Instituto Tecnológico de Aeronáutica

CSC-64 — Processamento Distribuído

Laboratório Exame: Ensemble Learning Distribuído

Professores:
Juliana de Melo Bezerra Celso Hirata

Alunos:
Denys Derlian Carvalho Brito
Lucas Silva Lima
João Lucas Rocha Rolim
Rafael Hoffmann Giannico
Samir Nunes da Silva

10 de dezembro de 2024

Denys Derlian
João Lucas
Lucas Lima
Rafael Hoffmann
Samir Nunes

CSC-27 — Lab Exame

Resumo

Este trabalho apresenta o desenvolvimento de uma aplicação distribuída de aprendizado de máquina, utilizando técnicas de ensemble learning e comunicação via RPC (Remote Procedure Call) e REST APIs. O objetivo principal foi abordar os desafios de escalabilidade e coordenação em sistemas distribuídos, permitindo o treinamento eficiente de modelos em grandes volumes de dados particionados entre vários nós.

A solução proposta distribui o treinamento de weak learners entre diversos nós e utiliza um processo central para a agregação das predições, garantindo consistência e eficiência. Os resultados obtidos mostram uma redução significativa no tempo de execução, com um desempenho 36,5% superior ao método sequencial. Além disso, foram implementados mecanismos de tolerância a falhas para garantir robustez ao sistema. Este projeto demonstra a viabilidade de utilizar sistemas distribuídos para aprendizado de máquina, oferecendo um modelo escalável e eficiente para lidar com grandes volumes de dados.

Denys Derlian
João Lucas
Lucas Lima
Rafael Hoffmann
Samir Nunes

CSC-27 — Lab Exame

Conteúdo

Denys Derlian
João Lucas
Lucas Lima
Rafael Hoffmann
Samir Nunes

1 Motivação

O avanço das aplicações de aprendizado de máquina (ML) trouxe a necessidade de lidar com grandes volumes de dados e altas demandas computacionais. No entanto, sistemas centralizados enfrentam limitações significativas em cenários que exigem processamento em larga escala.

Sistemas distribuídos surgem como uma solução eficiente para esses desafios, permitindo que a carga de trabalho seja dividida entre múltiplos nós de processamento. Isso possibilita o treinamento e a inferência de modelos de aprendizado de máquina de forma escalável, atendendo às demandas de aplicações reais, como sistemas de recomendação, análise de grandes volumes de dados financeiros e detecção de fraudes.

O desafio principal reside na coordenação entre os nós para garantir consistência, eficiência e robustez durante a execução. Este projeto foi motivado pela necessidade de explorar técnicas de aprendizado de máquina distribuído, utilizando RPC para comunicação entre os nós e REST APIs para interação com o sistema, como uma alternativa eficiente para lidar com esses desafios.

2 Trabalhos Relacionados

O aprendizado de máquina distribuído é uma área amplamente explorada na literatura devido à crescente demanda por processamento em larga escala. Galakatos et al. (2017) definem aprendizado de máquina distribuído como sistemas e algoritmos projetados para operar em múltiplos nós, melhorando desempenho, escalabilidade e precisão. Além disso, destacam o uso de frameworks como Hadoop e Spark, que facilitam a execução de tarefas distribuídas.

Outro ponto relevante abordado por Galakatos et al. é a aplicação de ensemble learning em ambientes distribuídos. Essa técnica combina predições de múltiplos modelos fracos (*weak learners*) para criar um modelo robusto, utilizando estratégias como bagging para combinar saídas de diferentes nós.

Apesar dessas contribuições, há lacunas relacionadas à simplicidade de implementação e integração com sistemas existentes. Frameworks como Hadoop e Spark possuem alto custo de entrada devido à complexidade de configuração e manutenção. Nosso trabalho aborda essa lacuna ao implementar uma solução baseada em RPC, que é mais leve e facilmente integrável, mantendo a escalabilidade e eficiência.

Além disso, trabalhos práticos utilizando ferramentas como PyTorch Distributed Training Framework mostram como técnicas distribuídas podem ser aplicadas em aprendizado profundo. No entanto, essas soluções focam majoritariamente em deep learning, enquanto nosso projeto explora uma abordagem mais geral, aplicável a diferentes contextos de aprendizado de máquina.

3 Objetivo

O objetivo deste trabalho é desenvolver uma aplicação de aprendizado de máquina distribuído que permita o treinamento eficiente de modelos em ambientes com grandes volumes de dados particionados entre múltiplos nós de processamento. A proposta inclui:

- Implementar um sistema distribuído baseado em técnicas de *ensemble learning*, onde cada nó treina um modelo fraco (*weak learner*) em uma partição dos dados.
- Utilizar RPC (*Remote Procedure Call*) para coordenação eficiente entre os nós e comunicação de predições para um processo central.
- Integrar REST APIs para possibilitar interações externas com o sistema, facilitando o uso em aplicações práticas.
- Garantir escalabilidade e robustez no sistema, permitindo seu uso em diferentes contextos, como análise de dados em larga escala e treinamento de modelos de aprendizado de máquina distribuído.

