deste contexto, podemos trabalhar com o conceito de engenharia de aprendizado de máquina, que também pode ser chamado de Machine Learning Operations (MLOps). Ele agrega um conjunto de ferramentas e princípios para apoiar o progresso ao longo do ciclo de vida de um projeto de ML, principalmente referente às etapas de dados, modelagem e implantação. A ideia central em MLOps é encontrar maneiras de pensar sobre o escopo de modelagem e implantação de dados e também ferramentas de *software*

para sustentar as melhores práticas. Ele permite que os desenvolvedores envolvidos no projeto colaborem e aumentem o ritmo em que os modelos de IA podem ser desenvolvidos, implantados, modelados, monitorados e retreinados (ASHMORE; CALINESCU; PATERSON, 2021).

A importância desse conceito, está no fato de que quando um algoritmo de predição é utilizado em um sistema de ML, ele ainda precisa passar por um processo de amadurecimento em produção, por fluxo iterativo do algoritmo sendo ele retreinado e ajustado para eventuais alterações em parâmetros não considerados previamente(NG, 2022a). Este processo pode não ser gerido e nem implementado muitas vezes de forma coerente.

A forma como é realizado o processo de implantação do sistema de ML em produção, está diretamente associada ao seu nível de desempenho, portanto sua estrutura deve ser cuidadosamente modelada para o cenário de aplicação.

Definir se o seu funcionamento ocorrerá em tempo real ou não, ou se ele será executado em nuvem, ou em *edge*, com o hardware e software em algum servidor local. Nestes cenários, é relevante considerar questões como latência de comunicação, que de fato pode impactar consideravelmente, segurança de dados também devem ser consideradas relevantes, já que o sistema trabalha com a análise de dados das mais diversas naturezas(NG, 2022a).

1.1 Objetivo geral

Aplicar conceitos de engenharia de aprendizado de máquina em produção, utilizando um modelo de detecção de anomalias, definindo boas práticas para gerenciar e realizar ajustes em toda a infraestrutura definida.

1.2 Objetivos específicos

Visando atingir o objetivo geral, os seguintes objetivos específicos deverão se aplicados:

1. Pesquisar e entender os conceitos de MLOps;
2. Descrever o ciclo de vida de um sistema de ML em produção;
3. Criar a infraestrutura para implantação do sistema;
4. Definir aplicação e base de dados utilizada;
5. Gerar o modelo de detecção de anomalias;
6. Implementar o modelo em um cenário de produção.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo tem a finalidade de fundamentar e explorar os conceitos e tecnologias utilizadas neste trabalho, para contextualizar a ideia e consolidar as motivações presentes nesta aplicação.

2.1 MLOps

Conforme a comunidade de ML cresce e se solidifica, uma tendência se torna evidente, o desenvolvimento e a implantação de sistemas utilizando essa tecnologia são inicialmente rápidos e econômicos, mas a manutenção a longo prazo se torna mais custosa e complexa. (SCULLEY et al., 2015).

A implantação de modelos em aprendizado de máquina vem evoluindo consideravelmente nos últimos anos, e tem sido uma área crescente de estudos. Este processo pode ser visto de forma semelhante ao estabelecido para a implantação no desenvolvimento de *software* tradicional (GARG et al., 2021).

Assim que desenvolvido, o sistema em produção deve operar de forma contínua com custo mínimo e, ao mesmo tempo, alcançar o desempenho máximo. No entanto, quando o sistema é implantado, surgem problemas do mundo real que afetam diretamente sua eficiência.

O MLOps combina os conceitos de ML com conceitos funcionais de desenvolvimento de *software* para nortear a elaboração do projeto, já o preparando para a implantação em produção. Desta forma, MLOps abrange como conceituar, construir e manter sistemas operados continuamente em produção utilizando ferramentas e metodologias bem estabelecidas para garantir a eficácia e eficiência. Diferentemente de modelagens de aprendizado de máquina padrão, concebidas em ambiente de desenvolvimento, os sistemas, quando em produção, precisam lidar com dados em constante evolução e mudanças, sejam elas graduais ou repentinas(NG, 2022a).

Objetivo do MLOps:

1. Rápida definição e desenvolvimento de modelo;
2. Rápida implantação de modelos atualizados em produção;
3. Garantia de qualidade;
4. Facilitar a comunicação entre o cientista de dados e o Engenheiro de ML.

MLOps é um processo que permite que cientistas de dados e equipes de desenvolvedores colaborem de forma otimizada para aumentar o ritmo de desenvolvimento de modelos, juntamente com integração, monitoramento e validação (NG, 2022a).

A complexidade em MLOps esta no fato de não existir processos estáticos, necessita de um engajamento com as necessidades específicas de cada projeto tendo como referência, os conceitos, as boas práticas e as ferramentas de apoio (NG, 2022a).

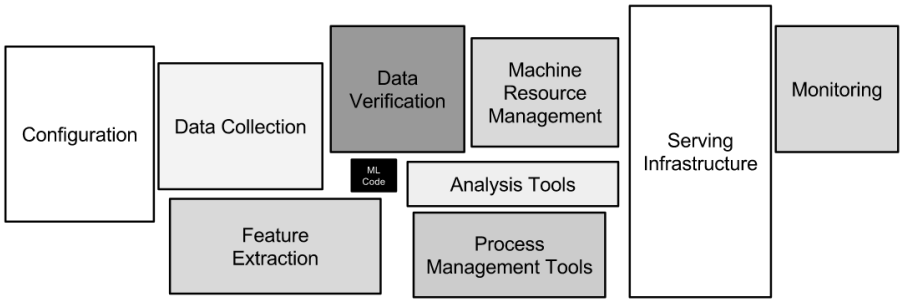
2.1.1 Componentes da infraestrutura de um sistema de aprendizado de máquina

Para garantir um sistema de aprendizado de máquina ML robusto e eficiente em produção, é necessário desenvolver uma infraestrutura para o projeto que garanta tais características.

O núcleo do sistema de fato é entregue pelo parte de código ML, e existem diversos algoritmos já implementados e testados pela comunidade científica para diferentes tipos de aprendizado de máquina, cada um com sua vantagem, desvantagens e peculiaridades.

Porém, conforme ilustra a Figura 1, quando se trata de um sistema de ML em produção, o código em si representa uma pequena fração de toda a infraestrutura, podendo representar algo entre 5-10%, de todo o conteúdo implantado, diferentemente de um ambiente de desenvolvimento por exemplo (SCULLEY et al., 2015).

Figura 1 – *Infraestrutura de um sistema de aprendizado de máquina*



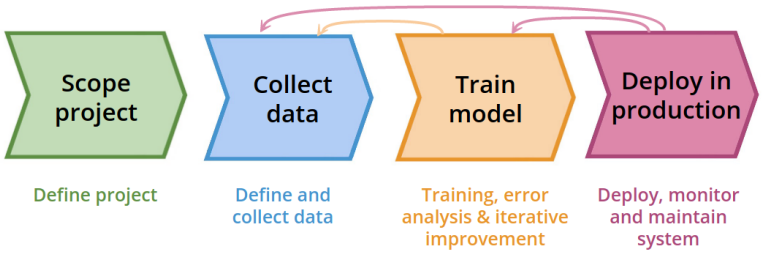
Fonte:Sculley et al. (2015)

Isso pode impactar em um grande diferencial de realizar uma Proof of Concept (POC) em ambiente de desenvolvimento, ou seja, quando apenas o algoritmo em si está aprovado antes de ser implantado em produção, sem considerar outros fatores que afetam o desempenho do projeto.

2.1.2 Ciclo de vida de um sistema de aprendizado de máquina em produção

O ciclo de vida de um sistema de ML, como ilustrado na Figura 2, é um processo iterativo e complexo. Ele desempenha um papel crucial ao segmentar e orientar o processo de ingestão de dados, que começa com a coleta dos dados usados para treinar um algoritmo de ML e culmina com a implantação desse sistema em produção (ASHMORE; CALINESCU; PATERSON, 2021). A contextualização detalhada de cada etapa do ciclo é fundamental para a consolidação dos conceitos aplicados neste trabalho.

Figura 2 – *Ciclo de vida*



Fonte:Ng (2022b)

2.1.2.1 Escopo

Conforme ilustrado na Figura 2, o ciclo é iniciado pelo **escopo do projeto**. Esta etapa tem como objetivo a especificação e o planejamento do projeto, sendo de extrema importância para o desenvolvimento adequado do mesmo (NG, 2022a).

Durante o processo de definição do escopo, é essencial abordar questões fundamentais, como *"Quais são os recursos necessários em termos de dados, tempo e pessoal para o desenvolvimento do projeto?"* e *"Quais são as métricas de sucesso do projeto?"*. Essas perguntas ajudam a garantir que o projeto seja viável tecnicamente e agregue valor ao negócio.

As métricas mais importantes a serem consideradas estão relacionadas ao sistema de ML. Elas incluem a precisão do algoritmo em cumprir seu propósito, os requisitos de *software* associados ao cenário técnico, a latência na comunicação e os recursos da infraestrutura. Além disso, é crucial considerar os recursos necessários em termos de mão de obra qualificada e a aquisição dos dados, que muitas vezes não são obtidos de forma simples ou econômica.

Também é crucial realizar uma análise minuciosa das métricas de negócio para determinar se o projeto é viável em termos de investimento. Definir esses parâmetros nos estágios iniciais do desenvolvimento de um sistema de aprendizado de máquina resulta em simplificações e otimizações significativas nas etapas subsequentes do ciclo de vida, reduzindo a probabilidade de imprevistos ocorrerem (NG, 2022a).

2.1.2.2 Dados

Tradicionalmente, esta é a etapa do ciclo mais consome tempo do desenvolvimento. Os dados passam por um processo de limpeza e rotulagem, que podem vir de várias fontes e formatos diferentes, e geralmente, necessita de um grau de normalização rígido (NG, 2022a).

Existe um processo de otimização na coleta dos dados, mesmo após o sistema já estar em produção, a necessidade a medida em que a qualidade dos dados melhora, é desejável utilizá-los para criar um modelo. Esta é a primeira necessidade de MLOps.

Pode ser realizada uma divisão, definindo um limiar que determina se o conjunto de dados é pequeno, do inglês *small data* ou grande, do inglês *big data*. Para cenários em que se aplique um *conjunto pequeno de dados*, é uma boa prática estabelecer um processo de rotulagem bem definido. O conjunto pode ser examinado manualmente e os rótulos corrigidos e refatorados facilmente devido ao pequeno volume de dados. Nos casos de se utilizar um *big data*, deve-se focar no processamento dos dados, por ser mais complexo o processo de rotulagem (NG, 2022a).

O problema da rotulagem, é que dependendo da situação, pode ser que torne a análise dos dados um processo mais complexo e custoso para o sistema. Para evitar este problema, é uma boa prática identificar possíveis redundâncias em certos cenários rotulados.

Para exemplificar, pode ser considerado um cenário em que um sistema de predição analisa *smartphones* em uma fábrica, sendo o seu objetivo, identificar possíveis imperfeições nos produtos ao final do seu ciclo de produção. Imaginando que os dados possuem duas rotulagens que identificam arranhões profundos e arranhões suaves, como a rotulagem identifica a mesma classe de imperfeição, é indicado em alguns cenários que seja realizada uma mesclagem, para otimizar a leitura dos dados, simplificando o processo, eliminando uma redundância de rótulos atacando o mesmo cenário.

Nesta etapa, é altamente recomendado não perder muito tempo com os dados, e fazer o mais rapidamente possível o ciclo de interação ilustrado na Figura 3, pois provavelmente mais inconsistências e situações não previstas surgirão nele (NG, 2022a).

Para realizar o pré-processamento dos dados, é recomendado desenvolver e implementar *scripts* que sejam replicáveis, de modo que sua validação seja flexível e os ajustes possam ser executados de forma ágil. É importante lembrar que este é um processo iterativo, sujeito a repetições inúmeras vezes.

Outra metodologia importante nesta etapa é o uso de metadados, que podem ser entendidos como os dados que referenciam outros dados. Por exemplo, no cenário da fábrica de smartphones que analisa imperfeições nos produtos usando fotos, os dados seriam as próprias fotos e os rótulos dos arranhões. Os metadados seriam os dados que

fornecem informações como o horário em que a foto foi tirada, o número da linha de produção na fábrica e as configurações da câmera que capturou a imagem.

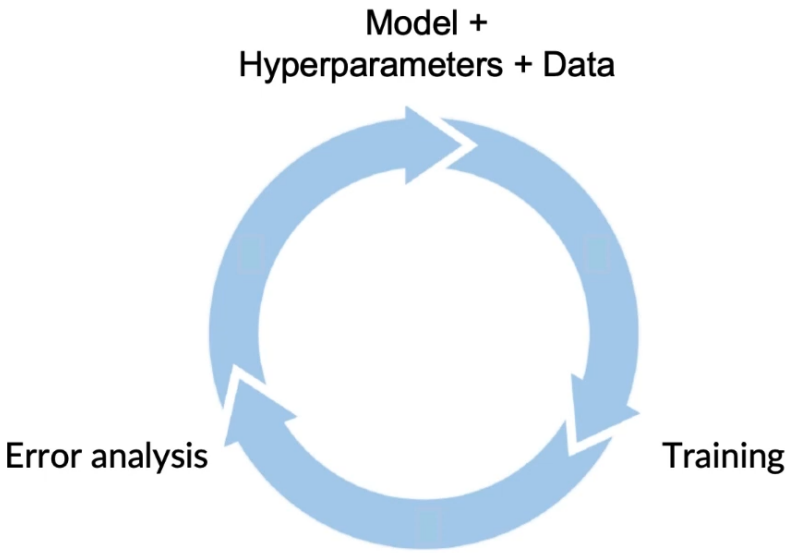
Por meio dos metadados, é possível identificar situações específicas que estão resultando em erros. Por exemplo, é possível identificar que uma linha específica de produtos em uma determinada fábrica está gerando mais erros do que os demais produtos analisados pelo algoritmo. Essas informações são essenciais para solucionar e otimizar o modelo, melhorando a análise de erros e detectando efeitos inesperados.

2.1.2.3 Modelagem

Sistemas de IA, em sua essência, são compostos de código, que representa o algoritmo e o modelo, em conjunto com os dados. Ao construir um sistema de ML, existe a necessidade de ter um modelo, e então o modelo é treinado nos dados, gerando o algoritmo de predição.

Existem também os *hiperparâmetros*, entradas adicionais, importantes para validar se existe uma taxa de aprendizado bem ajustada e um parâmetro de regularização adequado, este processo é chamado também de *tuning*.

Figura 3 – *Ciclo de modelagem*



Fonte:Ng (2022b)

Na Figura 3, está sendo ilustrado um ciclo de modelagem que ajuda a identificar como melhorar ou os dados, ou os hiperparâmetros, ou os modelos. O fato desse ciclo ser executado muito rapidamente, é a chave para melhorar o desempenho, e aqui se encontra o grande papel do MLOps, otimizar este processo (NG, 2022a). Após validado que um modelo adequado foi alcançado, é então feita a implantação.

Apesar de apresentar uma média baixa de erros ao longo do ciclo, isso não garante necessariamente que o sistema esteja cumprindo as regras de negócio estabelecidas para o projeto. Essa situação pode ser exemplificada através de um exemplo.

Ao realizar uma pesquisa no ***Google*** sobre uma receita de cozinha desejada, não há garantia de que todos os resultados exibidos pelo mecanismo de busca sejam as melhores receitas, ou que eles correspondam exatamente à receita desejada. No entanto, isso não significa que ocorreu um erro no processo, uma vez que a pesquisa foi baseada nos dados disponíveis. Por outro lado, se uma busca pela palavra *YouTube* for executada no ***Google***, o mecanismo identificará a procura por *www.youtube.com* de forma determinística.

Quando o usuário realiza uma busca clara, o algoritmo tem facilidade em identificar o melhor resultado para exibir. No entanto, nesse contexto, surgem as desproporcionalidades de relevância das informações. Mesmo que a média de erro de precisão seja baixa, considerando todos os parâmetros como iguais, é importante notar que o acerto na busca por *YouTube*, que desempenhou maior precisão, obteve o mesmo peso que o cenário da busca por uma receita, sendo menos preciso.

Para tratar este problema, existe uma metodologia cujo objetivo é realizar análises em fatias-chave, do inglês *key slices* dos dados (NG, 2022a). Digamos que foi feito um sistema de ML para decidir quais usuários provavelmente vão pagar um empréstimo, e assim então recomendar o empréstimo a instituição financeira que o fará.

Este sistema, em conformidade com a lei, não considera dados sensíveis dos clientes, como sexo, etnia ou cor. No entanto, mesmo que a média de erros seja baixa, pode ocorrer que o sistema crie tendências que erroneamente indicam que pessoas de alguma etnia, por exemplo, têm maior probabilidade de não pagar um empréstimo, levando o sistema a não recomendar o empréstimo. Essa situação revela um problema no âmbito do negócio.

Uma analogia semelhante pode ser feita com um sistema de comércio eletrônico que privilegia apenas produtos de marcas conhecidas em detrimento das marcas menores. Isso pode ser prejudicial para o negócio, uma vez que há demanda para todas as marcas.

