Otimização de Modelos de Aprendizado Federado com Redes Definidas por Software

Cândido Leandro de Queiroga Bisneto

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

João Pessoa, 2025

Cândido Leandro de Queiroga Bisneto

Otimização de Modelos de Aprendizado Federado com Redes Definidas por Software

Monografia apresentada ao curso Ciência da Computação
do Centro de Informática, da Universidade Federal da Paraíba,
como requisito para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Fernando Menezes Matos

Dezembro de 2025

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

Trabalho de Conclusão de Curso de Ciência da Computação intitulado ***Otimização de Modelos de Aprendizado Federado com Redes Definidas por Software*** de autoria de Cândido Leandro de Queiroga Bisneto, aprovada pela banca examinadora constituída pelos seguintes professores:

Prof. Dr. Fernando Menezes Matos
Universidade Federal da Paraíba

Prof. [Nome do Professor B]
Universidade Federal da Paraíba

Prof. [Nome do Professor C]
Universidade Federal da Paraíba

Coordenador(a) do Curso de Ciência da Computação
[Nome do Coordenador]
CI/UFPB

João Pessoa, Dezembro de 2025

DEDICATÓRIA

[Texto da dedicatória - opcional]

AGRADECIMENTOS

[Texto dos agradecimentos]

RESUMO

[Inserir resumo em português - um único parágrafo informativo contendo: objetivos do trabalho, justificativa e resultados alcançados. Máximo 200 palavras.]

Palavras-chave: Aprendizado Federado, Redes Definidas por Software, XGBoost, LightGBM, CatBoost, Flower Framework, Privacidade de Dados.

ABSTRACT

[Insert abstract in English - one informative paragraph containing: work objectives, justification, and results achieved. Maximum 200 words.]

Keywords: Federated Learning, Software-Defined Networking, XGBoost, LightGBM, CatBoost, Flower Framework, Data Privacy.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

API - Application Programming Interface

AUC - Area Under the Curve

CART - Classification and Regression Trees

CPU - Central Processing Unit

DP - Differential Privacy

FL - Federated Learning

GBDT - Gradient Boosting Decision Trees

GPU - Graphics Processing Unit

gRPC - Google Remote Procedure Call

HTTP - Hypertext Transfer Protocol

IID - Independent and Identically Distributed

LGPD - Lei Geral de Proteção de Dados

ML - Machine Learning

non-IID - Non-Independent and Identically Distributed

QoS - Quality of Service

ROC - Receiver Operating Characteristic

SDN - Software-Defined Networking

TCC - Trabalho de Conclusão de Curso

TCP - Transmission Control Protocol

UFPB - Universidade Federal da Paraíba

UDP - User Datagram Protocol

Sumário

1 INTRODUÇÃO

1.1 Motivação

O Aprendizado Federado tem se consolidado como uma abordagem promissora para treinamento distribuído de modelos de aprendizado de máquina, permitindo que dispositivos colaborem no desenvolvimento de modelos globais sem compartilhar dados sensíveis. Esta característica torna-se especialmente relevante no contexto atual, onde questões de privacidade e proteção de dados pessoais ganham cada vez mais importância (POR REFERENCIA), tanto do ponto de vista regulatório quanto ético. No entanto, a maioria das pesquisas em FL concentra-se em redes neurais profundas, que demandam recursos computacionais significativos, incluindo grande capacidade de processamento, memória e, frequentemente, unidades de processamento gráfico especializadas (POR REFERENCIA).

Modelos baseados em árvore, como XGBoost, LightGBM e CatBoost, apresentam-se como alternativas mais leves e eficientes para diversos problemas de aprendizado de máquina. Estes algoritmos são capazes de serem treinados em hardwares mais modestos, sem a necessidade de GPUs dedicadas ou grandes volumes de memória RAM. Esta característica torna-os particularmente adequados para ambientes distribuídos heterogêneos, onde dispositivos podem possuir capacidades computacionais limitadas, como dispositivos móveis, sistemas embarcados ou computadores pessoais convencionais (POR REFERENCIA). A leveza computacional destes modelos não apenas reduz as barreiras de entrada para participação em sistemas federados, mas também amplia significativamente o espectro de dispositivos que podem contribuir para o treinamento colaborativo.

Além disso, a eficiência energética dos modelos baseados em árvore contribui para um treinamento mais sustentável. Em um cenário onde a pegada de carbono dos sistemas de aprendizado de máquina é crescentemente questionada, a redução do consumo energético durante o treinamento representa um avanço importante. Esta sustentabilidade é particularmente relevante em ambientes federados, onde múltiplos dispositivos treinam simultaneamente, potencialmente multiplicando o impacto ambiental. A adoção de modelos mais leves pode, portanto, contribuir para a viabilidade de sistemas FL em cenários reais, especialmente em regiões com infraestrutura energética limitada ou em aplicações que priorizam eficiência operacional.

Apesar destas vantagens, a comunicação entre clientes e servidor em sistemas FL representa um gargalo significativo. O overhead de rede, a latência na transmissão de parâmetros e a necessidade de sincronização frequente podem impactar negativamente o desempenho global do sistema (POR REFERENCIA). Neste contexto, a integração com Redes Definidas por Software emerge como uma solução potencial para otimizar a infraes-

trutura de comunicação, possibilitando gerenciamento dinâmico de tráfego e priorização inteligente de fluxos de dados (POR REFERENCIA).

1.2 Objetivo

Este trabalho tem como objetivo principal investigar a eficiência de modelos baseados em árvore, especificamente, XGBoost, LightGBM e CatBoost, em ambientes de Aprendizado Federado quando otimizados através do uso de Redes Definidas por Software (SDN). A escolha destes 3 algoritmos justifica-se por suas diferentes abordagens ao problema de gradient boosting, cada um com características particulares que podem apresentar comportamentos distintos em ambientes distribuídos.