Ao final, busca-se validar a eficiência da abordagem distribuída por meio de experimentos comparativos com métodos sequenciais, destacando as melhorias em tempo de execução e desempenho geral.

4 Proposta

4.1 Requisitos Funcionais

- Execução distribuída para treinamento e inferência.
- Coordenação eficiente entre os nós de processamento utilizando RPC.
- Uso de um proxy server para gerenciar as chamadas aos servidores de ML e fornecer uma REST API para o cliente.
- Agregação das predições dos modelos treinados em cada nó.
- Tratamento de erros para garantir a robustez da execução distribuída.

4.2 Requisitos Não Funcionais

- Escalabilidade: permitir a adição de novos nós sem degradar significativamente o desempenho.
- Robustez: garantir redundância e failover por meio de mecanismos de backup no proxy server.

4.3 Funções de Avaliação e Predição

O sistema implementa duas funções principais para coordenar as operações distribuídas:

- **Função de Predição (*Predict*):** Quando uma requisição de predição é recebida pelo proxy server, este realiza um *broadcast* das features para todos os servidores de ML por meio de RPC. Cada servidor executa localmente suas predições e envia os resultados de volta ao proxy. O proxy então agrega essas predições (por exemplo, calculando a média) e retorna o resultado final para o cliente por meio da REST API.
- **Função de Avaliação (*Evaluate*):** Durante o processo de avaliação, cada servidor de ML calcula métricas locais (como erro ou precisão) em seu conjunto de validação e envia os resultados para o proxy. O proxy combina essas métricas (por exemplo, tirando a média) para gerar uma visão consolidada do desempenho geral do sistema. Este processo é essencial para monitorar a qualidade dos modelos durante o treinamento.

Essas funções foram projetadas para serem assíncronas e escaláveis, aproveitando goroutines no proxy server para paralelizar as chamadas RPC aos servidores de ML.

4.4 Arquitetura Proposta

A arquitetura do sistema distribuído consiste nos seguintes componentes:

- **Proxy Server:** Desenvolvido em Go, este componente atua como intermediário entre os clientes e os servidores de aprendizado de máquina (ML servers). Ele gerencia as chamadas RPC para os servidores de ML usando goroutines para paralelizar as operações e fornecer uma REST API para os clientes.
- **Servidores de ML:** Implementados em Python, cada servidor treina um modelo fraco (*weak learner*) com uma porção dos dados disponíveis. Esses servidores também fornecem resultados de predição e métricas de avaliação ao proxy.
- **Comunicação via RPC:** O proxy server se comunica de forma assíncrona com os servidores de ML utilizando RPC, garantindo baixa latência e eficiência no processo.
- **REST API:** Disponibilizada pelo proxy server para que os clientes possam enviar requisições de predição ou avaliação e receber os resultados agregados.

A Figura 1 ilustra o esquema da arquitetura proposta, incluindo a interação entre os componentes.

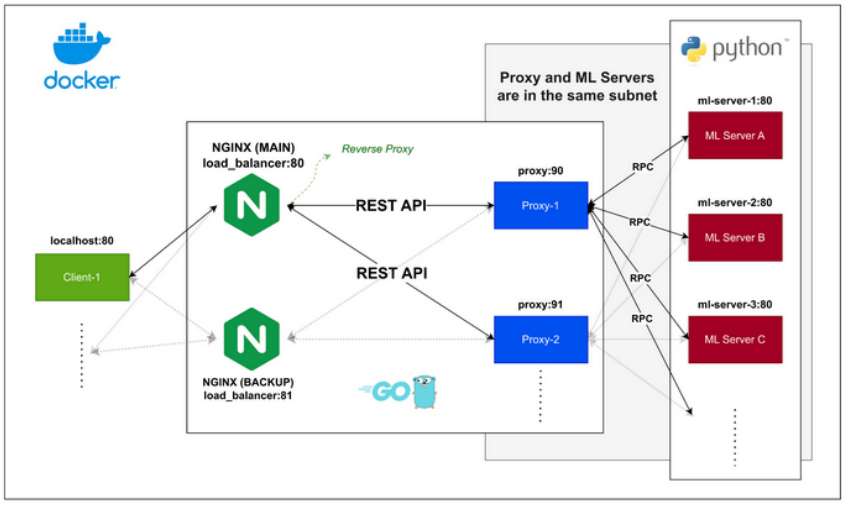

Figura 1: Esquema atualizado da arquitetura proposta.

Esta abordagem garante que o sistema seja escalável, eficiente e capaz de suportar

grandes volumes de dados, mantendo uma comunicação eficaz entre os componentes.