Uma situação semelhante, está associada ao conceito de classes raras, do inglês, *rare classes*, onde ocorrem situações pouco frequentes que, mesmo com uma análise altamente precisa, podem ser ignoradas devido à sua raridade.

Este caso esta diretamente associado à ideia de distribuição distorcida dos dados. Contextualizando para um sistema que visa identificar uma doença através de dados dos exames de pessoas, onde 99% da população não possui ela e 1% possui. O algoritmo mesmo tendo alta precisão, com margem de erro de apenas 1%, por exemplo, poderá afirmar que 0% da população possuem a doença, pela margem de erro, segue uma precisão alta, porém a informação pode ser desastrosa e se torna de certa forma inútil. O objetivo do MLOps nestes cenários, é garantir que o sistema de ML desenvolvido, atenderá as

regras de negócio (NG, 2022a).

Ao longo de todo o processo de modelagem, busca-se constantemente aumentar a precisão do algoritmo. No entanto, é crucial exercer cautela ao definir o limiar que determina que o nível de precisão é adequado.

Para isso, deve-se observar novamente o tipo de dados com que se esta trabalhando, e as formas em que estão organizados, que podem ser basicamente divididas em duas, estruturados e não estruturados, e esta informação ajuda a parametrizar o uso de uma *baseline* para ter referências de precisão em um algoritmo, indicada ser utilizada para ter sucesso a longo prazo com o projeto de ML. Este projeto terá foco na análise de dados não estruturados.

Dados estruturados são os *BigData*, planilhas com grandes volumes de informação, em que humanos não são tão confiáveis para analisar. E dados não estruturados, tendem a ser dados que humanos são bons em interpretar, como imagens, áudios e textos. Neste contexto, usar o conceito de Human Level Performance (HLP) como *baseline* é uma abordagem recomendável, pois a interpretação humana é uma referência confiável para validar a precisão do algoritmo..

Para estabelecer uma *baseline* adequada para cada caso, além de priorizar o HLP para dados não estruturados, também é uma boa prática realizar pesquisas na literatura, ou examinar resultados em testes implementados em projetos de código aberto. Também pode ser construída por testes em produção do mesmo modelo com versões mais simples e compactas, para através dos resultados, ter como referência neveis adequados de precisão. Para este caso, a agilidade ao implantar o sistema em produção facilita de forma considerável o processo.

Compreender que antes de iniciar o treinamento do algoritmo com milhares de dados, é mais eficiente começar com conjuntos menores. Treinar com esses conjuntos permite encontrar *bugs* de forma mais rápida e genérica. Por exemplo, no caso de um sistema de reconhecimento de imagens, se o algoritmo não funcionar com 100 imagens durante o treinamento, é improvável que funcione com 1000 ou 10000 imagens (NG, 2022a).

Portanto, uma das etapas mais importantes antes de iniciar o treinamento nos modelos de ML é realizar a análise dos erros. Isso ajuda a identificar oportunidades de melhoria no desempenho do algoritmo. Essa abordagem permite uma iteração mais eficiente e precisa durante o desenvolvimento do modelo.

Na Figura 4, é ilustrada uma boa prática para realizar a análise dos erros em um sistema de reconhecimento de voz que converte áudio em texto. Em cada caso em que o algoritmo interpretou erroneamente uma frase, é possível associar rótulos que indicam possíveis motivos para essa distorção. Esses rótulos podem incluir ruído de carros (*car noise*), ruído de pessoas (*people noise*) ou baixa largura de banda (*low bandwidth*), que

Figura 4 – *Rotulagem de erros*

Example	Label	Prediction	Car noise	People noise	Low bandwidth
1	“Stir fried lettuce recipe”	“Stir fry lettuce recipe”	1		
2	“Sweetened coffee”	“Swedish coffee”		1	1
3	“Sail away song”	“Sell away some”		1	
4	“Let’s catch up”	“Let’s ketchup”	1	1	1

Fonte:Ng (2022b)

pode representar perda na taxa de dados da comunicação. Essa abordagem permite identificar padrões nos erros e orientar as melhorias no modelo de reconhecimento de voz.

A utilização desses rótulos tem como objetivo mapear as causas dos erros do algoritmo e facilitar a resolução dos mesmos. É importante ressaltar que um mesmo erro pode envolver mais de um tipo de ruído. A segmentação em categorias e a análise de cada caso permite definir prioridades, levando em consideração a frequência em que um determinado tipo de ruído ocorre.

Por exemplo, se em todo o conjunto de dados, a *tag car noise* representar apenas 2% do total, sua relevância é menor em comparação com a *tag people noise*, que representa 80%. Esse processo é iterativo e pode ocorrer várias vezes ao longo do desenvolvimento do sistema. À medida que a análise avança, novos rótulos podem ser adicionados para melhorar a compreensão e o tratamento dos erros identificados.

Para parametrizar essa análise, é essencial questionar alguns cenários, como quanto espaço para melhoria existe, com que frequência essa categoria de rótulo aparece, o quão fácil é melhorar a precisão nessa categoria e o quão importante é melhorar essa categoria. Esses questionamentos são cruciais para orientar a tomada de decisão.

Ao selecionar um rótulo para trabalhar na melhoria, é recomendado coletar mais dados associados a essa categoria específica. Por exemplo, se ocorrerem muitas falhas devido a ruídos de carros, é desejável obter mais dados de áudio com ruídos de carros. No campo de ML, sempre é desejável possuir mais dados, mas em muitos casos, pode ser custoso obtê-los. No entanto, ao adotar essas práticas, os dados desejados se tornam mais seletivos, permitindo que sejam coletados em menor volume, focando na categoria em que se deseja aumentar a precisão. A classificação das categorias de rótulos fornece uma orientação clara para concentrar o aumento de entrada de dados nos campos que terão o maior impacto.

Nas últimas décadas, muito tem se discutido por estudiosos da área de como melhorar o código, de fato, muitas pesquisas de foram desenvolvidas realizando *downloads* de *datasets* existentes, e trabalhando na escolha de um modelo que funcione bem neles.

No entanto, para muitas aplicações, há a possibilidade de alterar os dados. Existem inúmeros projetos em que o algoritmo ou o modelo são, essencialmente, um componente;

um modelo pronto e genérico muitas vezes funcionará suficientemente bem. Nesses casos, é mais eficaz dedicar mais tempo ao aprimoramento dos dados, pois frequentemente precisarão ser personalizados para cada modelo de negócio e cenário específico.

Nesta etapa, ocorre a seleção do algoritmo de aprendizado de máquina que melhor se adapta às necessidades do projeto, juntamente com o processo de treinamento desse algoritmo. Além disso, é decidido se mais dados precisam ser inseridos no sistema, o que pode exigir um retorno à etapa anterior de preparação de dados. Esse ciclo iterativo entre dados e modelagem exerce uma influência significativa no resultado final do sistema a ser desenvolvido.

Esse conjunto de processos executados nesta etapa também é conhecido como *experimentação*. É semelhante a um processo de tentativa e erro, envolvendo várias combinações de algoritmos, recursos e *hiperparâmetros* diferentes até encontrar a combinação que resulte em um modelo adequado para o propósito do trabalho.

Nesse contexto, as tentativas que resultam em erros são extremamente importantes, pois a análise desses erros leva à definição de novos conjuntos de combinações a serem testados. Surge, então, um novo conjunto de necessidade para MLOps, rastrear as métricas, do inglês *track metrics*, das execuções do experimento, que representa, registrar as informações em cada teste, e mapear métricas baseado na ideia de ação e reação.

Estas tarefas, desde a experimentação, até a otimização estão fundamentadas no *pipeline* de construção do sistema como parte do MLOps. Uma vez que o cientista de dados cria o modelo com uma eficácia aceitável, o modelo precisa ser implantado em produção.

2.1.2.4 Implantação

A etapa de implantação em produção, além de ser fundamental para o processo de amadurecimento do projeto, é também a fase em que surgem diversos desafios, que podem ser divididos em duas categorias. A primeira está associada ao surgimento de problemas estatísticos. Na segunda, problemas associados ao software e ferramentas utilizadas para desenvolver o projeto.

Nesta etapa, é possível analisar de forma empírica se existe a necessidade de realizar mais ajustes em etapas anteriores do ciclo de vida. O mais importante é compreender como os dados variam com o tempo, e dentro desta variação, podem existir dois cenários. O primeiro, remete a variação gradual, do inglês *gradual change*, que representa uma mudança suave ao longo de um período. Pode-se fazer uma analogia com a variação que um idioma sofre ao longo do tempo, com a adição de termos, e ajustes linguísticos.

O segundo, caracteriza uma mudança repentina, do inglês *Sudden Shock*, que representa uma variação súbita em um curto período. Pode-se fazer uma analogia com uma pessoa anônima, que viraliza na internet, e passa a se tornar famosa muito rapidamente.

Outra analogia é o incidente da pandemia do Covid-19. Muitas pessoas começaram a fazer mais compras online, utilizando muito mais o cartão de crédito, caracterizando uma mudança repentina em pessoas que não o utilizavam com muita frequência. Desta forma, alguns sistemas antifraude baseados em ML realizaram alertas de fraude de forma equivocada, pois estavam em produção, e em seu desenvolvimento, a pandemia não havia sido um parâmetro de análise.

A presença destes problemas estatísticos de variação nos dados é melhor compreendida através dos conceitos de "desvio de dados"e "desvio de conceito".

O termo "desvio de conceito", em inglês *concept drift*, refere-se ao mapeamento desejado de um dado **X** para **Y**. Analogamente, pode-se definir que **X** seja o tamanho de uma determinada residência e **Y** seja o seu preço. Com a presença da inflação e outros indicadores econômicos, o preço da casa (**Y**) sofrerá variação, mas seu tamanho (**X**) será constante.

Um exemplo adicional para consolidar o conceito de *concept drift*, seria o comportamento de compra de clientes ao longo do tempo, influenciado pela força da economia de seu país ou região. Nesse caso, o poder aquisitivo não é especificado nos dados, e sua variação pode impactar na frequência de compra do cliente. Esses elementos são definidos como *contexto oculto*, do inglês *hidden context*.

O conceito de *desvio de dados*, do inglês *data drift*, é usado para descrever a variação na distribuição dos dados de entrada. No exemplo, com o mapeamento em que **X** seja o tamanho de uma determinada residência, e **Y** seja o seu preço, um cenário de *data drift*, pode ser considerado quando o tamanho das casas (**X**), de fato passe a variar, porém, o preço (**Y**) se mantém constante.

É importante prever e gerenciar essas duas variações, e como auxílio, existe um *checklist* a se fazer para melhor modelar a estrutura do sistema de predição em produção:

Tempo real ou não: Extremamente importante parametrizar se a predição será feita na em tempo real ou se será executada através de uma coleta de dados durante um período, e após isso, elaborado um relatório preditivo pelo sistema.

Cloud VS Edge: A escolha entre executar o projeto em nuvem ou em servidores locais (*edge*) é fundamental para determinar a infraestrutura necessária e definir as ferramentas de suporte a serem utilizadas, bem como suas configurações. A implementação da metodologia de Continuous Integration/Continuous Delivery (CI/CD), também varia de acordo com cada cenário, assim como a parametrização de indicadores como a latência na comunicação. Ao lidar com a análise de dados de diversas naturezas, é essencial estruturar a manipulação e segurança desses dados de forma coerente em cada ambiente. As melhores práticas para nuvem e servidores locais diferem significativamente, exigindo uma especificação detalhada do cenário para garantir uma abordagem adequada.

Recursos da hospedagem do sistema: Definir e monitorar os recursos de infraestrutura do sistema são aspectos críticos, pois o desempenho e a precisão das análises preditivas dependem diretamente deles. Três parâmetros fundamentais nesse sentido são o *armazenamento permanente*, o *armazenamento temporário* e a *capacidade de processamento*.

É essencial determinar a quantidade de armazenamento em disco para o armazenamento permanente, a capacidade de memória RAM para o armazenamento temporário e a potência de processamento necessária para garantir que o sistema possa executar suas tarefas sem enfrentar gargalos. Esses aspectos devem ser monitorados e especificados com base na carga do sistema operacional, utilizando métricas como a média de carga (*load average*), para assegurar um funcionamento adequado do sistema.

Logging: A implementação de uma estrutura de *logs* em um projeto de aprendizado de máquina é crucial para o MLOps, pois permite identificar problemas e determinar se o algoritmo precisa ser reajustado ou retreinado. Essa estrutura segue uma abordagem semelhante à de sistemas de *logs* estruturados em software convencional, usados para detecção de erros. No entanto, no contexto do MLOps, as informações registradas nesses são essenciais para a reestruturação do sistema e para a tomada de decisões sobre o retreinamento do modelo em produção, conforme proposto nos princípios do ciclo de vida em produção.

Segurança e privacidade: Com o surgimento da Lei Geral de Proteção de Dados Pessoais (LGPD) no Brasil, e a manifestação de outras regulamentações de dados no mundo digital, a sensibilidade do armazenamento e a segurança, tornou-se um parâmetro crucial na especificação de projetos, principalmente do ponto de vista jurídico. Com isso, como um sistema de ML é baseado em dados, sejam eles de qualquer natureza, este ponto torna-se ainda mais crítico, necessitando de uma validação cuidadosa e detalhada (NG, 2022a).

Desta forma, a implantação de um sistema de ML ocorre em dois processos, mover o projeto do ambiente de desenvolvimento, para a produção, onde ele atuará de fato, e o processo de monitoramento, com a manutenção do sistema, já em produção.

Este último está diretamente associada à análise de desempenho do sistema de forma automatizada e com o processo de identificação de anomalias na ingestão de dados no sistema. É através deste monitoramento, que conforme a Figura 2, ocorre o retreinamento, representados por setas fluindo da etapa de 2.1.2.4 para a etapa de 2.1.2.3, que por sua vez, flui para 2.1.2.2.

Pode ocorrer também, o fluxo da etapa de 2.1.2.4 diretamente para 2.1.2.2, definindo qual dinâmica o processo vai executar, é o tipo de conclusão que o monitoramento do sistema vai chegar baseado na análise de *logs*.

Após implantar o sistema desenvolvido em produção, é importante definir alguma proposta de validação do algoritmo de predição, antes de colocá-lo para tomar decisões em cenário real.

Para isso, existem algumas estratégias que podem mitigar riscos de modelos em produção estarem inadequados. Uma delas é chamada de *shadow mode*, ou modo sombra em tradução literal, que consiste em utilizar de algum julgamento externo para validar conclusões do algoritmo. Esta validação é executada em paralelo ao algoritmo, e comparada baseada nas decisões. O instrumento de comparação pode ser um julgamento humano por uma inspeção visual do cenário.

Definir o grau apropriado de automação do *pipeline* de ML é também importante, nem sempre ser totalmente automatizado, caracteriza o cenário ideal para a aplicação. A IA por exemplo, pode encaminhar informações para um humano decidir, quando ela identificar que não possui um decisão com uma acurácia satisfatória. Neste cenário, existe um trabalho a ser realizado para o próprio sistema identificar estes cenários, e tomar a decisão correta.

Para isso, é importante definir o monitoramento do sistema, uma das melhores formas de realizar isso, é monitorando *dashboards*, elaborados pelo próprio sistema indicando parâmetros importantes para as tomadas de decisão, ou que definem se o projeto esta sendo executado de forma adequada.

Estes *dashboards* podem realizar comparações como carga do servidor, ou uso de recursos do ambiente, para validar a questão da infraestrutura, que pode caracterizar algum tipo de erro no software.

Para identificar possíveis problemas nas métricas de entrada e de saída, análises como comparação entre frações de valores nulos ou coeficiente de perda de dados na saída, em comparação com a entrada, pode caracterizar falhas no algoritmo.

Após definir as métricas para analisar, é necessário definir alerta para estes limiares. Ao definir se o problema é proveniente do ambiente em que o sistema está hospedado, como carga do servidor, ou uso de memória, é necessário refatorar o software, porém no caso do problema ocorrer no algoritmo, será necessário retreiná-lo, podendo ser de forma manual ou automática.

Com isso, a complexidade e importância da etapa de implantação, se mostra evidente, tornando indispensável a execução de boas práticas e ferramentas de apoio.

2.2 Ferramentas para implantação

Para implementar os conceitos de MLOps neste trabalho, serão utilizadas diversas ferramentas de apoio, cada uma associada a uma camada da estrutura definida do projeto.

2.2.1 Docker

A necessidade de ciclos de desenvolvimento cada vez mais curtos e entrega contínua cada vez mais ágil, o uso de contêineres se mostra cada vez mais convidativo em infraestruturas de sistemas de software, por serem mais flexíveis que as máquinas virtuais e fornecem desempenho quase nativo. (COMBE; MARTIN; PIETRO, 2016).

O desenvolvimento e as operações que incorporam o conceito de Integração Contínua e Entrega Contínua CI/CD, atendem diretamente esta necessidade de agilidade na entrega de atualizações no software (GARG et al., 2021).

a Integração Contínua Ajuda na gestão do tempo e permite curtos ciclos de atualização para melhorar a qualidade do software e aumentar a produtividade geral da equipe. A implantação Contínua ajuda na automação da implantação do software em produção e realizar verificações de qualidade através de monitoramento.