4.5 Tolerância a Falhas

O sistema foi projetado para ser robusto frente a falhas em seus componentes. As seguintes estratégias foram implementadas:

- **Tolerância a falhas nos servidores de ML:** Caso um servidor de ML falhe, o proxy server registra a falha em seu log e continua o processo corrente (seja *fit*, *evaluate* ou *predict*) utilizando apenas os resultados dos servidores restantes. Embora simples, esta abordagem permite que o sistema continue funcionando, mas pode impactar a qualidade geral das predições devido à ausência dos modelos dos servidores falhos.
- **Tolerância a falhas no proxy server:** Para evitar que uma falha no proxy server interrompa o sistema, foram configurados dois proxies redundantes gerenciados por um *load balancer* baseado em NGINX. Esse *load balancer* alterna as requisições entre os proxies e redireciona automaticamente as requisições caso um dos proxies falhe, garantindo alta disponibilidade e distribuição de carga.
- **Tolerância a falhas no balanceador de carga:** Embora o balanceador de carga NGINX ofereça alta disponibilidade, uma falha crítica no NGINX pode ser resolvida manualmente por meio de reinicialização ou redirecionamento direto das requisições ao proxy funcional. Este procedimento foi validado durante os testes, mostrando que a execução do sistema pode ser restaurada mesmo em cenários onde o balanceador de carga falha.

5 Protótipo

5.1 Ferramentas e Configuração do Ambiente

Para garantir a modularidade e a facilidade de execução do sistema, foram utilizadas as seguintes ferramentas e configurações:

- **Poetry:** Ferramenta utilizada para gerenciar as dependências dos servidores de ML implementados em Python. Essa abordagem simplificou o controle de versões e a instalação de bibliotecas em diferentes ambientes.
- **Docker:** Cada componente do sistema (servidores de ML, proxy server e cliente) foi encapsulado em containers Docker, o que facilitou a separação de dependências e a replicação do ambiente de execução.
- **Docker Compose:** Utilizado para orquestrar múltiplos containers, incluindo os servidores de ML, o proxy server e o cliente. Essa ferramenta permitiu inicializar o sistema completo com um único comando, simplificando o processo de testes e validação.
- **NGINX:** Configurado como *load balancer* para gerenciar múltiplos proxy servers, garantindo redundância e distribuição de carga.

5.2 Scripts de Automação

Para facilitar o teste e a validação do sistema, foram criados os seguintes scripts:

- **Geração de Dados de Teste:** Um script em Python foi desenvolvido para gerar automaticamente conjuntos de dados em formato JSON, permitindo testar cenários variados de predição e avaliação.
- **Configuração de Containers:** Scripts específicos para inicializar os containers Docker de maneira local (`--local`) ou em produção (`--prod`).

Essas ferramentas e configurações garantiram a portabilidade do sistema, permitindo sua execução em diferentes plataformas de forma consistente e eficiente.

5.3 Implementação

O sistema foi implementado utilizando as linguagens Python e Go, aproveitando suas respectivas forças em aprendizado de máquina e comunicação distribuída. Os principais componentes do protótipo incluem:

- **Nós distribuídos:** Cada nó foi implementado em Python, responsável por treinar um modelo fraco (*weak learner*) em uma partição dos dados locais.

- **Servidor central:** Desenvolvido em Go, este componente realiza a agregação das predições recebidas dos nós por meio de RPC.
- **APIs REST:** Implementadas para fornecer interfaces externas para enviar dados e requisitar inferências do sistema.

5.4 Implementação das Funções de Avaliação e Predição

As funções de *predict* e *evaluate* foram implementadas nos seguintes componentes:

- **Nos servidores de ML:** Cada servidor possui endpoints RPC dedicados para receber requisições de predição e avaliação, processá-las localmente e retornar os resultados ao proxy.
- **No proxy server:** O proxy utiliza goroutines para enviar requisições simultâneas a todos os servidores de ML. Após receber os resultados, realiza a agregação e fornece o resultado final ao cliente via REST API.

5.5 Tratamento de Erros

Foram implementados mecanismos para garantir a robustez do sistema:

- Detecção de nós inativos ou falhos durante a execução. Ao se detectar que um nó falhou, as informações daquele nó são descartadas.
- Detecção de proxy falho durante a execução. Neste caso o balanceador de cargas aloca as informações dos nós em outro proxy funcional.
- Detecção de erro no balanceador. Neste caso a mudança de balanceador pode ser feita manualmente, para que a execução continue.

5.6 Testes Realizados

Diversos testes foram realizados para validar o funcionamento e a robustez do sistema em cenários variados. Esses testes abrangeram desde a funcionalidade básica até situações de falha controlada.

5.6.1 Testes de Predição

5.6.1.1 Teste de *Fit* Para validar a funcionalidade de treinamento distribuído (*fit*), foi realizado um teste utilizando o endpoint correspondente do sistema. O comando POST foi enviado ao *proxy server*, que distribuiu os dados entre os servidores de ML para o treinamento. A resposta "Models trained successfully"confirmou o sucesso do processo de treinamento.

A Figura 2 apresenta o teste realizado utilizando uma ferramenta de requisições HTTP, destacando o tempo de resposta e a mensagem retornada pelo sistema.

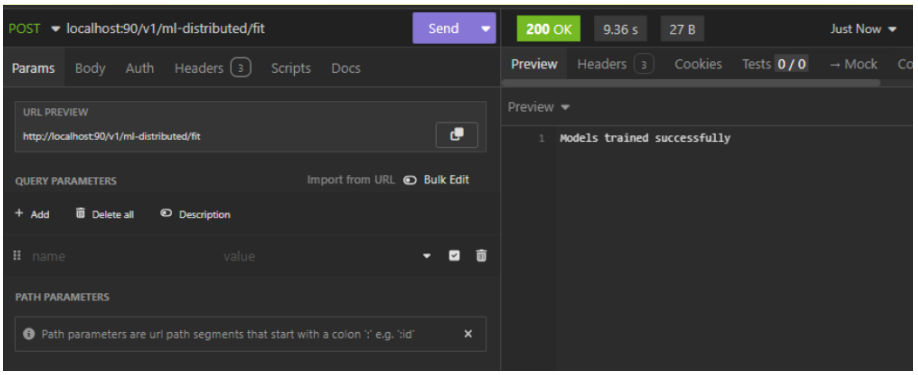

Figura 2: Teste do treinamento distribuído (*fit*) realizado com sucesso.

5.6.2 Testes de Avaliação

5.6.2.1 Teste de *Evaluate* Para validar a funcionalidade de avaliação (*evaluate*), foi realizado um teste utilizando o endpoint correspondente do sistema. O comando POST foi enviado ao *proxy server*, que coletou as métricas de avaliação de cada servidor de ML e agregou os resultados. A resposta retornou métricas como erro absoluto médio (*mean absolute error*) e o coeficiente R^2, confirmando a execução bem-sucedida do processo.

A Figura 3 apresenta o teste realizado, destacando os logs detalhados do terminal, onde é possível observar as interações entre o *proxy server* e os servidores de ML.

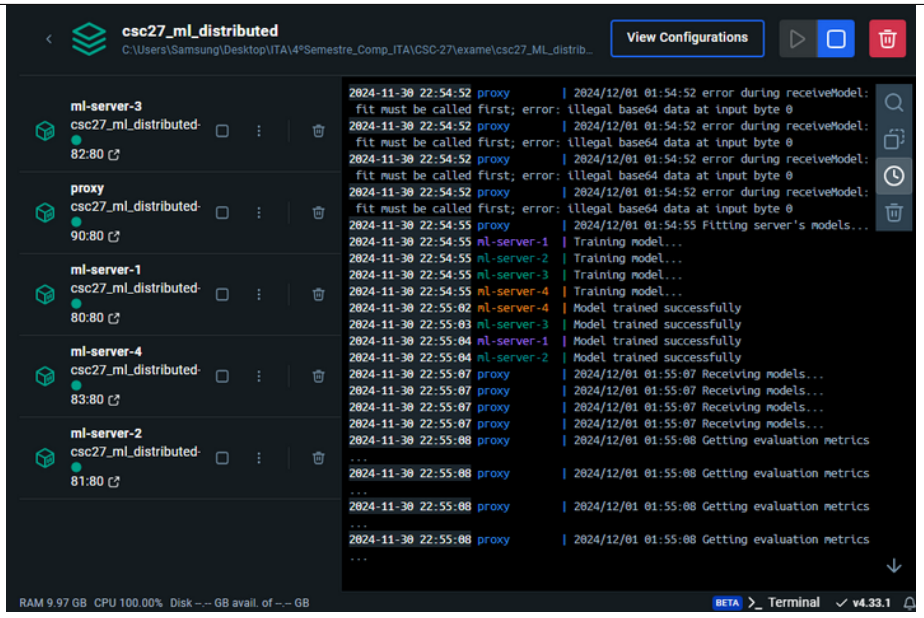

Figura 3: Teste de avaliação (*evaluate*) realizado com sucesso.

5.6.3 Testes de Tolerância a Falhas

Diversos testes foram realizados para validar a capacidade do sistema de lidar com falhas em diferentes componentes. Cada cenário é detalhado a seguir, com logs e representações gráficas dos testes realizados.

5.6.3.1 Teste 1: Falha em um Servidor de ML Neste teste, o servidor de ML *ml-server-2* foi desligado utilizando o comando `docker stop ml-server-2`. A Figura 4 apresenta o log do terminal que registra a falha, enquanto a Figura 5 mostra a representação gráfica da falha. O sistema registrou a ausência do servidor nos logs, mas continuou a realizar as operações utilizando os servidores restantes (*ml-server-1* e *ml-server-3*). Apesar de uma leve degradação no desempenho do modelo, o sistema permaneceu funcional.