Empregar *pipelines* de CI/CD em uma aplicação que incorpora componentes de MLOps resulta em grandes desafios. Porém, incorporar este conceito para implantação automatizada de modelos de IA resulta em agilidade, escalabilidade, modularidade e consequentemente em grande agregação de valor ao projeto (GARG et al., 2021).

Conforme ilustra a Figura 2, no ciclo de vida do aprendizado de máquina existe um processo de reutilização dos componentes, combinações distintas de compartilhamento de informações entre eles.

A conteinerização do *pipeline* do sistema de aprendizado de máquina, o torna modular, escalável e facilita a personalização independente de cada componente do ciclo, sem comprometer o compartilhamento de recursos e desempenho.

Como existe o envolvimento de múltiplos *containers*, é convidativa a utilização de algum recurso para gerenciá-los, desta forma, o Docker se apresenta amplamente utilizado para a administração deles, permitindo que o desenvolvimento utilize recursos de automação de contêiner, implantação, e redes de forma otimizada.

O Docker implementa uma camada de virtualização nos *containers* basicamente fazendo o papel de uma Virtual Machine (VM), porém muito mais muito leve, e utilizando menos recursos. Ele implementa uma interface de comunicação entre o container e o *host* em que o sistema está hospedado, controlando o uso de seus recursos, como processamento, *RAM*, *bitrate*, disponibilizando todas as bibliotecas necessárias.

Cada container esta associado a uma imagem Docker, que representa basicamente um modelo de sistema de arquivos, um container específico, gerado a partir desta imagem. Uma subcamada virtualizada é criada de processos abaixo do Docker, que está associado ao processo inicial que iniciou um determinado container.

2.2.2 Nexus

O Nexus Repository Manager, comumente conhecido como Nexus, atua como um repositório centralizado que armazena e organiza artefatos, sendo essencialmente utilizado para armazenar imagens de contêiner Docker. Isso simplifica a gestão e distribuição dessas imagens, fornecendo uma fonte única e confiável para acessá-las(INC., 2022).

Além de armazenamento, o Nexus facilita o gerenciamento de dependências ao permitir a especificação de versões específicas de imagens Docker. Isso contribui para a consistência e reprodutibilidade em diferentes ambientes de desenvolvimento e implementação.

Um aspecto importante é o controle de acesso e segurança oferecido pelo Nexus. Ele implementa recursos robustos para garantir que apenas usuários autorizados possam acessar e modificar as imagens armazenadas, o que é crucial para manter a integridade e a segurança do ambiente.

Ao configurar o Nexus para armazenar imagens de contêiner Docker, é comum criar um repositório Docker Hosted no Nexus. Esse repositório age como o ponto central para armazenar as imagens, e o Docker pode ser configurado para interagir com esse repositório para operações como *push* e *pull* de imagens. Vale ressaltar que os detalhes específicos da configuração podem variar com base na versão do Nexus utilizada, sendo recomendável consultar a documentação oficial para orientações precisas.

2.2.3 TensorFlow

Para implementação dos conceitos de MLOps, será utilizado o TensorFlow (TF). Como este trabalho possui objetivo de implantar um sistema de ML em produção, a relevância de utilizar uma ferramenta de auxílio para gestão dos componentes se mostra indispensável.

O TF foi desenvolvido pela Google, sendo uma plataforma robusta que facilita a criação e a implantação de modelos de ML. Ele estabelece um *pipeline* específico para tarefas de aprendizado de máquina de alto desempenho em produção através de componentes que são construídos através de bibliotecas customizáveis, para melhor atender cada cenário.

Seu uso será direcionado para integrar os componentes do ciclo de vida de um sistema de ML, discutidos no capítulo 2.1.2, em uma plataforma única. O objetivo é aumentar a padronização dos componentes envolvidos, simplificar a configuração, realizar experimentos de forma mais rápida e reduzir o tempo de operação em produção. Isso fornecerá estabilidade e minimizará interrupções nos processos (GOOGLE, 2024a).

2.2.4 Fast API

Para uma interface e simplificar o gerenciamento do acesso aos resultados do sistema de detecção de anomalias desenvolvido, será utilizada a ferramenta Fast API.

É um *framework web* utilizado para construção de Application Programming Interface (API), com características de alto desempenho. Sua proposta é ser simples e com característica de viabilizar o desenvolvimento rápido de funcionalidades, muito adequada para o cenário que este trabalho propõe.

Seu desempenho é semelhante a de ferramentas conceituadas no mercado como *NodeJS* e *GO* conforme a documentação oficial (RAMíREZ, 2022), porém com caráter mais simplista.

Segundo o trabalho (BANSAL; OUDA, 2022), foi realizada uma avaliação em resultados experimentais que mostraram que o desempenho do modelo de ML utilizando FastAPI foi melhorado em quase 45% em comparação ao Flask, outra ferramenta com proposito semelhante muito utilizada no mercado.

2.2.5 Streamlit

O Streamlit é uma biblioteca em Python utilizada para o desenvolvimento de interfaces *web* interativas e dinâmicas de forma simplificada. Projetado para ser fácil de usar, o Streamlit permite que desenvolvedores criem aplicativos interativos com poucas linhas de código, utilizando principalmente Python.

A biblioteca é especialmente adequada para a visualização de dados e análise exploratória, proporcionando uma experiência fluida e ágil na criação de *dashboards* interativos. Sua sintaxe minimalista e intuitiva reduz significativamente a complexidade de implementação, permitindo que os desenvolvedores foquem mais na lógica da aplicação e nas análises visuais.

No contexto acadêmico, de forma eficiente ele apresenta resultados de experimentos, análises de dados, ou demonstra modelos e algoritmos de forma desobstruído. Ao integrar facilmente elementos como gráficos, tabelas e controles interativos, ele proporciona uma plataforma eficaz para comunicar descobertas e *insights* de maneira intuitiva (SNOWFLAKE, 2024).

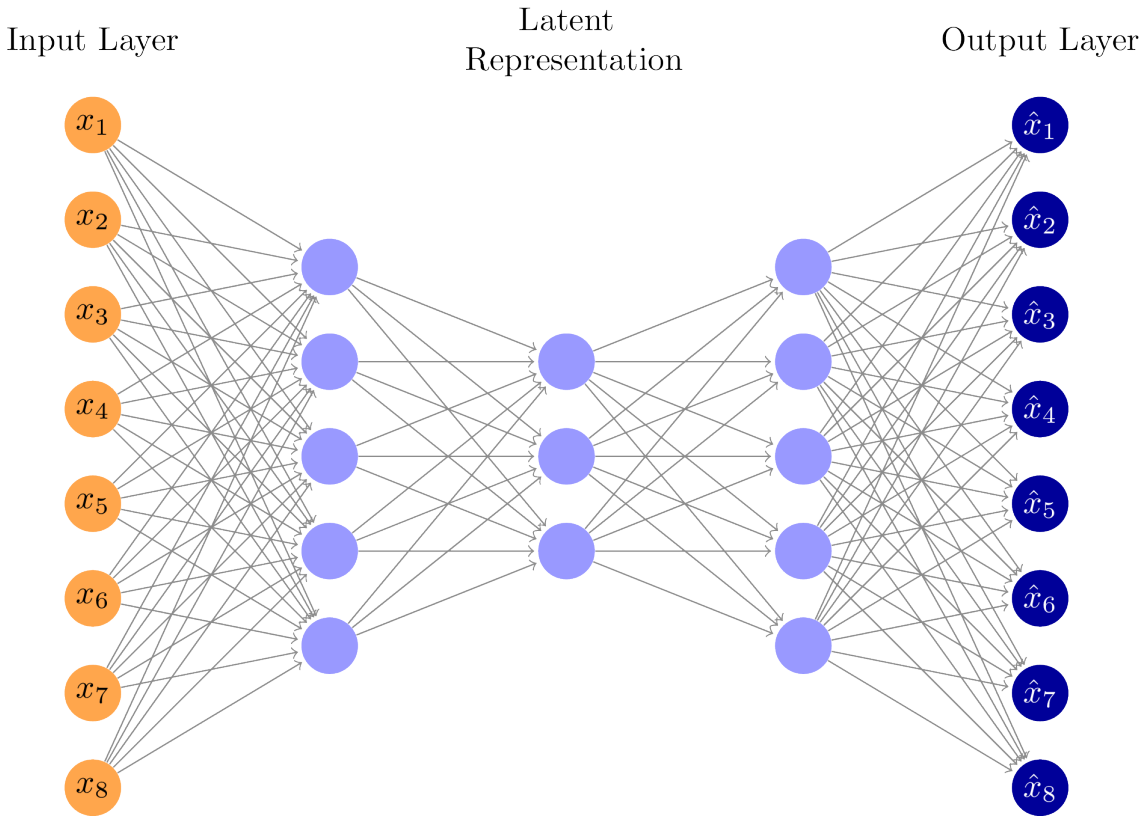
2.3 Detecção de anomalias

Em ML, em vez serem criadas regras em uma linguagem de programação, como ocorre no desenvolvimento de *software* tradicional, existe uma máquina mapeando as respostas para os dados de entrada, desta forma, a máquina descobre as regras.

Se obtiver todos os dados de entrada corretamente rotulados, basta treinar o modelo, para detectar variações.

A Figura 5, ilustra o princípio de funcionamento de um esquema de *auto-encoder*.

Figura 5 – *Auto-Encoder*



Fonte:Riebesell (2022)

Basicamente pode ser dividido em duas redes neurais, em camadas de codificação e decodificação dos dados, da esquerda para a direita respectivamente.

Os dados de entrada, possuem um alto nível de dimensionalidade, o processo de codificação, deve aprender uma maneira de representar os dados em uma dimensão muito menor, fluindo da esquerda para a direita, para compactar os dados na camada de entrada. Este processo deve ser encerrado assim que encontrado o menor nível de representação latente das informações originais.

Após isso, a rede deve aprender como reconstruir os dados a partir da menor representação latente encontrada, no processo de decodificação, na mesma dimensionalidade dos dados originais, perdendo o mínimo de informação.

O objetivo é construir um modelo de aprendizado de máquina, que aprenda a representação dos dados de entrada, para utilizar então este modelo, em dados futuros que não se parecem com os dados de entrada do treinamento, realizando assim a detecção de anomalias.

3 DESENVOLVIMENTO

Para o desenvolvimento deste trabalho, será utilizado um exemplo pronto utilizando um *dataset* previamente rotulado e ajustado, caracterizando-se por ser relativamente simples e não possuir uma estrutura de escalabilidade ou infraestrutura preparada para ambientes de produção. A proposta é transformar esse exemplo inicial em um ambiente robusto e escalável, aplicando os princípios de MLOps para desenvolver uma infraestrutura que permita a execução eficiente do sistema em cenários reais.

Importante ressaltar que, neste contexto, o foco principal é fornecer uma visão prática e introdutória na aplicação de conceitos de MLOps. A abordagem prioriza a criação de dois *pipelines* sendo responsável pelo treinamento de modelos e o outro pelas inferências dos dados novos, juntamente com a integração do consumo de dados em um fluxo contínuo.

Destaca-se que, devido ao escopo deste trabalho, não será aplicada uma implementação completa de MLOps destacados em 2.1. A ênfase está em proporcionar uma estrutura básica e inicial, servindo como ponto de partida para compreensão e aplicação de conceitos fundamentais no contexto de aprendizado de máquina em produção.

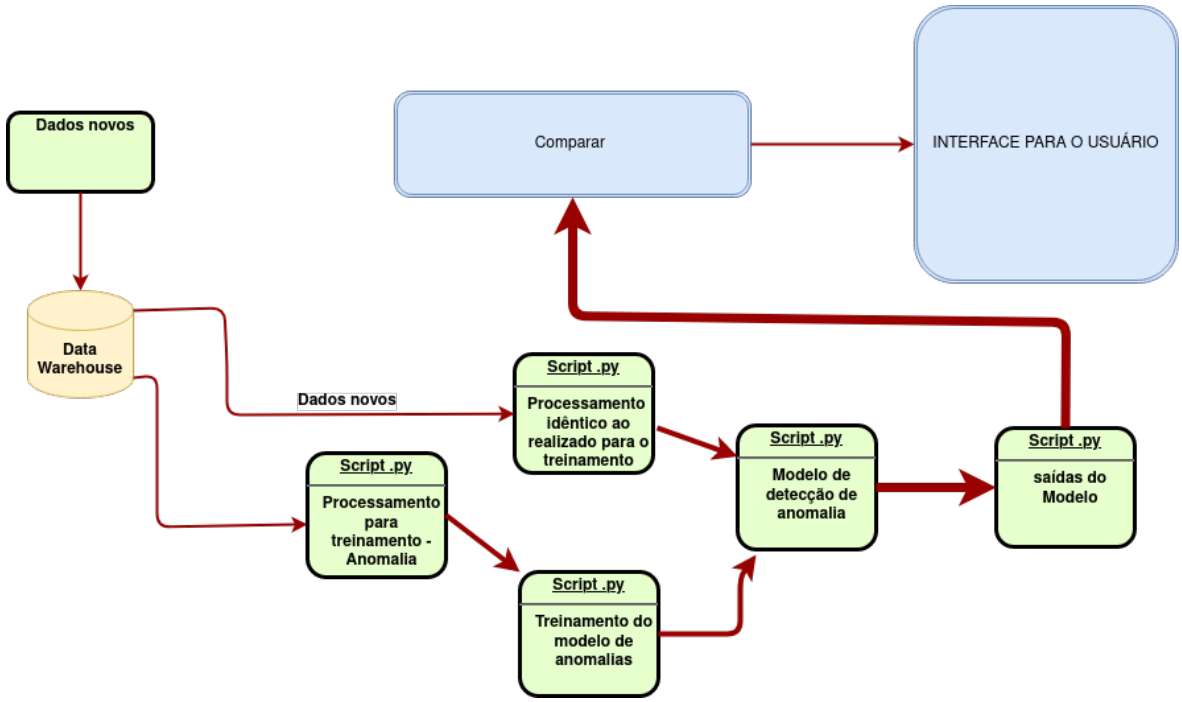
O exemplo utilizado como base para este protótipo é disponibilizado pelo TensorFlow. Mais especificamente, o exemplo é proveniente de um Jupyter Notebook acessível por meio deste link[1]. Esse material é uma introdução ao uso de *autoencoders* com TensorFlow (GOOGLE, 2024b).

A Figura 6 ilustra um escopo da arquitetura do sistema de ML deste trabalho. Um banco de dados denominado *Datawarehouse* armazena um conjunto de dados de Eletrocardiograma (ECG). A arquitetura foi projetada para suportar dois fluxos de dados distintos. O fluxo superior representa a entrada de novos dados de ECG no sistema, que passam por uma série de *scripts* para realizar um processamento pré-definido. O fluxo inferior, por sua vez, representa dados já separados e rotulados como normais ou anômalos em uma tabela específica, que são usados para treinar um novo modelo de ML. Ambos os fluxos de dados passam por processamentos semelhantes. No final de cada fluxo, os resultados de uma inferência são apresentados por meio de uma interface.

A infraestrutura proposta para o desenvolvimento deste trabalho, ilustrada na Figura 7, que representa uma evolução da arquitetura delineada na Figura 6, utilizando o Docker como ferramenta de apoio para sua construção. A arquitetura original resolve os problemas de análise de dados e algoritmo de ML, mas ao ser implantada em um ambiente de produção, mostra-se inflexível e difícil de escalar. Isso ocorre porque a arquitetura

[1] https://www.tensorflow.org/tutorials/generative/autoencoder

Figura 6 – *Arquitetura do sistema de ML*



Fonte: O próprio autor.