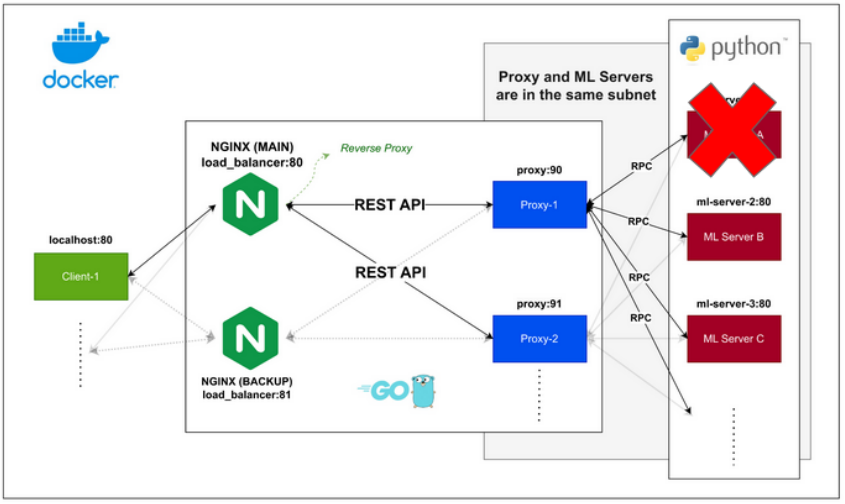

Figura 4: Log do terminal mostrando a falha no *ml-server-2*.

```
(base) denysderlian@DESKTOP-CFPH681:~/projects/csc27/exam/csc27_ML_distributed$
docker stop ml-server-2
ml-server-2
(base) denysderlian@DESKTOP-CFPH681:~/projects/csc27/exam/csc27_ML_distributed$
curl -X POST http://localhost:80/v1/ml-distributed/fit
Models trained successfully
(base) denysderlian@DESKTOP-CFPH681:~/projects/csc27/exam/csc27_ML_distributed$
curl -X POST http://localhost:80/v1/ml-distributed/evaluate
{"mean_absolute_error":34187.501688082884,"mean_squared_error":2554595652.261851
3,"mean_squared_log_error":0.06280747444350222,"r2_score":0.439572720919071}
(base) denysderlian@DESKTOP-CFPH681:~/projects/csc27/exam/csc27_ML_distributed$
curl -X POST http://localhost:80/v1/ml-distributed/predict -H "Content-Type: app
lication/json" -d @./data/house_pricing/other/one_prediction_house_pricing_test.
json
{"Prediction":119498.17697747484,"ProblemType":"regression"}
proxy-1      | 2024/12/03 02:43:01 error during fitting: Post "http://ml-server
2:80": dial tcp: lookup ml-server-2 on 127.0.0.11:53: no such host
proxy-2      | 2024/12/03 02:43:11 error during sending model: Post "http://ml-s
erver-2:80": dial tcp: lookup ml-server-2 on 127.0.0.11:53: no such host
proxy-2      | 2024/12/03 02:43:15 error during evaluation: Post "http://ml-serv
er-2:80": dial tcp: lookup ml-server-2 on 127.0.0.11:53: no such host
proxy-1      | 2024/12/03 02:43:47 error during prediction: Post "http://ml-serv
er-2:80": dial tcp: lookup ml-server-2 on 127.0.0.11:53: no such host
```

Figura 5: Representação gráfica da falha no *ml-server-2* e continuidade com os servidores restantes.

5.6.3.2 Teste 2: Falha no Proxy Principal Neste teste, o *Proxy-1* foi deliberadamente desligado utilizando o comando `docker stop proxy-1`. A Figura 6 exibe o log do terminal registrando a falha, enquanto a Figura 7 apresenta a representação gráfica do cenário. O sistema continuou funcional, com o *load balancer* redirecionando as requisições automaticamente para o *Proxy-2*. As operações de *fit*, *evaluate* e *predict* ocorreram normalmente, demonstrando a eficácia do mecanismo de failover do balanceador.

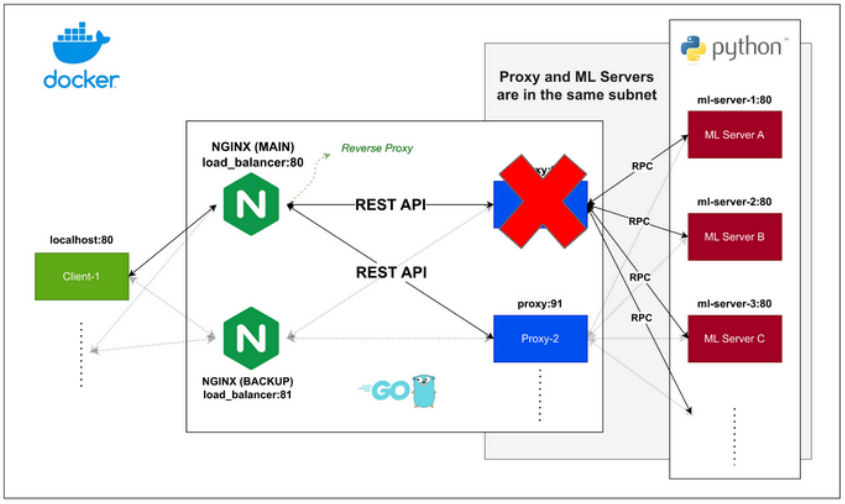

Figura 6: Log do terminal registrando o redirecionamento após a falha no *Proxy-1*.

Figura 7: Representação gráfica da falha no *Proxy-1* e redirecionamento para *Proxy-2*.

5.6.3.3 Teste 3: Falha no Balanceador de Carga Para simular uma falha no *load balancer* principal (*nginx-1*), este foi desligado utilizando o comando `docker stop nginx-1`. A Figura 8 exibe o log do terminal registrando a falha, e a Figura 9 apresenta a representação gráfica do cenário. As requisições falharam temporariamente, mas o sistema pôde ser manualmente reconfigurado para redirecionar as requisições diretamente ao *Proxy-2*, restaurando a funcionalidade.

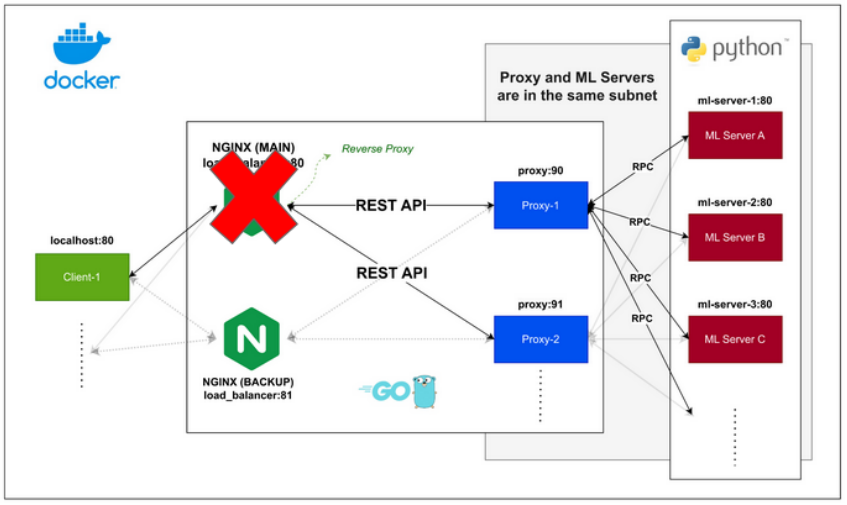

Figura 8: Log do terminal registrando a falha no *nginx-1*.

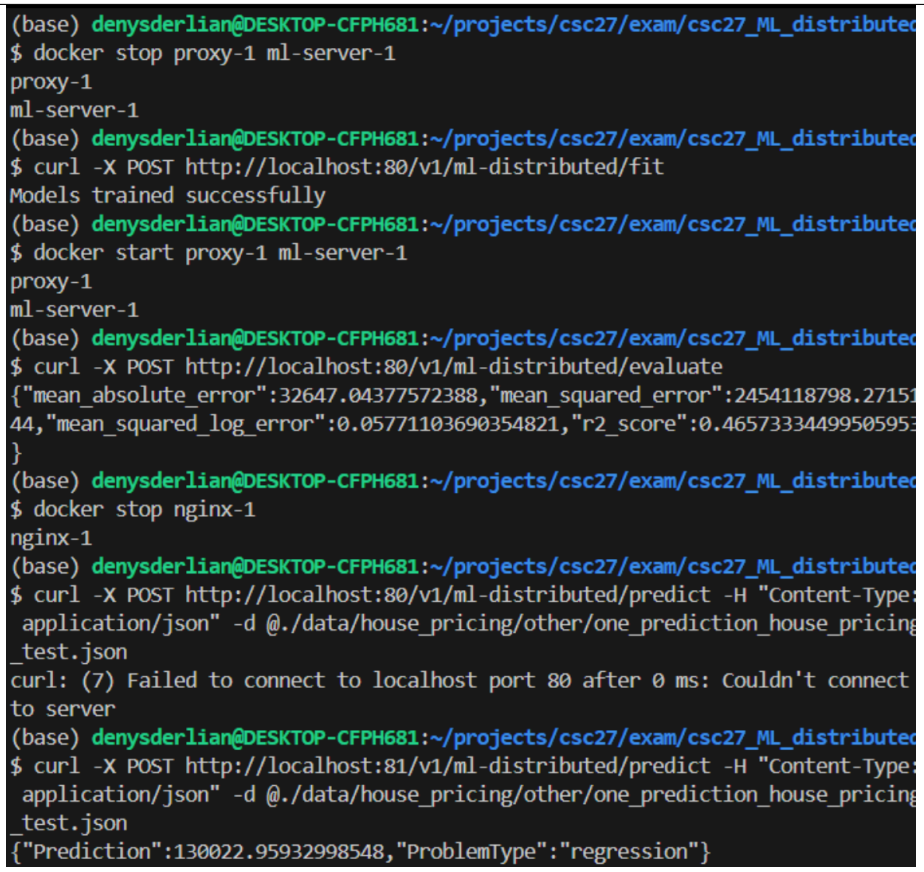

Figura 9: Representação gráfica da falha no *nginx-1* e reconfiguração manual.

5.6.4 Resultados dos Testes

Os testes realizados confirmaram a eficiência, robustez e escalabilidade da arquitetura distribuída proposta. A seguir, destacamos os principais resultados obtidos:

5.6.4.1 Desempenho Comparou-se o tempo de execução entre a solução sequencial e a solução distribuída. Como apresentado na Figura **??**, o tempo de execução sequencial foi de aproximadamente 12,6 minutos, enquanto o tempo de execução distribuído foi reduzido para 8 minutos, representando uma melhoria de 36,5%. Esse resultado valida o ganho de eficiência da abordagem distribuída.

```
PS C:\Users\Samsung\Desktop\ITA\4ºSemestre_Comp_ITA\CSC-27
\exame\csc27_ML_distributed\tests\server> poetry run pytho
n sequential.py
Sequential time: 756.705714225769
```

Figura 10: Tempo de execução da solução sequencial.