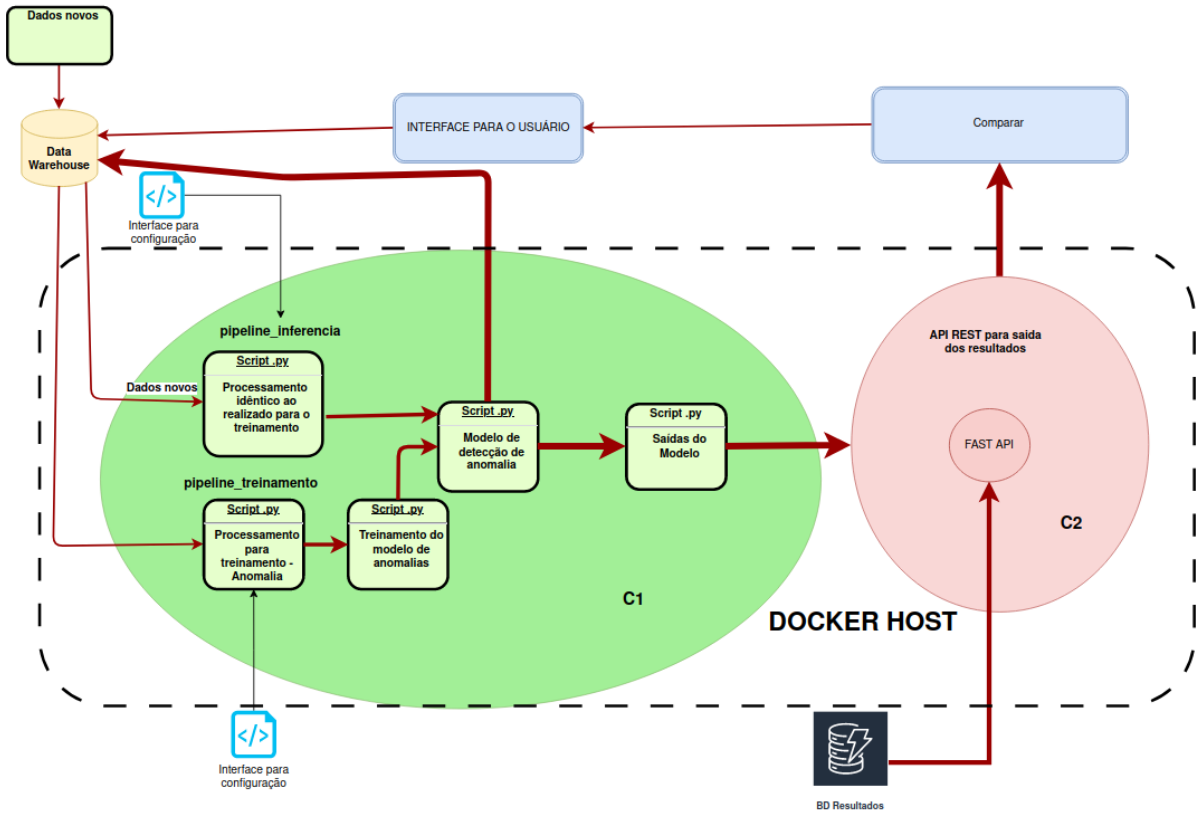
original não foi projetada para lidar com as demandas do ambiente de produção, onde a flexibilidade e a capacidade de escalar são essenciais.

Os pipelines foram dockerizados em um contêiner chamado *C1*, como ilustra a Figura 7. Esses *pipelines* são responsáveis pelo treinamento de novos modelos e pela inferência de novos dados, utilizando a plataforma *TensorFlow*, cujos benefícios estão destacados na seção 2.2.3. Os resultados das inferências são inseridos em outra base de dados por meio de uma API em outro contêiner. Essa abordagem de uso de contêineres para encapsular e padronizar os *pipelines* os torna portáteis e eficientes para implantação em diferentes ambientes de desenvolvimento. A utilização do Docker neste contexto garante a consistência e reprodutibilidade das operações de desenvolvimento e teste, contribuindo para a integridade e confiabilidade do sistema de ML. Isso permite validar diferentes estruturas de *pipelines* sem modificar outros serviços.

O segundo contêiner, ilustrado como *C2* na Figura 7, contém uma API desenvolvida em FastAPI. Ela recebe os resultados da inferência realizada no contêiner de *pipelines*, e os insere na base de resultados. Essa base de dados retém informações cruciais, como os dados originais do ECG que foram submetidos à inferência e as predições realizadas pela reconstrução do modelo. Todos os componentes dockerizados dessa infraestrutura proposta, desfrutam dos benefícios abordados na seção 2.2.1.

Uma interface é disponibilizada para analisar os dados originais do ECG e as predições geradas pelo sistema de ML. Após a análise, o usuário pode validar se concorda

Figura 7 – *Infraestrutura do sistema de detecção de anomalias*



Fonte: O próprio autor.

ou não com a decisão do sistema sobre o eletrocardiograma ser anômalo. Esse *feedback* do usuário é armazenado na base de dados principal, no Datawarehouse, na tabela destinada para treinamentos, já rotulada.

Quando necessário, é possível realizar novos treinamentos utilizando os dados armazenados no Datawarehouse. Isso permite gerar novos modelos com base em mais informações, aprimorando a capacidade do sistema de tomar decisões mais precisas sobre a detecção de anomalias em eletrocardiogramas.

Cabe mencionar que as bases de dados não são dockerizadas, o que significa que estão fora dos contêineres Docker, indicando uma arquitetura mais flexível e separação de preocupações entre a lógica da aplicação e o armazenamento persistente dos dados.

3.1 Base de dados

O armazenamento de dados neste trabalho, é uma das principais atividades desenvolvidas. Tendo isso como parâmetro, a escolha de um sistema de armazenamento adequado se faz essencial.

O MySQL é um Sistema Gerenciador de Banco de Dados (SGBD) que utiliza a

linguagem Structured Query Language (SQL) como interface. Ele é uma escolha popular para gerenciamento de banco de dados, mas sua aplicação específica em cenários de armazenamento de ECG e modelos de aprendizado de máquina possui nuances a serem consideradas.

Em geral possui desempenho eficiente e se mostra adequado tanto para as consultas associadas a inferência dos eletrocardiogramas, quanto para consultas para obter os modelos. Isso é fundamental para a recuperação rápida de registros, especialmente em ambientes que exigem respostas em tempo real.

O suporte a transações Atomicidade, Consistência, Isolamento e Durabilidade (ACID) garante a integridade dos dados, crucial ao lidar com informações críticas, como registros de ECG e modelos de aprendizado de máquina.

Também a grande comunidade de usuários do MySQL e a vasta documentação disponível facilitam o desenvolvimento, a manutenção e a solução de problemas relacionados ao banco de dados, isso se torna essencial ao longo do tempo, pois possíveis alterações na modelagem das tabelas podem desencadear erros.

O MySQL pode encontrar limitações em termos de escalabilidade horizontal quando comparado a algumas soluções Not Only Structured Query Language (NoSQL). Isso pode ser uma desvantagem em cenários de grande volume de dados, como armazenamento massivo de ECG.

Se o modelo de ML lida com dados não-estruturados, como imagens de eletrocardiogramas em formato bruto, pode haver uma complexidade adicional na modelagem dos dados.

Em ambientes de alta concorrência, o MySQL pode enfrentar problemas de bloqueio que afetam o desempenho. Isso pode ser crítico em sistemas onde a atualização frequente de modelos é necessária.

Além da escolha do gerenciador da base de dados, a decisão entre dockerizar ou não representa uma etapa significativa. Dockerizar proporciona diversos benefícios, tais como isolamento de ambientes, consistência na execução em diferentes plataformas e facilidade na distribuição.

A capacidade de escalar horizontalmente, adicionando ou removendo contêineres, é uma vantagem significativa em cenários de grande carga. Isso otimiza o uso de recursos e melhora o desempenho.

No entanto, a não dockerização também tem seus méritos, especialmente em termos de simplicidade e praticidade. Evitar a dockerização pode simplificar a configuração e manutenção do ambiente, reduzindo a complexidade associada à gestão de contêineres. Esta escolha pode ser motivada por fatores como a familiaridade com ambientes tradicionais

de execução ou a minimização em termos de recursos. A execução de contêineres adiciona uma camada de virtualização, introduzindo um pequeno *overhead* de recursos. Em sistemas de pequena escala, isso pode ser considerado desnecessário.

A decisão final entre dockerizar ou não deve ser cuidadosamente ponderada, considerando as demandas específicas do projeto, as habilidades da equipe e as metas de desempenho. Ambas as abordagens têm suas vantagens e desvantagens, e a escolha ideal dependerá do contexto e dos requisitos particulares do sistema em desenvolvimento.

Por se tratar de uma infraestrutura menor, com menos necessidade de escalabilidade dinâmica em relação ao *datwarehouse*, este trabalho não terá este componente do sistema dockerizado. Essa escolha pode simplificar a configuração e manutenção, especialmente em projetos de menor escala e em fases iniciais de desenvolvimento.

3.1.1 Armazenamento de ECGs

Para este trabalho, foram propostas bases de dados destinados a armazenar informações relacionadas a eletrocardiogramas ECG, são elas a base que armazena os dados utilizados pelos *pipelines* de treinamento e inferência, e a base para registros da saída dos *pipelines*. Cada registro na base é composto por cento e quarenta registros, que representam um segmento específico normalizado, juntamente com a data de registro diária para proporcionar uma identificação temporal.

Duas tabelas estão sendo utilizadas na base dos *pipelines*, uma para o treinamento de modelos, e a outra para armazenar dados novos, vindos de coletas de ECG utilizados para predizer se um novo eletrocardiograma, que não possui um rótulo predefinido normal ou anômalo.

No entanto, é importante notar que neste projeto, a alimentação automática de dados nesta tabela não está sendo implementada. Os dados novos armazenados nela são provenientes de análises de equipamentos que medem ECG, cujo processo pode ser realizado de várias maneiras, específicas para cada caso. Portanto, o projeto foca na inferência de dados novos, sem a responsabilidade de adquirir os dados, uma tarefa que é delegada a processos fora da infraestrutura projetada.

Como este trabalho parte de um exemplo pronto, utilizando um *dataset* de ECG já existente, na tabela de treinamento já estão armazenados cinco mil exemplares de eletrocardiogramas. Em cada um, há um campo reservado para a rotulagem dos registros, indicando se o eletrocardiograma é classificado como normal ou anômalo. Este campo classifica o ECG como normal (1) ou anômalo (0), estabelecendo uma referência para o treinamento e avaliação dos modelos.

Duas tabelas também são utilizadas pela base associada a saída dos resultados do sistema de ML, uma para armazenar o eletrocardiograma que passou pelo processo de

inferência, e a outra para armazenar as predições realizadas pelo modelo.

Inicialmente, o trabalho teve como ponto de partida um conjunto de dados previamente rotulado na tabela de treinamento, que inclui eletrocardiogramas anômalos e normais. Esse conjunto serve como base para o treinamento inicial dos modelos, possibilitando o sistema aprender a reconhecer padrões distintos.

Dados dessa base de treinamento foram selecionados aleatoriamente para passar pelo processo de inferência, com o intuito de validar a infraestrutura proposta que foi concebida para suportar inferências em novos dados, possibilitando uma avaliação contínua sobre se um ECG recém registrado é considerado normal ou anômalo.

3.1.2 Armazenamento de Modelos

Foi desenvolvida uma tabela dedicada para o armazenamento de modelos, projetada para fornecer informações detalhadas e abrangentes sobre cada modelo registrado.

Cada modelo é identificado por uma etiqueta (tag), que permite uma diferenciação entre eles. Essas etiquetas, juntamente com um identificador único padrão, compõem a chave primária da tabela.

No contexto de ML, a escolha e definição dos campos de registros que devem compor a tabela são importantes para avaliar a eficácia do modelo em identificar padrões anômalos e normais. Para este trabalho, alguns campos foram selecionados como mais relevantes.

(Valor Máximo): Este campo do tipo *float* armazena o valor máximo alcançado durante o treinamento do modelo. Em um contexto de detecção de anomalias, esse valor pode representar uma referência importante para avaliar a capacidade do modelo em lidar com variações extremas.

(Valor Mínimo): Analogamente ao campo anterior, o valor mínimo, do tipo *float*, indica a menor magnitude alcançada durante o treinamento. Isso proporciona *insights* sobre a sensibilidade do modelo a padrões específicos.

(Limiar): Representando um limiar predefinido, o campo *threshold*, do tipo *float*, é crucial para determinar se uma predição é considerada anômala ou normal. É uma medida ajustável que influencia diretamente a precisão do modelo.

(Acurácia): A acurácia, em formato *float*, expressa a medida de quão preciso é o modelo em suas predições. Indica a proporção de predições corretas em relação ao total de predições.

(Precisão): A precisão, também do tipo *float*, é uma métrica que avalia a proporção de verdadeiros positivos entre todas as instâncias identificadas como positivas pelo modelo. Em um contexto de detecção de anomalias, isso representa a capacidade do modelo em

não rotular erroneamente instâncias normais como anômalas.

(Recall): O campo *recall*, em formato *float*, refere-se à proporção de verdadeiros positivos entre todas as instâncias que são, de fato, positivas. Em outras palavras, representa a capacidade do modelo em identificar corretamente instâncias anômalas.

(Pesos do Modelo): Esse campo do tipo *blob* armazena os pesos do modelo, que representam os parâmetros ajustáveis aprendidos durante o treinamento. Os pesos são fundamentais para a inferência, pois definem como o modelo interpreta e responde aos dados de entrada.

Essa estrutura abrangente da tabela de modelos proporciona um ambiente rico em informações, permitindo uma análise aprofundada do desempenho de cada modelo, suas características distintivas e a possibilidade de ajustes contínuos para otimização do sistema de detecção de anomalias.

Armazenar modelos no banco de dados oferece vantagens notáveis em termos de centralização, controle de acesso, transações atômicas e backup consistente. Ao centralizar todos os dados relacionados ao projeto em um local único, a gestão e manutenção tornam-se mais convenientes. A capacidade de realizar transações atômicas assegura a integridade do modelo durante operações de atualização. Além disso, o controle de acesso permite gerenciar permissões de forma eficaz, contribuindo para a segurança dos dados. Durante operações de backup, o modelo é incluído, garantindo que a versão mais recente seja salva juntamente com outros dados.

No entanto, essa abordagem apresenta desvantagens, como o possível *overhead* do banco de dados, especialmente em termos de desempenho, e limitações relacionadas ao tamanho do modelo. Operações de leitura e gravação podem ser mais lentas, e pode haver dificuldades em atualizações parciais do modelo.

Por outro lado, o armazenamento em sistema de arquivos oferece vantagens em termos de facilidade de atualização, melhor desempenho em leitura e escrita, flexibilidade de backup e escalabilidade simples.

A substituição ou atualização de partes específicas do modelo torna-se mais fácil, e as operações de leitura e gravação são potencialmente mais rápidas. Backup e restauração podem ser realizados de maneira flexível, e a escalabilidade é facilitada, especialmente em sistemas de armazenamento distribuído.

A escolha entre essas abordagens dependerá das necessidades específicas do projeto, considerando fatores como tamanho do modelo, operações de atualização, desempenho e requisitos de gerenciamento de dados. Neste trabalho foi escolhido o armazenamento no banco de dados.

3.2 Pipelines

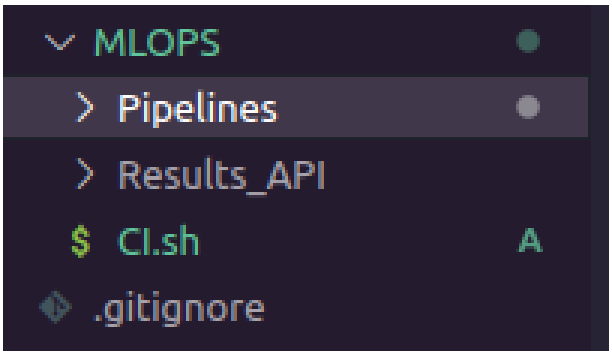
Nos *pipelines*, a estrutura é projetada com um enfoque escalável e modular, incorporando princípios de Programação Orientada a Objetos (POO), padrões de projeto e uma abordagem eficiente para gerenciamento de *logs*. Essa arquitetura foi concebida para aprimorar a flexibilidade do sistema, facilitando sua expansão e adaptação a diferentes demandas.

O conceito central por trás da divisão entre os *scripts* de treinamento e inferência é promover uma arquitetura orientada a objetos, onde cada um representa uma entidade autônoma e especializada em sua tarefa específica. Essa abordagem não apenas simplifica o entendimento do código, mas também facilita futuras extensões e modificações.

Ao dividir esses processos em *pipelines* distintos, cada um dedicado a uma fase específica do ciclo de vida do modelo, cria-se uma arquitetura escalável. Isso permite que cada *pipeline* seja dimensionado de maneira independente, respondendo de forma eficaz ao aumento da carga de trabalho ou a novas exigências.

O padrão de projeto adotado reflete a preocupação com a manutenibilidade e expansibilidade do sistema. A separação de responsabilidades entre os pipelines, combinada com a aplicação de padrões de projeto como o *Singleton* para garantir instâncias únicas e o *Factory Method* para criar objetos conforme a necessidade, contribui para um código mais organizado e resiliente. A estrutura geral do sistema é ilustrada na Figura 8, onde pode ser observada divisão entre os serviços em diretórios.

Figura 8 – *Estrutura geral do projeto*



Fonte: O próprio autor.

No âmbito da escalabilidade, a modularização dos processos em objetos e a adoção de um paradigma orientado a eventos, onde eventos significativos são registrados nos *logs*, proporcionam uma visão abrangente do fluxo de execução. Isso não só facilita a detecção de problemas, mas também a otimização contínua do desempenho do sistema.

Além disso, a incorporação de *logs* em todo o código, seguindo as melhores práticas de registro de eventos, fornece *insights* para o monitoramento do sistema. A integração

com ferramentas de monitoramento e análise dos registros contribui para a identificação de possíveis gargalos, erros ou oportunidades de otimização.

A arquitetura dos *pipelines* não apenas reflete uma abordagem pragmática para treinamento e inferência de modelos de detecção de anomalias, mas também é um testemunho do compromisso com a escalabilidade, a modularidade e a robustez do sistema. Esses princípios para atender às demandas dinâmicas de ambientes médicos e clínicos, onde a confiabilidade e a eficiência são de extrema importância.