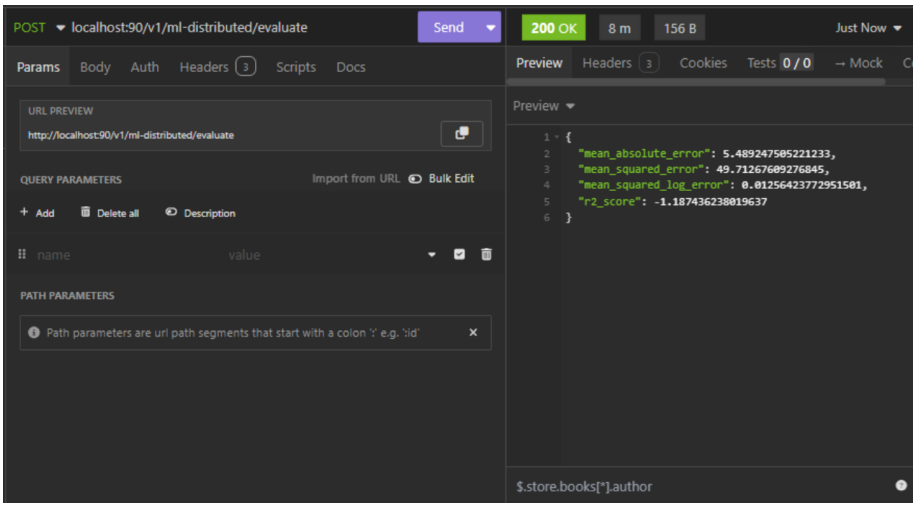

Figura 11: Tempo de execução da solução distribuída.

5.6.4.2 Robustez O sistema foi testado com conjuntos de dados contendo até 400.000 linhas por nó, totalizando mais de 10 GB de memória em uso. Apesar do consumo elevado de RAM, o sistema manteve-se funcional, demonstrando sua capacidade de lidar com grandes volumes de dados.

5.6.4.3 Coordenação entre Nós Os testes também demonstraram a capacidade do sistema de coordenar os processos de treinamento e predição distribuídos. A aplicação integrou os modelos treinados localmente em cada nó, garantindo consistência nos resultados finais.

5.6.4.4 Desafios Superados Durante os testes, foram enfrentados e resolvidos desafios relacionados a:

- Coordenação eficiente entre processos distribuídos utilizando RPC.
- Gerenciamento de memória, especialmente em cenários com grandes volumes de dados.

- Implementação de métodos de agregação para resultados consistentes.

5.6.4.5 Limitações Apesar dos resultados positivos, algumas limitações foram identificadas:

- Consumo elevado de memória, principalmente devido ao *multiprocessing* e aos parâmetros dos modelos.
- Infraestrutura limitada, restringindo o número de nós utilizados nos testes.
- Falta de suporte para paralelismo em GPUs, o que poderia melhorar ainda mais o desempenho.

5.6.4.6 Impacto Os resultados obtidos confirmam que a solução distribuída é uma alternativa viável e eficiente para aprendizado de máquina em larga escala. Além de demonstrar o potencial do *ensemble learning* em sistemas distribuídos, o projeto abre caminho para melhorias futuras, como o uso de GPUs, armazenamento externo para gerenciamento de modelos e bibliotecas especializadas em aprendizado distribuído.

5.7 Execução do Protótipo

A execução do sistema envolve os seguintes passos:

1. Certifique-se de que o Docker está instalado e em execução.
2. Utilize o comando `docker compose up` para iniciar os servidores de aprendizado de máquina (ML servers) e o proxy.

Após a configuração inicial, os logs indicarão que o proxy e os servidores foram iniciados com sucesso. A Figura 12 apresenta um exemplo desses logs gerados durante a inicialização.

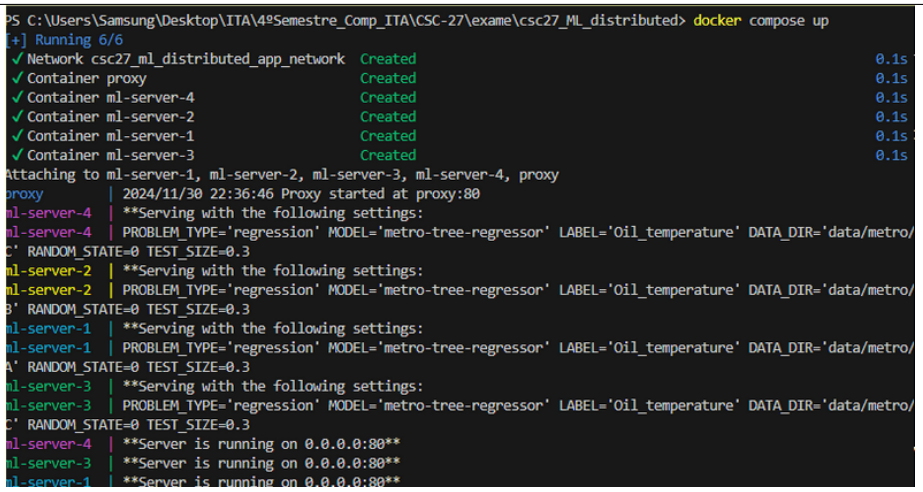

Figura 12: Logs do Docker mostrando a configuração do proxy e dos servidores de ML.

Os seguintes endpoints podem ser utilizados para testar o sistema:

- **(POST)** `localhost:90/v1/ml-distributed/fit`: Realiza o treinamento dos modelos distribuídos.
- **(POST)** `localhost:90/v1/ml-distributed/evaluate`: Avalia os modelos treinados, retornando métricas de desempenho.
- **(POST)** `localhost:90/v1/ml-distributed/predict`: Realiza predições com base nos modelos treinados. Para esta operação, é necessário fornecer os valores das *features* como parâmetros na requisição.

A Figura 13 apresenta um exemplo do uso dos endpoints com uma ferramenta de requisições HTTP, validando a execução do sistema.

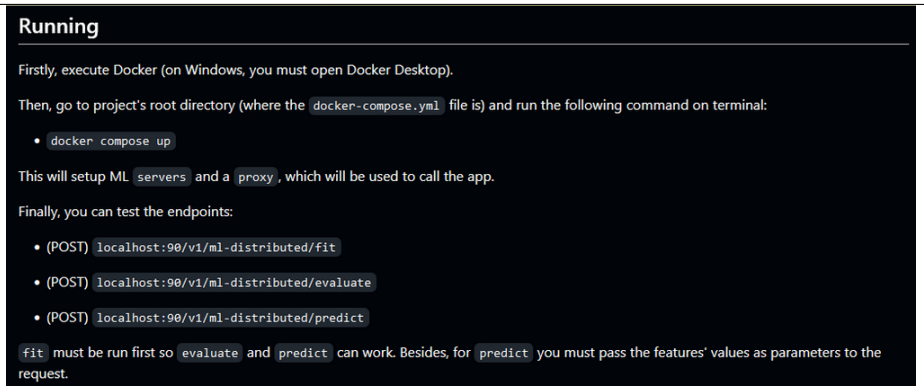

Figura 13: Teste de execução utilizando os endpoints do sistema.

É importante observar que o endpoint `fit` deve ser executado antes dos endpoints `evaluate` e `predict`, garantindo que os modelos estejam treinados previamente.

6 Conclusões

O desenvolvimento deste sistema distribuído de aprendizado de máquina demonstrou a viabilidade de combinar técnicas de *ensemble learning* com comunicação via RPC para lidar com grandes volumes de dados de forma eficiente. O trabalho alcançou os seguintes resultados principais:

- **Desempenho aprimorado:** O treinamento distribuído reduziu o tempo de execução em 36,5% em comparação à abordagem sequencial, validando a eficiência da arquitetura proposta.
- **Escalabilidade:** A abordagem baseada em nós distribuídos mostrou-se capaz de lidar com grandes conjuntos de dados, evidenciando seu potencial para expansão.
- **Robustez:** Foram implementados mecanismos de tratamento de erros que garantem a continuidade da execução mesmo em cenários de falhas nos nós.

Apesar dos resultados positivos, algumas limitações foram identificadas:

- O alto consumo de memória, devido ao uso de *multiprocessing* e parâmetros dos modelos, limita o treinamento em conjuntos de dados maiores.
- A infraestrutura disponível restringiu a quantidade de nós que poderiam ser utilizados, impactando diretamente na escalabilidade.
- A ausência de paralelismo em GPUs deixou espaço para melhorias no desempenho geral do sistema.

6.1 Trabalhos Futuros

Para continuar o desenvolvimento e aprimorar o sistema, sugerimos os seguintes passos:

- Adotar um *model store*, como o MLFlow, para reduzir o uso de memória e melhorar o gerenciamento dos modelos.
- Explorar paralelismo avançado utilizando GPUs e bibliotecas especializadas como TensorFlow Distributed ou PyTorch RPC.
- Ampliar a infraestrutura de testes para incluir mais nós, avaliando o impacto no desempenho e na escalabilidade.
- Implementar balanceamento dinâmico de carga entre os nós para aumentar a eficiência em cenários de dados heterogêneos.

Concluímos que o sistema desenvolvido é uma solução promissora para aprendizado de máquina distribuído, oferecendo eficiência, escalabilidade e robustez em cenários de alta demanda computacional. Seu impacto pode ser significativo em aplicações práticas, como análise de grandes volumes de dados e sistemas de recomendação.

7 Organização do Grupo

A distribuição das tarefas realizadas por cada integrante do grupo, incluindo as responsabilidades específicas e o percentual de contribuição foi dada por:

- Samir Nunes da Silva - Implementação do coordenador RPC e agregação de predições - 25%

- João Lucas Rocha Rolim - Desenvolvimento do módulo de treinamento distribuído e particionamento de dados - 20%

- Denys Derlian Carvalho Brito - Criação dos scripts de teste e análise comparativa de desempenho - 20%

- Lucas Silva Lima - Implementação do tratamento de erros e monitoramento dos nós - 20%

- Rafael Hoffmann Giannico - Criação dos diagramas de arquitetura e configuração do ambiente - 15%