3.2.1 Treinamento

O *pipeline* de treinamento foi projetado para treinar um modelo de detecção de anomalias usando o conjunto de dados rotulado da tabela de treinamento.

Ele importa as bibliotecas necessárias, incluindo bibliotecas específicas do TensorFlow, e outras para manipulação de dados e métricas de avaliação, e realiza a configuração do sistema de *log* padronizado para registrar eventos e mensagens relevantes durante sua execução.

O *script* de treinamento do *autoencoder* nos *pipelines* é estruturado incorporando princípios fundamentais de modularidade e flexibilidade para atender a diversas necessidades do processo. Ao longo do código, é evidente um cuidadoso design orientado a objetos e a implementação de padrões de projeto que favorecem a escalabilidade e facilitam a manutenção.

Uma característica desse *script* é a divisão em funções específicas, cada uma desempenhando um papel em diferentes estágios do treinamento. Essa abordagem modular não apenas melhora a clareza e a legibilidade do código, mas também facilita intervenções e modificações futuras. Por exemplo, a função responsável por preparar os dados garante uma abordagem específica para a segmentação entre ritmos normais e anômalos, sendo uma camada de abstração que pode ser ajustada para acomodar variações nas características dos dados.

A parametrização do processamento dos dados é configurada neste *pipeline*. Variáveis como datas de início e fim, proporção de dados de teste, semente de aleatoriedade, otimizador e função de perda. Isso não apenas fornece flexibilidade para adaptar o treinamento a diferentes conjuntos de dados e contextos, mas também facilita a replicação do experimento em cenários diversos.

A escalabilidade é aprimorada pela incorporação de variáveis de ambiente, permitindo que o *script* se ajuste dinamicamente a diferentes condições sem a necessidade de alterações no código-fonte. Essa abordagem é relevante em ambientes médicos, onde as características dos dados e os requisitos específicos podem variar substancialmente.

Ao considerar a manutenção do código, a modularidade visada se justifica. Cada função é encarregada de uma responsabilidade, tornando a localização e a resolução de problemas mais intuitivas. Essa divisão não só facilita a compreensão do código, mas também acelera o processo de adaptação a mudanças nos requisitos do treinamento ou nos conjuntos de dados.

A obtenção dos dados de treinamento é realizada através de uma função que interage com a base de dados para recuperar os dados relevantes dentro do intervalo de datas fornecido.

O treinamento do modelo é realizado configurando o modelo de *autoencoder* com o otimizador, função de perda e outros parâmetros definidos nas variáveis de ambiente.

Após o treinamento, o *script* utiliza o modelo treinado para realizar a detecção de anomalias nos dados de teste, calculando a perda de reconstrução e comparando-a com um limiar fixo.

As métricas de avaliação, como acurácia, precisão e *Recall*, são calculadas com base nas previsões do modelo em relação aos rótulos reais.

O modelo treinado, juntamente com informações como máximo, mínimo, *Limiar* e estatísticas de desempenho, é salvo na base de dados. A biblioteca *pickle* é utilizada para serializar os pesos do modelo antes de armazená-los no banco de dados.

O *script* pode ser executado a partir da linha de comando dentro do container, onde o argumento fornecido é a *tag* do modelo. Assim, o modelo treinado é salvo na base de dados associado a essa *tag*.

Este *script* desempenha um papel crucial no pipeline de treinamento do sistema de detecção de anomalias, seguindo a abordagem proposta neste trabalho, e representa uma etapa fundamental na implementação prática dos conceitos de MLOps.

Portanto, o *pipeline* treinamento não é apenas uma implementação eficaz do treinamento do *autoencoder*, mas também de programação, destacando a importância da modularidade, flexibilidade e escalabilidade em ambientes complexos.

3.2.2 Inferencia

O pipeline de inferência foi elaborado para realizar a inferência de dados de ECG usando um modelo previamente treinado. A estrutura modular do *script* de inferência nos pipelines é um elemento fundamental que proporciona uma série de benefícios notáveis para a eficiência e evolução contínua do sistema.

A configuração do sistema de *log* padronizado é realizada para registrar eventos e mensagens durante a sua execução. O *script* é capaz de recuperar um modelo previamente treinado da base de dados, usando a *tag* do modelo como identificador. Os pesos do

modelo são obtidos antes de serem carregados no modelo de *autoencoder*.

Os pesos do modelo referem-se aos parâmetros ou coeficientes que foram aprendidos durante o treinamento de um modelo de ML, neste caso o *autoencoder*, e são necessários para fazer previsões ou inferências. Em termos mais simples, são os valores atribuídos aos diferentes parâmetros do modelo que o capacitam a realizar tarefas específicas.

Quando um modelo é treinado, ele ajusta seus pesos com base nos dados de treinamento para minimizar a diferença entre as previsões do modelo e os rótulos reais. Esses pesos capturam padrões e relações importantes nos dados.

A abordagem modular adotada apresenta uma reusabilidade de código. Cada função no *script* pode ser compreendida e reutilizada em diversos contextos. Essa modularidade não apenas facilita a manutenção do código, tornando-a mais simples e direta, mas também oferece separação de responsabilidades, proporcionando um escopo delimitado para cada função.

A adaptabilidade é uma característica chave dessa estrutura modular. As mudanças ou melhorias podem ser implementadas de forma isolada, minimizando o impacto em outras partes do código. Isso ajuda a contribui para a evolução contínua do sistema ao longo do tempo.

A escalabilidade do *script* é evidente em várias frentes. A capacidade de integrar diferentes modelos com facilidade, alterando a *tag* do modelo, destaca a flexibilidade do sistema. Além disso, a configuração parametrizada através de variáveis de ambiente oferece uma flexibilidade considerável, especialmente em ambientes complexos com diferentes configurações de rede.

A arquitetura cliente-servidor, representada pela comunicação eficiente com a API através de funções dedicadas, é um componente chave para a escalabilidade do *script*. Isso o torna adaptável a diferentes infraestruturas e cenários de implementação.

A gestão centralizada de modelos no banco de dados é um ponto crucial para a escalabilidade na administração de múltiplos modelos. A capacidade de adicionar, remover ou atualizar modelos de forma centralizada no banco de dados simplifica consideravelmente a gestão operacional.

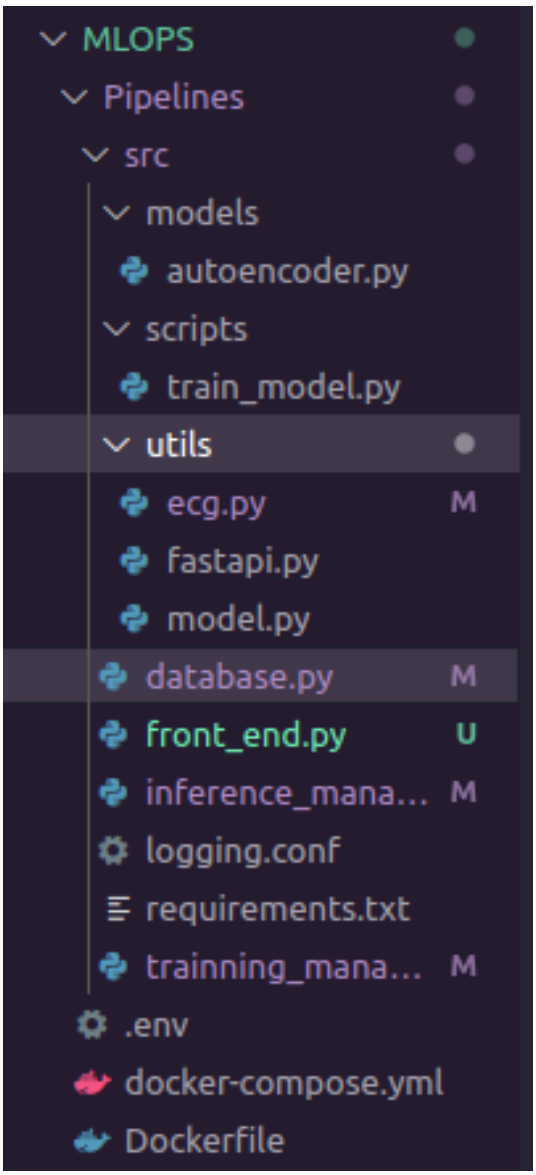
A implementação de princípios de orientação a objetos e padrões de projeto na estrutura do *script* fortalece ainda mais sua adaptabilidade e escalabilidade ao longo do tempo. Em resumo, a estrutura modular e escalável do *pipeline* de inferência não apenas otimiza as operações diárias, mas também prepara o sistema para enfrentar desafios futuros com robustez e flexibilidade.

Os resultados da inferência, incluindo a detecção de anomalias, são formatados e enviados para a API para a rota de predições e de ECG.

O *script* pode ser executado a partir da linha de comando, fornecendo a *tag* do modelo como argumento. Dessa forma, o *script* realiza a inferência usando o modelo associado a essa *tag*, interagindo com a API e registrando eventos conforme necessário.

Conforme ilustra a Figura 9, a estrutura dos Pipelines, é organizada de forma segregada, e planejada para receber futuras alterações no sistema, sem que haja retrabalho. Componentes do sistema são divididos em categorias, estabelecendo uma hierarquia estrutural escalável, como é o caso dos objetos que representam, o *ecg*, *fastapi* e *model*, o que representa que novos componentes podem ser agregados ao sistema sem grande complexidade.

Figura 9 – *Estrutura de arquivos dos pipelines*



Fonte: O próprio autor.

Portanto, esse *pipeline* desempenha um papel crucial no processo de inferência do sistema de detecção de anomalias, seguindo a abordagem proposta no neste trabalho, e

representa uma etapa fundamental na aplicação prática dos conceitos de MLOps.

3.2.3 Frontend

A interface desenvolvida utilizando a biblioteca **streamlit**, do *python*, vista em 2.2.5. Ela proporciona uma maneira acessível e compreensível de examinar os resultados gerados pelo sistema.

Além da simples visualização, ela oferece a capacidade de realizar auditorias nos resultados. Esse processo de auditoria envolve a rotulagem manual ou revisão técnica de eletrocardiogramas que passaram pelo *pipeline* de inferência.

A capacidade de rotular os resultados do ECG por meio de conhecimento técnico humano é fundamental no processo de evolução do modelo, pois permite a validação e correção das decisões automáticas realizadas nos *pipelines*. Especialistas podem aplicar sua experiência e discernimento para garantir que as classificações automáticas estejam alinhadas com o conhecimento médico e as nuances específicas do domínio.

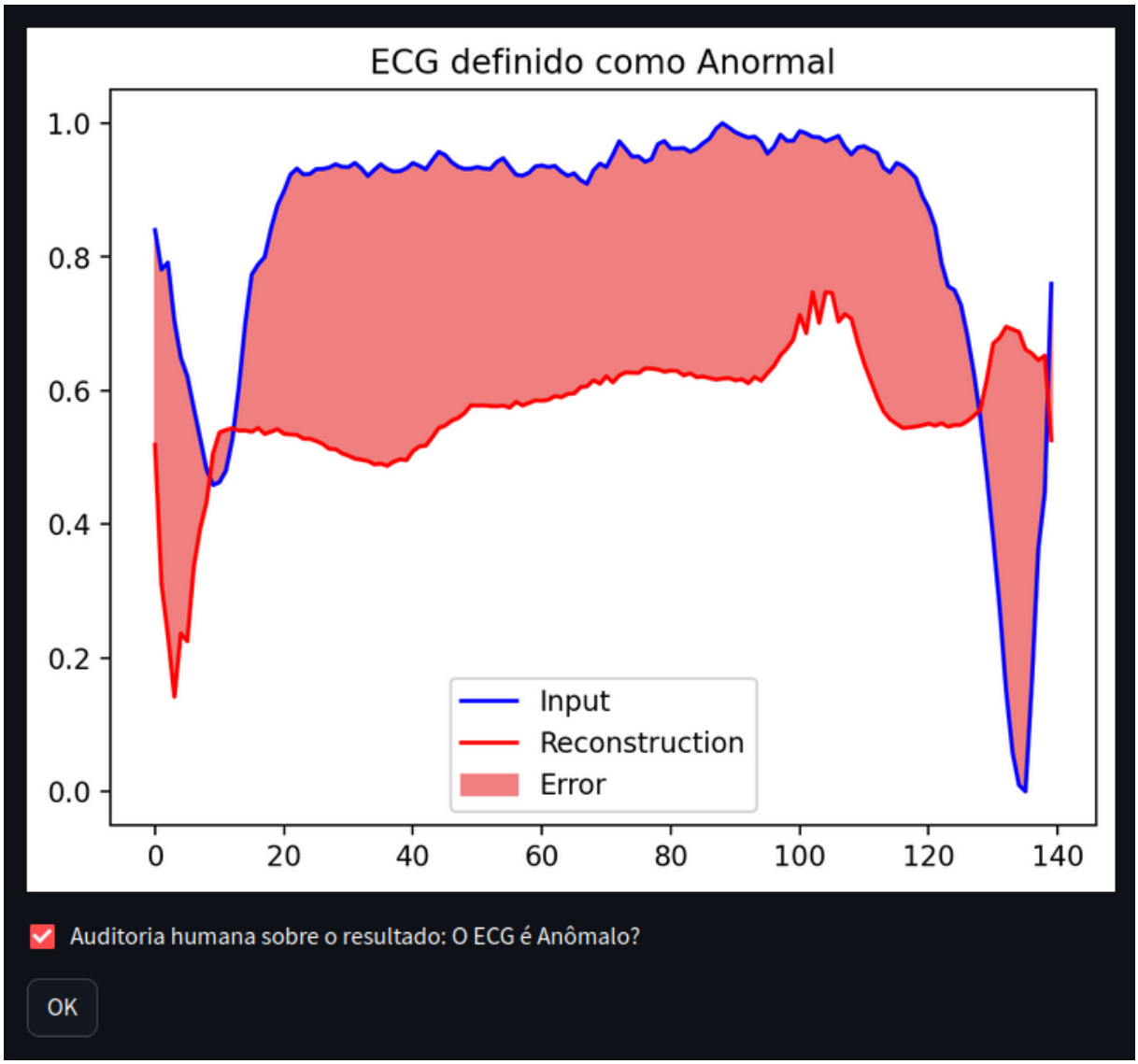
Essa funcionalidade é crucial para aprimorar a confiabilidade do sistema, uma vez que incorpora uma camada adicional de validação humana. Além disso, ao fornecer uma interface que facilita a interação humana com os resultados, a usabilidade do sistema é melhorada, possibilitando uma colaboração mais eficaz entre a inteligência artificial e os profissionais de saúde.

Este protótipo de interface foi integrado ao *pipeline* de inferência, visando fornecer uma perspectiva inicial sobre a interação entre os resultados do modelo e os profissionais de saúde. É importante ressaltar que este é um trabalho preliminar, e o objetivo principal não é desenvolver um *frontend* completo e robusto, mas sim apresentar uma abordagem para realimentar o ciclo de vida de aprimoramento do modelo.

A simplicidade da interface neste estágio é intencional, destacando que a ênfase está na metodologia e na possibilidade de interação humana com os resultados gerados pela inteligência artificial. Essa abordagem visa fornecer uma direção para melhorias futuras, reconhecendo que a interface pode evoluir e se aprimorar com base em *feedbacks*, requisitos específicos e aprofundamento na compreensão das necessidades dos usuários finais, como os profissionais da área da saúde.

A apresentação da interface ocorre após a execução de uma inferência, onde uma busca pelos dados mais recentes do ECG na tabela de resultados dos *pipelines* é realizada, na base de dados própria da saída dos *pipelines*. Esses dados incluem não apenas os registros do eletrocardiograma, mas também informações como a *tag* do modelo associado e a classificação de anomalia previamente atribuída pelo modelo, informações de vital importância para conclusões sobre o processo.

Figura 10 – *Interface de resultados- ECG anômalo*

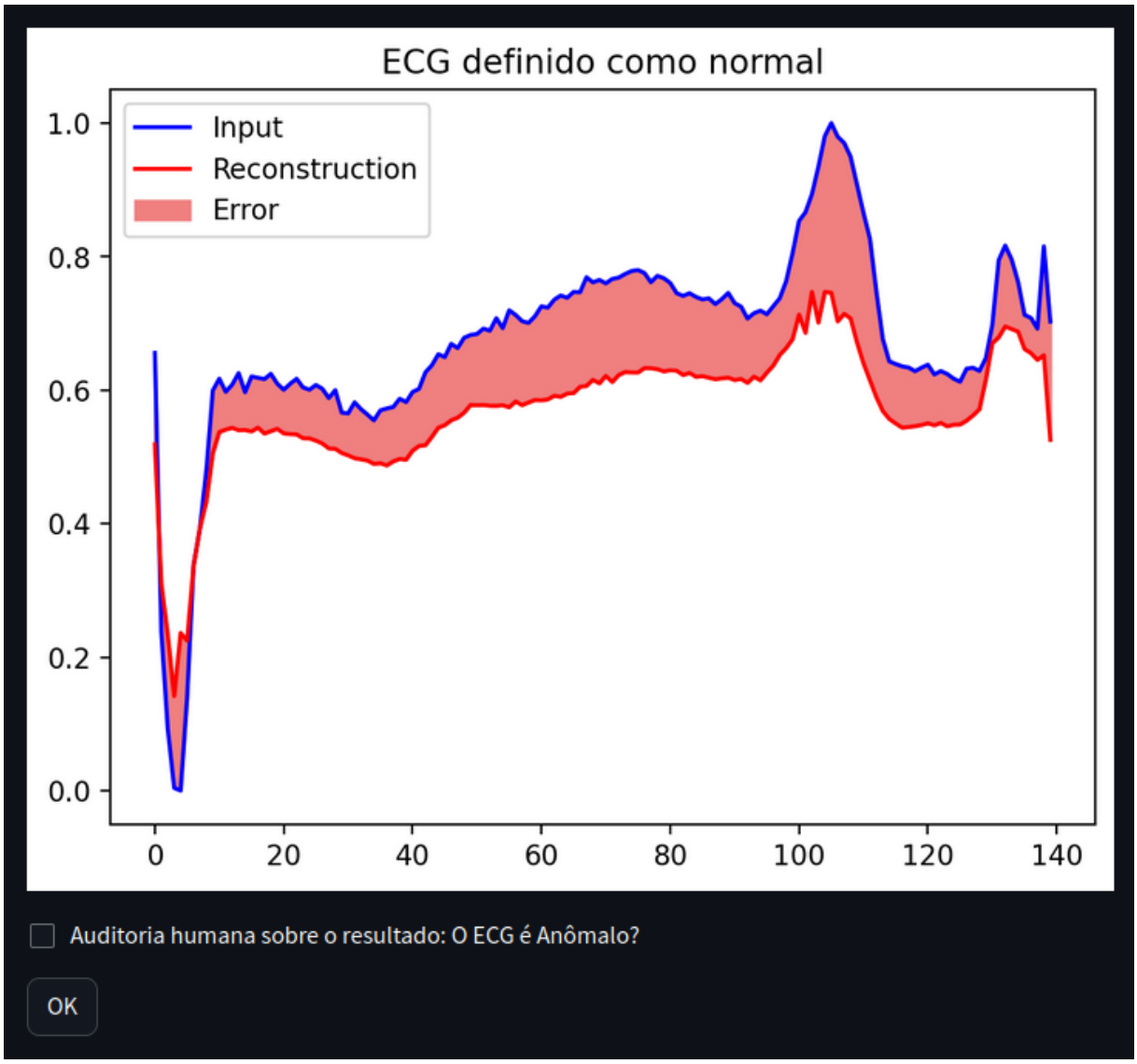


Fonte: O próprio autor.

Após isso, são coletados os dados da mesma base, porém em outra tabela destinada a armazenar as predições. Utilizando a biblioteca **matplotlib**, o *script* cria um gráfico que sobrepuja os registros do ECG original, as previsões do modelo e o erro absoluto entre eles. Esse gráfico destinado para que usuários possam examinar visualmente como as previsões do modelo se comparam aos dados reais.

Sua estrutura esta sendo ilustrada na Figura 10, em um caso de uso onde a leitura do sistema definiu o ECG inferido como anômalo. Nela a interface permite que os usuários inspecionem comportamento do modelo. Pode ser analisada a discrepância entre as previsões e os dados reais, decidindo se concordância ou não da classificação de anomalia feita pelo modelo. Um *checkbox* específico na interface permite que seja indicado se percebe-se o ECG como anômalo ou não. Isso incorpora o julgamento clínico, crucial

Figura 11 – *Interface de resultados- ECG normal*



Fonte: O próprio autor.

para aprimorar a precisão do modelo.

Um botão de confirmação associado a conclusão da auditoria sobre os resultados esta presente na interface. Ao ser pressionado, o usuário fornece *feedback* em tempo real sobre a classificação do modelo. Se o *checkbox* indicar que o ECG é anômalo, essa informação é refletida nos dados. As alterações são organizadas e ajustadas para corresponder à estrutura da tabela de treinamento do modelo. Isso garante que os dados validados pelos médicos sejam incorporados ao processo de treinamento do modelo, alimentando uma melhoria contínua.

Essa abordagem permite uma interação significativa entre o sistema de aprendizado de máquina e os médicos, criando um ciclo de *feedback* vital. Os eletrocardiogramas validados pelos médicos se tornam parte integrante do conjunto de treinamento, contri-

buindo para o desenvolvimento de modelos mais precisos e confiáveis ao longo do tempo. Esse processo iterativo é essencial para a adaptação contínua do sistema às nuances e complexidades dos dados médicos.

A Figura 11 ilustra uma inferência de um ECG classificado como normal, refletindo a sensibilidade do sistema de ML e da infraestrutura desenvolvida quanto a variação nos dados de inferência. Para este caso, o limiar de decisão estatístico do modelo, representado pelo valor 0.040375, é comparado com a perda média de reconstrução do ECG usado em cada inferência, classificando assim, como sendo anômalo ou não.

A medida que a compreensão das necessidades dos usuários se aprofunda, pode-se incorporar funcionalidades mais avançadas, aprimorar a usabilidade e garantir uma experiência mais intuitiva.

3.3 API de resultados

A API de resultados destaca-se como um componente essencial no panorama da aplicação, desempenhando um papel na gestão e arquivamento de dados pertinentes às predições e inferências de ECG. A estrutura do *script*, encapsulada no arquivo de rotas, reflete uma abordagem concisa e eficaz no emprego do *framework* FastAPI.

Ao explorar as funcionalidades da API, evidenciam-se dois *endpoints* principais que atendem distintos propósitos ***predictions*** e ***ecg***. Cada um se dedica a uma etapa específica do processo, permitindo uma demarcação entre os dados relacionados às predições e aqueles provenientes das inferências de eletrocardiogramas. Esse *design* modular contribui significativamente para a clareza e manutenção do código.

Destaca-se que a implementação dos *endpoints* é embasado no paradigma Representational State Transfer (REST), proporcionando um modelo de interação consistente e intuitivo. A rota *predictions* assume a responsabilidade de gerenciar dados associados a predições, enquanto a rota *ecg* concentra-se nas informações provenientes de inferências de ECG.

O FastAPI emerge como uma escolha acertada para o desenvolvimento dessa API devido à sua natureza assíncrona e baseada em python, o que simplifica a criação de forma eficiente. A abordagem propositiva e a capacidade de integração com o SQLAlchemy para interação com o banco de dados aprimoram a produtividade e legibilidade do código.

O padrão de injeção de dependência, uma característica marcante do FastAPI, é habilmente empregado nas funções associadas às rotas, promovendo uma estrutura modular e favorecendo a testabilidade do código. Esse padrão oferece vantagens expressivas na gestão de dependências, facilitando a manutenção e escalabilidade do sistema.

O tratamento de exceções é executado de maneira robusta, evidenciando uma

postura proativa em relação ao gerenciamento de erros. O registro adequado de mensagens de erro nos *logs* e a resposta consistente com uma exceção *HTTP 500* em caso de falhas na criação de registros contribuem para a estabilidade e resiliência do sistema.

Sua implementação, percebe-se uma escolha que ressoa na arquitetura do código, a adoção do padrão de projeto que preconiza a separação entre *schemas* e *models*. Essa decisão reflete uma abordagem consciente e orientada a melhores práticas de desenvolvimento de *software*.

Com essa abordagem, observa-se que os *schemas* desempenham um papel fundamental na validação dos dados recebidos nas requisições. Essas estruturas são projetadas para definir a forma e a estrutura dos dados, impondo critérios específicos, como tipos de dados e validações, garantindo que apenas informações conformes sejam processadas pela aplicação. Isso contribui para a integridade e consistência dos dados, minimizando potenciais problemas de integração.

Por outro lado, os *models* entram em cena como representações mais fiéis das entidades de dados no contexto do banco de dados. Essencialmente, eles encapsulam a lógica de como esses dados são armazenados e recuperados, estabelecendo uma ponte crucial entre a aplicação e a persistência dos dados.

Essa separação de responsabilidades propicia uma nível de escalabilidade, que possibilita alterações específicas em cada camada sem afetar desnecessariamente a outra.

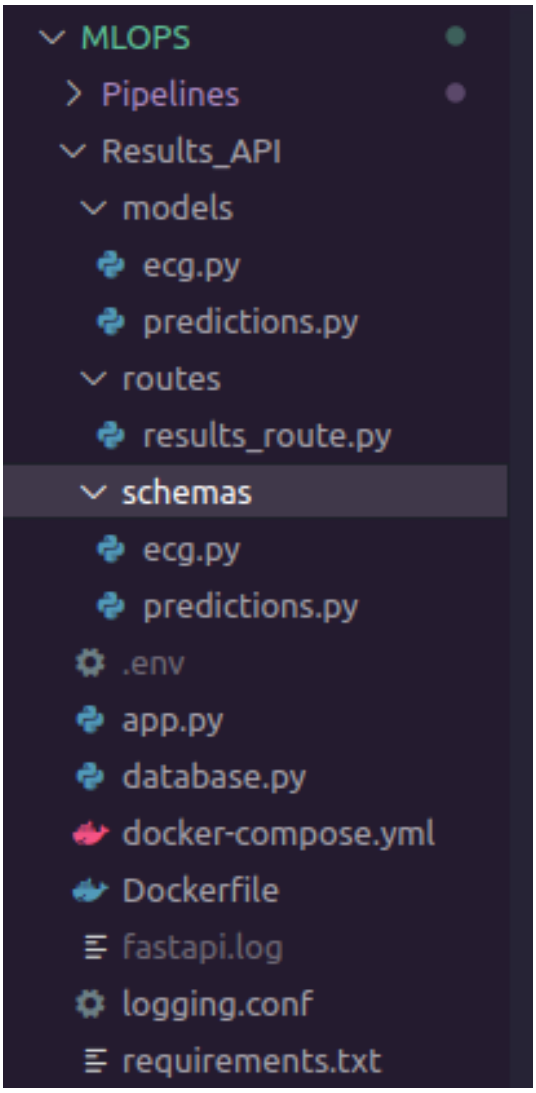
A abordagem de utilizar *schemas* para validação e *models* para persistência segue uma mentalidade coesa com os princípios de POO, paradigma adotado para implementação deste trabalho.

A separação de preocupações é alcançada, possibilitando que cada componente foque em sua responsabilidade única. Essa modularidade não apenas facilita a manutenção, mas também propicia uma base sólida para a evolução da aplicação. Essa abordagem alinhada com padrões de projeto fortalece a robustez da aplicação, simplificando o entendimento e propiciando uma base sólida para futuras expansões.

Toda a estrutura é ilustrada na Figura 12, onde fica evidente a organização dos diretórios, bem como o modelo de dockerização, cujo objetivo visa que a execução de eventuais mudanças na API ocorram mais facilmente.

Em resumo, a API de resultados incorpora boas práticas de desenvolvimento, garantindo uma abordagem organizada e eficiente para o gerenciamento de dados de predições e inferências de ECG. O emprego criterioso do FastAPI e a ênfase na modularidade e clareza do código reforçam a qualidade da implementação no contexto do sistema de detecção de anomalias.

Figura 12 – *Estrutura de arquivos dos resultados*



Fonte: O próprio autor.

3.4 Desenvolvimento e implantação

Tanto o desenvolvimento quanto a implantação deste projeto utiliza o Docker Compose, considerando que a maior parte dele é composto por contêineres em Docker, uma metodologia organizada e estruturada se torna importante. O docker compose é uma ferramenta que permite definir e gerenciar múltiplos contêineres Docker como uma aplicação única. Com ele, é possível descrever a configuração de todos os serviços, redes e volumes necessários para uma aplicação em um arquivo Ain't Markup Language (YAML), facilitando a orquestração eficiente de serviços, redes e volumes. Esse recurso garante ambientes reproduzíveis e evita conflitos entre diferentes versões de bibliotecas.

A orquestração simplificada oferecida pelo Docker Compose é útil em cenários onde a infraestrutura precisa ser escalada para lidar com diferentes volumes de tarefas, como treinamento e inferência de modelos. Além disso, a facilidade de colaboração é promovida,

pois o arquivo Docker Compose, muitas vezes versionado junto com o código-fonte, garante uma configuração consistente para toda a equipe.

A estrutura modular, onde cada componente é encapsulado em seu próprio contêiner, simplifica a manutenção, facilita a experimentação e otimiza a escalabilidade. A independência entre os contêineres permite que cada parte do sistema evolua de maneira autônoma, minimizando o impacto de alterações em um componente sobre os outros. Isso não apenas simplifica as operações de manutenção, correção de *bugs* e implementação de novos recursos, mas também oferece flexibilidade para atualizações específicas.

A capacidade de substituir ou atualizar facilmente o contêiner que abriga a componente de ML facilita a experimentação com diferentes modelos ou técnicas de detecção de anomalias. Essa flexibilidade contribui para testes comparativos, permitindo avaliar e escolher a abordagem mais eficaz.

A escolha de executar manualmente os *scripts* de treinamento e inferência via linha de comando dentro do Docker apresenta desvantagens significativas. Essa abordagem envolve intervenção humana, o que pode resultar em erros, atrasos e inconsistências. Além disso, ela não é escalável para ambientes com grandes volumes de dados ou operações frequentes.

A execução manual carece de eficiência operacional e pode levar a esquecimentos ou negligência de tarefas programadas. A automação é crucial para garantir escalabilidade, especialmente em cenários que exigem treinamento regular de modelos ou inferências em tempo real. Além disso, a execução manual dificulta o monitoramento detalhado e a análise de métricas, em comparação com sistemas automatizados.

Apesar dessas limitações, é importante ressaltar que a decisão de não otimizar a coleta de dados de ECG ou integrar diretamente com agentes de coleta primários é parte de uma estratégia voltada para escalabilidade e modularidade. O foco principal do trabalho é estabelecer uma infraestrutura flexível, capaz de se adaptar a diferentes componentes e evoluir eficientemente diante de mudanças nos requisitos ou nas fontes de dados.

No ambiente Docker, a configuração de redes na comunicação entre contêineres e no acesso externo é importante. Dois modos comuns de redes são *bridge* e *host*.

O modo *bridge* isola contêineres em redes separadas, enquanto o modo *host* permite que eles compartilhem a rede do *host* onde o Docker está instalado, eliminando a barreira de isolamento. Neste trabalho, optou-se pela configuração *host* para simplificar a comunicação entre os processos de *pipelines* e resultados, permitindo fácil acesso a serviços fora dos contêineres, como os bancos de dados.

Essa escolha foi feita visando simplicidade, mas é possível explorar outras configurações de rede para atender a requisitos específicos de segurança e arquitetura. Por exemplo, criar redes separadas para os contêineres de *pipelines* e resultados proporcionaria

um maior isolamento, uma prática mais robusta em ambientes sensíveis à segurança. Essa flexibilidade na configuração de redes no Docker destaca a adaptabilidade do projeto para incorporar práticas mais avançadas conforme necessário.

O processo de desenvolvimento e implantação utilizando arquivos Docker Compose revela uma abordagem eficiente para gerenciar os serviços. O arquivo Docker Compose principal utilizado localmente durante o desenvolvimento, define um serviço construído a partir de um contexto local usando o Dockerfile, cujo papel é especificar a estrutura e configurações do contêiner usado nos serviços. O volume de cada serviços é mapeado, permitindo que o código-fonte local seja refletido dentro do contêiner, facilitando atualizações em tempo real durante o desenvolvimento.

As configurações de *log* são personalizadas para garantir a manutenção dos registros do serviço. Um arquivo oculto é utilizado para carregar variáveis de ambiente relevantes de parametrização de todos os contêineres. Foi estabelecido um sistema de *logs* que compreende diferentes elementos, como *loggers*, que definem a categoria e o nível de registros, *handlers*, responsáveis por direcioná-los para destinos específicos e *formatters*, que formatam sua apresentação.

Essa configuração padronizada facilita o entendimento e a análise dos registros, contribuindo para a clareza e a legibilidade do sistema. A utilização de um arquivo de configuração externo, promove escalabilidade ao possibilitar ajustes na configuração sem a necessidade de alterações no código-fonte, proporcionando uma abordagem flexível e fácil de manter.

O docker compose permite realizar uma configuração de limitação do tamanho dos arquivos de registro dos *logs*. A configuração define o *driver* utilizado no padrão do arquivos, sendo o *json-file* adotado neste trabalho, indicando que os *logs* são registrados em um formato JavaScript Object Notation (JSON) para facilitar o processamento e análise posterior.

É também estabelecido um limite de tamanho para cada arquivo de *log* gerado, neste caso, definindo que cada arquivo não deve exceder dez megabytes. Essa configuração é uma prática importante em ambientes de MLOps para evitar que o tamanho dos arquivos cresça indefinidamente e consumam todo o espaço disponível no disco do *host*, principalmente em ambientes de produção.

Ao atingir o tamanho máximo definido, um novo arquivo de *log* é criado, e o antigo pode ser arquivado, rotacionado ou removido, dependendo da configuração do docker compose.

Durante o desenvolvimento é importante fazer o levantamento das bibliotecas e de suas respectivas versões em um arquivo designado para isso, para manter as versões utilizadas registradas. A dockerização então desempenha um papel crucial na gestão

eficiente de bibliotecas, especialmente no ecossistema do TensorFlow. Sua integração com um arquivo de requisitos oferece benefícios significativos, pois simplifica e assegura a compatibilidade de versões entre todas as bibliotecas essenciais no desenvolvimento de modelos de ML.

Ao definir todas as dependências, incluindo bibliotecas específicas, no arquivo de requisitos, cria-se uma especificação explícita do ambiente necessário para o projeto. O Docker, então, utiliza esse arquivo para construir uma imagem provendo todas as bibliotecas e configurações requeridas. Esse ambiente encapsulado é portátil, garantindo que, independentemente do ambiente de execução, as versões corretas das bibliotecas estejam disponíveis.

No contexto do TensorFlow, onde a compatibilidade de versões pode ser crucial, o Docker oferece um ambiente isolado e independente para cada projeto. Dessa forma, diferentes projetos podem coexistir em um mesmo *host*, mesmo que demandem versões distintas de bibliotecas. Isso evita conflitos entre projetos que poderiam surgir se todos compartilhassem o mesmo ambiente do sistema operacional.

Uma configuração definida de reinício automático é configurada para garantir que o serviço dentro do contêiner seja reiniciado em caso de falha. Ela desempenha um papel fundamental nas práticas de MLOps, sendo importante para manter a estabilidade e a disponibilidade dos serviços. Isso minimiza o tempo de inatividade e contribui para uma recuperação rápida de falhas temporárias, como erros no código ou exceções não tratadas. Essa prática em sistemas de ML se torna indispensável, uma vez que modelos são frequentemente treinados, inferências realizadas e atualizações aplicadas.

Em resumo, o desenvolvimento local é facilitado pelo Docker Compose, proporcionando um ambiente de teste fácil de configurar, fomentando a possibilidade de criar um Continuous Integration (CI) para criar e versionar imagens, armazenando-as em um repositório próprio. Na implantação, o Docker Compose é novamente acionado, garantindo que os serviços sejam executados com as versões específicas das imagens armazenadas no repositório. Essa abordagem baseada em contêineres e repositórios centralizados oferece uma integração contínua eficiente e um histórico versionado para o ciclo de vida do aplicativo.

3.4.1 Continuous integration

Na fase de CI implementado neste trabalho, o Docker Compose é utilizado para construir imagens Docker. O processo envolve a execução dos Dockerfiles especificados nos serviços envolvendo os contêineres de *pipelines* e de resultados. As imagens resultantes são marcadas com uma *TAG* específica, criando uma versão identificável.

Durante o processo, essas imagens são enviadas para o repositório de artefatos,

onde são armazenadas de forma organizada. Isso cria um histórico versionado e acessível das imagens geradas durante o desenvolvimento.

Um *script* em **Bash** foi desenvolvido para representar o processo de *pipeline* de integração contínua. Sua função principal é automatizar diversas etapas do processo de versionamento utilizando Git, na construção de contêineres Docker e *upload* de artefatos para um repositório. Ele é acionado ao momento em que a equipe de desenvolvimento converge para uma versão estável do sistema e decide gerar uma *tag* de versão do Git, o *script* então é executada e deve receber como argumento o nome da tag desejada para a versão, bem como o nome de usuário, senha do repositório que armazena as imagens docker. Deve ser ajustado diretamente no arquivo o endereço do *host* em que o repositório esta configurado.

Além disso, o script gera arquivos compactados para os pipelines e resultados, encapsulando os artefatos relevantes. Esses arquivos são então enviados para o repositório usando a ferramenta cURL, autenticando-se com as credenciais fornecidas.

Em suma, o *script* automatiza uma série de tarefas essenciais para garantir uma entrega contínua e eficiente, facilitando o versionamento, construção de contêineres e armazenamento organizado de artefatos. Esse processo é fundamental para manter a consistência e a qualidade na evolução do software.

3.4.2 Variaveis de ambiente

A escolha estratégica de utilizar variáveis de ambiente em arquivos ocultos nomeados de *.env* proporciona uma série de vantagens cruciais para o desenvolvimento e a escalabilidade de sistemas.

Em primeiro lugar, essa abordagem simplifica consideravelmente a configuração de diferentes ambientes, tornando fácil a modificação de configurações, como *hosts* ou credenciais de banco de dados, apenas ajustando o arquivo de correspondente.

Além disso, ao centralizar todas as configurações em um único local, a prática facilita enormemente o gerenciamento e a manutenção dessas variáveis, especialmente em projetos complexos com diversas partes interdependentes. A padronização do uso de variáveis de ambiente também se alinha a boas práticas da indústria, promovendo um código mais compreensível para novos desenvolvedores ou membros da equipe.

Do ponto de vista da segurança, o emprego de variáveis de ambiente contribui para a proteção de informações sensíveis, como senhas e chaves de API. Evitar a exposição direta desses dados no código-fonte é crucial para mitigar riscos de vazamentos acidentais ou comprometimento de informações críticas.

A flexibilidade proporcionada por esse método é outra vantagem destacável. A

possibilidade de ajustar dinamicamente a configuração do sistema, sem a necessidade de alterações no código-fonte, em ambientes onde as configurações podem variar frequentemente. Além disso, a compatibilidade com ferramentas de orquestração, como Docker Compose ou Kubernetes, é aprimorada, facilitando a portabilidade e a execução consistente em diferentes ambientes.

Por fim, a manutenção simplificada é um benefício adicional dessa abordagem. Isolando as configurações em um arquivo, as alterações podem ser realizadas sem a necessidade de modificar o código-fonte. Esse método também facilita o rastreamento de alterações, especialmente quando se utiliza controle de versão, proporcionando uma visão e organizada das mudanças nas configurações ao longo do tempo.

3.4.3 Armazenamento no Nexus

Neste trabalho, como repositório de artefatos, foi escolhido o Nexus, articulado na seção 2.2.2. O Nexus, desempenha um papel crucial, as imagens construídas durante o processo de CI são armazenados nele. Isso cria um histórico organizado e versionado das imagens, permitindo que as versões específicas sejam recuperadas e implantadas posteriormente.

Quando se trata da implantação em ambientes de produção ou teste, o Docker Compose é novamente utilizado. Desta vez, ele puxa as imagens armazenadas no Nexus, garantindo consistência entre ambientes.

Neste contexto, embora a implementação de entrega contínua CI/CD não tenha sido automatizada diretamente, a utilização do Nexus desempenha um papel crucial na intermediação para essa integração.

A utilização de *tags* para representar versões estáveis no Nexus é uma prática comum, proporcionando uma maneira eficaz de rastrear e implantar versões específicas do produto.

Essa prática ágil facilita a rápida disponibilidade de versões estáveis do sistema, gerenciadas por *tags* específicas no Nexus. As imagens Docker associadas a essas versões podem ser implantadas de maneira escalável e modular em diversos cenários de produção. A implementação do um fluxo de CI neste trabalho, juntamente com o Nexus, proporciona não apenas eficiência no desenvolvimento, mas também uma base sólida para a expansão futura do processo de entrega contínua.

Além disso, destaca-se que a automação adicional, como o uso de ferramentas como Ansible, pode otimizar ainda mais o processo de implantação e gerenciamento do sistema de ML em ambientes diversos.

4 CONCLUSÕES

O presente trabalho se propôs a enfrentar os desafios de desenvolver um sistema de ML capaz de operar eficientemente em ambientes de produção. A complexidade na preparação e implementação da infraestrutura necessária para suportar esse sistema em um ambiente real, destacou que de fato, o desenvolvimento do código de ML em si não necessariamente é a etapa mais complexa, mas também o trabalho de habilitar esse código e o sistema para operar em ambientes reais.

Desenvolver a infraestrutura necessária para operações em produção exigiu um planejamento e uma implementação robusta. Desde a escolha e configuração de contêineres Docker até a orquestração eficiente desses contêineres e a implementação de práticas de CI/CD.

A implementação das práticas de MLOps evidenciou que a transição do desenvolvimento para a produção é uma jornada complexa que demanda planejamento, execução cuidadosa e um entendimento profundo das necessidades do sistema. Ao explorar as práticas de MLOps, uma infraestrutura robusta e escalável foi desenvolvida, que não apenas suporta a execução de modelos de ML, mas também facilita sua implantação e gerenciamento em ambientes de produção. A escolha e configuração cuidadosa de contêineres Docker, a orquestração eficiente desses contêineres, a implementação de práticas de CI/CD e a gestão de variáveis de ambiente foram apenas algumas das muitas etapas necessárias para construir essa infraestrutura.

Ao integrar uma interface que permite auditar as decisões do modelo e retroalimentar essas informações para a base de treinamento, o ciclo de melhoria contínua do modelo é consumado. Isso não apenas aumenta a transparência do sistema, mas também contribui para o aprimoramento do desempenho do modelo ao longo do tempo.

Uma das principais conclusões a extrair deste trabalho é que o desenvolvimento de um notebook Jupyter, é apenas o primeiro passo de uma jornada muito mais longa e desafiadora. A transição para um ambiente de produção requer um conjunto diferente de habilidades e conhecimentos, que vão além da análise de dados. Essa transição envolve aspectos como escalabilidade, confiabilidade e eficiência operacional, que são essenciais para garantir o bom funcionamento do sistema em condições reais.

Em suma, este trabalho demonstrou que desenvolver um sistema de ML capaz de operar em ambientes de produção é uma tarefa complexa que requer uma abordagem cuidadosa e uma compreensão profunda dos desafios envolvidos. As práticas de MLOps oferecem uma estrutura sólida e eficiente para lidar com esses desafios, permitindo que soluções desenvolvidas utilizando IA explorem ao máximo seu potencial em suas operações

em produção.

4.1 Trabalhos futuros

Para melhorias e continuidade da pesquisa, sugere-se:

1. Finalizar a entrega continua do CI/CD;
2. Criar um *frontend* robusto;
3. Criar um sistema de autenticação na API e mecanismos de segurança na infraestrutura como um todo.

REFERÊNCIAS

ASHMORE, R.; CALINESCU, R.; PATERSON, C. Assuring the machine learning lifecycle: Desiderata, methods, and challenges. *ACM Comput. Surv.*, Association for Computing Machinery, New York, NY, USA, v. 54, n. 5, may 2021. ISSN 0360-0300. Disponível em: <https://doi.org/10.1145/3453444>. 12, 13, 16

BANSAL, P.; OUDA, A. Study on integration of fastapi and machine learning for continuous authentication of behavioral biometrics. In: *2022 International Symposium on Networks, Computers and Communications (ISNCC)*. [S.l.: s.n.], 2022. p. 1–6. 28

COMBE, T.; MARTIN, A.; PIETRO, R. D. To docker or not to docker: A security perspective. *IEEE Cloud Computing*, v. 3, n. 5, p. 54–62, 2016. 26

GARG, S. et al. On continuous integration / continuous delivery for automated deployment of machine learning models using mlops. In: *2021 IEEE Fourth International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*. [S.l.: s.n.], 2021. p. 25–28. 12, 14, 26

GOOGLE. *TensorFlow*. 2024. Https://www.tensorflow.org/. 27

GOOGLE. *TensorFlow autoencoder*. 2024. Https://www.tensorflow.org/tutorials/generative/autoencoder. 30

INC., S. *Nexus*. 2022. Https://www.sonatype.com/products/sonatype-nexus-repository. 27

NG, A. *Machine Learning Engineering for Production (MLOps)*. 2022. Https://www.deeplearning.ai/courses/machine-learning-engineering-for-production-mlops/. 13, 14, 15, 16, 17, 18, 19, 20, 24

NG, A. *Machine Learning Engineering for Production (MLOps)*. 2022. Urlhttps://www.deeplearning.ai/resources/. 16, 18, 21

RAMíREZ, S. *Fast API*. 2022. Https://fastapi.tiangolo.com/. 28

RIEBESELL, J. *Autoencoder*. 2022. Https://tikz.net/autoencoder/. 29

SCULLEY, D. et al. Hidden technical debt in machine learning systems. In: CORTES, C. et al. (Ed.). *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2015. v. 28. Disponível em: <https://proceedings.neurips.cc/paper/2015/file/86df7dcfd896fcaf2674f757a2463eba-Paper.pdf>. 14, 15

SNOWFLAKE. *Streamlit*. 2024. Https://https://streamlit.io/. 28

Apêndices

APÊNDICE A – REPOSITÓRIO DO PROJETO

O *link* abaixo fornece acesso ao repositório com o código desenvolvido neste trabalho.

https://github.com/Faracoeng/TCC2

APÊNDICE B – TRECHO PIPELINE DE INFERÊNCIA

O *pipeline* de inferência foi desenvolvido em *python3*.

Código B.1 – Pipeline de inferência

```python
from models.autoencoder import AnomalyDetector
from datetime import datetime
from utils.ecg import ECG
from utils.model import *
from utils.fastapi import *
import tensorflow as tf
import logging.config
from database import *
import pickle
import time
import subprocess

logging.config.fileConfig('/app/src/logging.conf')
logger = logging.getLogger()
autoencoder = AnomalyDetector()
ecg = ECG()

def load_model_from_database(tag):
    try:
        engine = create_database_engine(db_origem)
        Session = sessionmaker(bind=engine)
        session = Session()

        model_info = session.query(Model).filter_by(tag=tag).first()
        if model_info:
            model_weights_deserialized = pickle.loads(model_info.model_weights)
            new_model = AnomalyDetector()
            new_model.build((1, 140))
            new_model.set_weights(model_weights_deserialized)
            return new_model
    except Exception as e:
        logger.error(f"Erro ao realizar inferência: {str(e)}")
        return None

def get_model_max_min_values(tag):
    try:
        session = get_session_Pipelines() #Session()
```

```
38          model_info = session.query(Model).filter_by(tag=tag).first()
39          if model_info:
40              return model_info.max_value, model_info.min_value
41          else:
42              logger.error(f"Modelo com a tag '{tag}' não encontrado na base de dados
    .")
43              return None, None
44      except Exception as e:
45          logger.error(f"Erro ao realizar inferência: {str(e)}")
46          return None, None
47
48  def get_model_threshold(tag):
49      try:
50          session = get_session_Pipelines()
51          model_info = session.query(Model).filter_by(tag=tag).first()
52          if model_info:
53              return model_info.threshold
54          else:
55              logger.error(f"Modelo com a tag '{tag}' não encontrado na base de dados
    .")
56              return None
57
58      except Exception as e:
59          logger.error(f"Erro ao realizar inferência: {str(e)}")
60          return None
61
62  def get_api_environment():
63      try:
64          api_envs = {
65              "api_port": os.environ.get('API_PORT'),#"5000",
66              "api_host": os.environ.get('API_HOST'),#"localhost",
67              "predictions_route": os.environ.get('PREDICT_ROUTE'),#"/predictions",
68              "ecg_route": os.environ.get('ECG_ROUTE')#"/ecg"
69          }
70          logger.info("Variáveis de ambiente da API carregadas com sucesso")
71      except Exception as e:
72              logger.error(f"Erro ao obter variáveis de ambiente da API: {str(e)}")
73      return api_envs
74
75  def api_client(data, host, port, route):
76      try:
77          client = FastAPIClient()
78          client.set_host(host)
79          client.set_port(port)
80          logger.info(f"Enviando dados para a rota {route}.")
81          response = client.send_api_post(route, data)
82          logger.info("Resposta da API: %s", response)
```

```
83          logger.info("Post enviado com sucesso para a API")
84      except Exception as e:
85          logger.error(f"Erro ao enviar dados para a API: {e}")
86
87  def inference_manager(model_tag):
88      data= ecg.get_ECG_inference_data()
89      max_value, min_value = get_model_max_min_values(model_tag)
90      normalazed_inference_data = ecg.normalize_data(data, max_value, min_value)
91      try:
92          model = load_model_from_database(model_tag)
93          if model:
94              api_environment = get_api_environment()
95              predictions = model.predict(normalazed_inference_data)
96              data_reconstructions_loss = tf.keras.losses.mae(predictions,
    normalazed_inference_data)
97              model_threshold = get_model_threshold(model_tag)
98              logger.info(f"data_reconstructions_loss é:   ---> {
    data_reconstructions_loss}")
99              logger.info(f"model_threshold é:   ---> {model_threshold}")
100             if float(data_reconstructions_loss) > float(model_threshold):
101                 anomaly_detected = 0
102             else: anomaly_detected = 1
103             if anomaly_detected==0:
104                 logger.info("Anomalia detectada")
105             inference_ecg_data = {
106                 "dt_measure": datetime.utcnow().isoformat(),
107                 "is_anomalous": anomaly_detected,
108                 "model_tag": model_tag,
109                 "values": normalazed_inference_data[0].numpy().tolist()  #
    Convertendo o array NumPy para uma lista
110             }
111             api_client(inference_ecg_data, api_environment['api_host'],
    api_environment['api_port'], api_environment['ecg_route'])
112             prediction_data = {
113                 "dt_measure": datetime.utcnow().isoformat(),
114                 "model_tag": model_tag,
115                 "values": [],
116             }
117             enumerated_predictions = list(enumerate(predictions[0]))
118             for i, value in enumerated_predictions:
119                 prediction_data["values"].append(float(value))
120             api_client(prediction_data, api_environment['api_host'],
    api_environment['api_port'], api_environment['predictions_route'])
121             time.sleep(2)
122             script_path = 'front_end.py'
123             command = f'streamlit run {script_path}'
124             subprocess.run(command, shell=True)
```

```
125             logger.info("Inferência concluída com sucesso")
126         else:
127             logger.error("Falha ao realizar inferência. Modelo não encontrado.")
128             return None
129     except Exception as e:
130         logger.error(f"Erro ao realizar inferencia: {str(e)}")
131         return None
132 if __name__ == "__main__":
133     if len(sys.argv) < 2:
134         logger.error("Por favor, forneça o nome do modelo como argumento que deseja
      utilizar na inferência.")
135         sys.exit(1)
136     model_tag = sys.argv[1]
137     logger.info(f"Argumento do modelo recebido: {model_tag}")
138     inference_manager(model_tag)
```

APÊNDICE C – TRECHO PIPELINE DE TREINAMENTO

O *pipeline* de treinamento foi desenvolvido em *python3*.

Código C.1 – Pipeline de treinamento

```
import os
from models.autoencoder import AnomalyDetector
from datetime import datetime
from utils.ecg import ECG
from utils.model import *
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn.metrics import accuracy_score, precision_score, recall_score
import logging.config
from database import *
import pickle

logging.config.fileConfig('/app/src/logging.conf')
logger = logging.getLogger()

autoencoder = AnomalyDetector()
ecg = ECG()


def get_environment_variables():
    try:
        processing_variables = {
            "begin_date": os.environ.get('BEGIN_DATE'),#"2023-11-24",
            "end_date":  os.environ.get('END_DATE'),#"2023-11-28",
            "test_size": os.environ.get('TEST_SIZE'),#0.2,
            "random_state": os.environ.get('RANDOM_STATE'),#21,
            "optimizer": os.environ.get('OPTIMIZER'),#"adam",
            "loss_function": os.environ.get('LOSS_FUNCTION'),#"mae",
            "epochs": os.environ.get('EPOCHS'),#20,
            "batch_size": os.environ.get('BATCH_SIZE')#512
        }
        logger.info("Variáveis de ambiente de processamento dos dados carregadas
    com sucesso")

    except Exception as e:
```

```
36          logger.error(f"Erro ao obter variáveis de ambiente de processamento dos
        dados: {str(e)}")
37
38      return processing_variables
39
40  def get_train_data():
41      try:
42          start_date = get_environment_variables()["begin_date"],#"2023-11-24"
43          end_date = get_environment_variables()["end_date"],#"2023-11-28"
44          start_date = datetime.strptime(start_date[0], '%Y-%m-%d').date()
45          end_date = datetime.strptime(end_date[0], '%Y-%m-%d').date()
46          raw_data = ecg.get_ECG_train_data(start_date, end_date)
47          logger.info("Dados de treinamento obtidos da base de dados")
48          return raw_data
49      except Exception as e:
50          logger.error("Certifique-se de que as datas estão no formato 'YYYY-MM-DD'."
        )
51
52  def predict(model, data, threshold):
53    reconstructions = model(data)
54    loss = tf.keras.losses.mae(reconstructions, data)
55    return tf.math.less(loss, threshold)
56
57  def get_model_stats(predictions, labels):
58      logger.info("Accuracy = {}".format(accuracy_score(labels, predictions)))
59      logger.info("Precision = {}".format(precision_score(labels, predictions)))
60      logger.info("Recall = {}".format(recall_score(labels, predictions)))
61      accuracy = accuracy_score(labels, predictions)
62      precision = precision_score(labels, predictions)
63      recall = recall_score(labels, predictions)
64
65      return accuracy, precision, recall
66
67  def train_manager(model_tag):
68      try:
69          raw_data = get_train_data()
70          labels = ecg.get_labels(raw_data)
71          data = ecg.get_ecg_points(raw_data)
72          train_data, test_data, train_labels, test_labels = ecg.get_train_test_split
        (data, labels, float(get_environment_variables()['test_size']), int(
        get_environment_variables()['random_state']))
73          max_value, min_value = ecg.get_min_max_val(train_data)
74          max_value_python = max_value.numpy()
75          min_value_python = min_value.numpy()
76          normalazed_train_data = ecg.normalize_data(train_data, max_value, min_value
        )
77          normalazed_test_data = ecg.normalize_data(test_data, max_value, min_value)
```

```
78          normal_train_data, normal_test_data = ecg.normal_train_test_generate(
    normalazed_train_data, normalazed_test_data, train_labels, test_labels)
79          autoencoder.compile(optimizer=get_environment_variables()['optimizer'],
    loss=get_environment_variables()['loss_function'])
80          history = autoencoder.fit(normal_train_data, normal_train_data,
81                  epochs=int(get_environment_variables()['epochs']),
82                  batch_size=int(get_environment_variables()['batch_size']),
83                  validation_data=(normalazed_test_data, normalazed_test_data),
84                  shuffle=True)
85          normal_train_data_reconstructions = autoencoder.predict(normal_train_data)
86          normal_train_data_reconstructions_train_loss = tf.keras.losses.mae(
    normal_train_data_reconstructions, normal_train_data)
87          threshold = np.mean(normal_train_data_reconstructions_train_loss) + np.std(
    normal_train_data_reconstructions_train_loss)
88          print("Threshold: ", threshold)
89          preds = predict(autoencoder, normalazed_test_data, threshold)
90          accuracy, precision, recall = get_model_stats(preds, test_labels)
91          model_weights = autoencoder.get_weights()
92          save_model_to_database(model_tag, max_value_python, min_value_python,
    threshold, model_weights, accuracy, precision, recall)
93      except Exception as e:
94          logger.error(f"Erro ao criar modelo: {e}")
95
96  def save_model_to_database(tag, max_value, min_value, threshold, model_weights,
    accuracy, precision, recall):
97      try:
98          Base.metadata.create_all(engine_Pipelines)
99
100         Session = sessionmaker(bind=engine_Pipelines)
101         session = Session()
102         model_weights_serialized = pickle.dumps(model_weights)
103         model_info = Model(
104             tag=tag,
105             max_value=max_value,
106             min_value=min_value,
107             threshold=threshold,
108             accuracy=accuracy,
109             precision=precision,
110             recall=recall,
111             model_weights=model_weights_serialized
112         )
113         session.add(model_info)
114         session.commit()
115
116         logger.info(f"Modelo e estatisticas salvas na base com a tag '{tag}'")
117
118     except Exception as e:
```

```
119         logger.error(f"Erro ao salvar o modelo na base: {str(e)}")
120
121 if __name__ == "__main__":
122     if len(sys.argv) < 2:
123         logger.error("Por favor, forneça o nome do modelo como argumento.")
124         sys.exit(1)
125
126     model_tag_argument = sys.argv[1]
127     logger.info(f"Argumento do modelo recebido: {model_tag_argument}")
128     train_manager(model_tag_argument)
```

APÊNDICE D – UTILS PIPELINES

Foram criados *scripts*, também em python3 para representar objetos, sendo eles o Modelo de ML, um ECG e a API.

Código D.1 – Modelo de ML

```
1
2  from sqlalchemy import Column, Integer, String, Float, LargeBinary
3  from sqlalchemy.ext.declarative import declarative_base
4
5  Base = declarative_base()
6
7  class Model(Base):
8      __tablename__ = 'Model'
9
10     id = Column(Integer, primary_key=True, index=True, autoincrement=True)
11     tag = Column(String(80), primary_key=True)
12     max_value = Column(Float)
13     min_value = Column(Float)
14     threshold = Column(Float)
15     accuracy = Column(Float)
16     precision = Column(Float)
17     recall = Column(Float)
18     model_weights = Column(LargeBinary)
```

Código D.2 – API

```
1  import requests
2  from datetime import datetime
3  import logging
4
5  class FastAPIClient:
6      def __init__(self):
7          self.base_url = "http://localhost:5000"
8          self.host = "localhost"
9          self.port = "5000"
10
11     def set_host(self, host):
12         self.host = host
13         self.base_url = f"http://{self.host}:{self.port}"
14
15     def set_port(self, port):
16         self.port = port
17         self.base_url = f"http://{self.host}:{self.port}"
```

```
18
19      def send_api_post(self, route, data):
20          route = route.lstrip('/')
21          try:
22              response = requests.post(f"{self.base_url}/{route}", json=data)
23              response.raise_for_status()
24              return response.json()
25          except requests.exceptions.RequestException as e:
26              logging.error(f"Erro ao enviar post para a API: {str(e)}")
27              return None
28
29      def get_base_url(self):
30          return self.base_url
```

Código D.3 – ECG

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3  import pandas as pd
4  from database import *
5  from sqlalchemy import text
6
7  import tensorflow as tf
8  from sklearn.metrics import accuracy_score, precision_score, recall_score
9  from sklearn.model_selection import train_test_split
10 from models.autoencoder import AnomalyDetector
11 autoencoder = AnomalyDetector()
12
13 class ECG:
14     def get_ECG_inference_data(self):
15         try:
16             ecg_query = text("SELECT * FROM Inference_ECG ORDER BY dt_measure DESC
       LIMIT 1")
17             dataframe = pd.read_sql(ecg_query, engine_Pipelines)
18             dataframe = dataframe.iloc[:, :-2]
19             raw_data = dataframe.values
20             logger.info("Dados diários para inferência obtidos da base de dados")
21             return raw_data
22
23         except Exception as e:
24             logger.error(f"Erro ao obter ECG's diários da base de dados para inferê
       ncia: {str(e)}")
25     def get_ECG_train_data(self, begin_date, end_date):
26         try:
27             ecg_query = text(f"SELECT * FROM ECG WHERE dt_measure BETWEEN '{
       begin_date}' AND '{end_date}'")
28             dataframe = pd.read_sql(ecg_query, engine_Pipelines)
29             dataframe = dataframe.iloc[:, :-2]
```

```
30              raw_data = dataframe.values
31              logger.info("Dados obtidos da base de dados com sucesso")
32              return raw_data
33
34          except Exception as e:
35              logger.error(f"Erro ao obter ECG's da base de dados: {str(e)}")
36      def get_labels(self, raw_data):
37          labels = raw_data[:, -1]
38          return labels
39      def get_ecg_points(self, raw_data):
40          data = raw_data[:, 0:-1]
41          return data
42
43      def get_train_test_split(self, data, labels, test_size, random_state):
44          train_data, test_data, train_labels, test_labels = train_test_split(data,
        labels, test_size=test_size, random_state=random_state)#(data, labels,
        test_size=0.2, random_state=21)
45          return train_data, test_data, train_labels, test_labels
46
47      def get_min_max_val(self, train_data):
48          min_val = tf.reduce_min(train_data)
49          max_val = tf.reduce_max(train_data)
50          return max_val, min_val
51
52      def normalize_data(self, data, max_val, min_val):
53          data = (data - min_val) / (max_val - min_val)
54          data = tf.cast(data, tf.float32)
55          return data
56
57      def normal_train_test_generate(self, train_data, test_data, train_labels,
        test_labels):
58              train_labels = train_labels.astype(bool)
59              test_labels = test_labels.astype(bool)
60              normal_train_data = train_data[train_labels]
61              normal_test_data = test_data[test_labels]
62              return normal_train_data, normal_test_data
63
64      def anomalous_train_test_generate(train_data, test_data, train_labels,
        test_labels):
65          anomalous_train_data = train_data[~train_labels]
66          anomalous_test_data = test_data[~test_labels]
67          return anomalous_train_data, anomalous_test_data
```

APÊNDICE E – FASTAPI

A parte de aplicação da FastAPI, foi dividida em dois *scripts*, sendo um o principal, e o outro contendo a logica dos *endpoints*.

Código E.1 – Aplicação principal da FastAPI

```python
from fastapi import FastAPI

from database import *
from models.ecg import *
from models.predictions import *
from schemas.ecg import *
from schemas.predictions import *
from routes.results_route import router

logging.config.fileConfig('/app/logging.conf')
logger = logging.getLogger('fastapi')

app = FastAPI()
@app.on_event("startup")
def startup():
    logger.info("FastAPI iniciada com sucesso!")
    print("FastAPI iniciada com sucesso!")
    try:
        InferenceECG.metadata.create_all(bind=engine)
        Predictions.metadata.create_all(bind=engine)
    except Exception as e:
        print("Erro ao criar as tabelas do banco de dados:", e)
        logger.error("Erro ao criar as tabelas do banco de dados: %s", str(e))
@app.on_event("shutdown")
def shutdown():
    logger.info("Encerrando a aplicação e fechando a sessão do banco de dados.")
app.include_router(router)
```

Código E.2 – Endpoints API

```python
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from models.ecg import InferenceECG
from models.predictions import Predictions
from schemas.ecg import InferenceECGCreateSchema
from schemas.predictions import PredictionsCreateSchema
import logging.config
from database import get_session
```

```
logging.config.fileConfig('/app/logging.conf')
logger = logging.getLogger('fastapi')

router = APIRouter()
@router.post("/predictions", response_model=PredictionsCreateSchema)
def create_prediction(prediction_data: PredictionsCreateSchema, db: Session = Depends(get_session)):
    try:
        prediction = Predictions(**prediction_data.dict())
        db.add(prediction)
        db.commit()
        db.refresh(prediction)
        logger.info(f"Predição criada com sucesso: {prediction}")
        #return prediction
    except Exception as e:
        import traceback
        traceback.print_exc()  # Imprime o traceback completo no console
        logger.error(f"Erro ao criar a Predição: {str(e)}")
        raise HTTPException(status_code=500, detail="Erro ao criar a Predição")
@router.post("/ecg", response_model=InferenceECGCreateSchema)
def create_inference_ecg(inference_data: InferenceECGCreateSchema, db: Session = Depends(get_session)):
    try:
        inference_ecg = InferenceECG(**inference_data.dict())
        db.add(inference_ecg)
        db.commit()
        db.refresh(inference_ecg)
        logger.info("ECG criado com sucesso")
        #return inference_ecg
    except Exception as e:
        import traceback
        traceback.print_exc()  # Imprime o traceback completo no console
        logger.error(f"Erro ao criar a Inferência ECG: {str(e)}")
        raise HTTPException(status_code=500, detail="Erro ao criar a Inferência ECG")
```

APÊNDICE F – SCHEMAS FASTAPI

Dois *scripts* são usados para definir a estrutura dos dados que serão recebidos pela API, representando o ECG original e suas predições pelo modelo.

Código F.1 – Schema ECG original

```
from datetime import datetime
from pydantic import BaseModel, ValidationError, validator
from typing import List

class InferenceECGCreateSchema(BaseModel):
    values: List[float]
    dt_measure: datetime
    is_anomalous: int
    model_tag: str
```

Código F.2 – Schema Predições

```
from datetime import datetime
from pydantic import BaseModel, ValidationError, validator
from typing import List

class PredictionsCreateSchema(BaseModel):
    values: List[float]
    dt_measure: datetime
    model_tag: str
```

APÊNDICE G – MODELS FASTAPI

Dois *scripts* são usados para representar as entidades de dados mapeados para as respectivas tabelas do banco de dados.

Código G.1 – Model ECG original

```python
from sqlalchemy import Column, Integer, Float, DateTime, Boolean, String
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class InferenceECG(Base):
    __tablename__ = 'Inference_ECG'

    id = Column(Integer, primary_key=True, autoincrement=True)
    dt_measure = Column(DateTime, nullable=True)
    is_anomalous = Column(Integer, default=False)
    model_tag = Column(String(80))  # Tag do modelo

    # Adicionando colunas para os números 0 até 139
    for i in range(140):
        locals()[str(i)] = Column(Float, default=None)

    def __init__(self, dt_measure, is_anomalous, model_tag, values):
        self.dt_measure = dt_measure
        self.is_anomalous = is_anomalous
        self.model_tag = model_tag
        for i, value in enumerate(values):
            setattr(self, str(i), value)

    def __repr__(self):
        return f'<InferenceECG(id={self.id}, dt_measure={self.dt_measure},
    anomalous={self.anomalous}, tag={self.model_tag}, ...)>'
```

Código G.2 – Model Predições

```python
from sqlalchemy import Column, Integer, Float, DateTime, String
from sqlalchemy.ext.declarative import declarative_base
from datetime import datetime

Base = declarative_base()

class Predictions(Base):
    __tablename__ = 'Predictions'
```

```
9
10      id = Column(Integer, primary_key=True, autoincrement=True)
11      dt_measure = Column(DateTime, nullable=True)
12      model_tag = Column(String(80))
13
14      for i in range(140):
15          locals()[f'value_{i}'] = Column(Float, default=None)
16
17      def __init__(self, dt_measure, model_tag, values):
18          self.dt_measure = dt_measure
19          self.model_tag = model_tag
20          for i, value in enumerate(values):
21              setattr(self, f'value_{i}', value)
22      def __repr__(self):
23          return f'<Predictions(id={self.id}, timestamp={self.dt_measure}, tag={self.
    model_tag}, value_0={self.value_0}, value_1={self.value_1}, ...)>'
```