—————— Especificamente, este estudo busca avaliar se a integração de SDN pode melhorar o desempenho destes modelos através de múltiplas dimensões. Primeiro, investiga-se a possibilidade de redução de overhead de comunicação mediante o gerenciamento inteligente de tráfego de rede, evitando congestionamentos e otimizando o uso de banda disponível. Segundo, analisa-se o potencial de otimização de latência através da priorização de fluxos críticos e seleção dinâmica de rotas de menor tempo de resposta. Terceiro, examina-se a viabilidade de implementação de políticas de Quality of Service (QoS) que priorizem inteligentemente o tráfego de rede entre diferentes clientes e o servidor central. ——————

A hipótese central que orienta esta pesquisa é que a combinação da leveza computacional dos modelos baseados em árvore com a capacidade de gerenciamento dinâmico de rede proporcionada por SDN pode resultar em sistemas FL significativamente mais eficientes, sustentáveis e escaláveis para aplicações práticas em diversos domínios. Esta hipótese fundamenta-se na premissa teórica e empiricamente fundamentada de que, ao reduzir simultaneamente os custos computacionais locais nos dispositivos clientes e os custos de comunicação na infraestrutura de rede distribuída, pode-se alcançar um sistema federado com desempenho superior ao obtido com abordagens convencionais que otimizam apenas uma destas dimensões. Mais especificamente, argumenta-se que a natureza menos intensiva em recursos dos modelos baseados em árvore, quando combinada com políticas inteligentes de roteamento e priorização de tráfego habilitadas por SDN, pode reduzir significativamente o tempo total de convergência do modelo global, minimizar o consumo de energia agregado do sistema, e melhorar a robustez do treinamento federado frente a condições adversas de rede, tais como alta latência, perda de pacotes, ou largura de banda limitada. Esta abordagem sinérgica representa uma contribuição metodológica importante para o campo do Aprendizado Federado, ao considerar de forma holística tanto os aspectos computacionais quanto os aspectos de infraestrutura de rede que influenciam o desempenho global do sistema.

Adicionalmente, este trabalho visa contribuir para o corpo de conhecimento em Aprendizado Federado ao explorar uma combinação ainda pouco investigada na literatura científica: modelos baseados em árvore otimizados com Redes Definidas por Software. Enquanto a maioria dos estudos existentes concentra-se primariamente em redes neurais profundas ou explora otimizações isoladas focando em apenas uma dimensão do problema (POR REFERENCIA), esta pesquisa propõe uma abordagem holística e integrada que considera simultaneamente tanto a eficiência computacional dos algoritmos de aprendizado quanto a eficiência de comunicação da infraestrutura de rede subjacente. Os resultados obtidos através desta investigação poderão orientar futuras implementações práticas de sistemas FL em ambientes com múltiplas restrições de recursos, sejam elas computacionais (processamento e memória), energéticas (consumo elétrico e autonomia de bateria), ou de infraestrutura de rede (largura de banda, latência e confiabilidade). Além disso, espera-se que este trabalho contribua para a identificação de melhores práticas e diretrizes de design para sistemas de Aprendizado Federado em ambientes reais, particularmente em contextos onde a heterogeneidade dos dispositivos participantes e a variabilidade das condições de rede representam desafios significativos para a convergência e qualidade do modelo global resultante. A abordagem metodológica adotada, combinando experimentação empírica com análise comparativa de diferentes algoritmos e estratégias de agregação, busca fornecer insights práticos que possam ser aplicados em cenários reais de produção, contribuindo assim para a transição do Aprendizado Federado de um paradigma predominantemente acadêmico para uma tecnologia madura e amplamente adotada na indústria.

1.3 Estrutura da monografia

Esta monografia está dividida da seguinte maneira: na Seção 2 é feita a revisão da literatura sobre aprendizado federado e redes SDN. Na Seção 3 são apresentados os fundamentos teóricos necessários para compreensão do trabalho. Na Seção 4 é apresentada a metodologia utilizada no desenvolvimento do trabalho. Na Seção 5 o dataset utilizado é descrito em detalhes. Na Seção 6 a configuração experimental é apresentada. Na Seção 7 os resultados obtidos são apresentados e analisados. Na Seção 8 é feita a discussão dos resultados. Na Seção 9 é feita a conclusão deste trabalho e trabalhos futuros são propostos.

2 REVISÃO DE LITERATURA

2.1 Trabalhos Relacionados sobre Aprendizado Federado

2.2 Trabalhos sobre Modelos Baseados em Árvores

2.3 Trabalhos sobre SDN e Otimização de Rede

2.4 Análise Comparativa dos Trabalhos

3 FUNDAMENTOS TEÓRICOS

Este capítulo apresenta os conceitos fundamentais que sustentam o desenvolvimento deste trabalho. Inicialmente, são discutidos os princípios do Aprendizado Federado, incluindo suas características essenciais, arquitetura cliente-servidor, formalização matemática do processo de otimização distribuída e as diferentes estratégias de agregação que determinam como os modelos locais são combinados para formar o modelo global. Em seguida, são detalhados os modelos baseados em árvore de decisão utilizados neste estudo: XGBoost, LightGBM e CatBoost, explorando suas formulações matemáticas, algoritmos de otimização, estratégias de regularização e características que os tornam adequados para ambientes de Aprendizado Federado. Por fim, é apresentado o framework Flower, ferramenta central para a implementação de sistemas de Aprendizado Federado, descrevendo sua arquitetura modular, protocolo de comunicação baseado em gRPC, interfaces de programação para clientes e servidores, e os modos de simulação e deployment que facilitam tanto a pesquisa acadêmica quanto aplicações em produção.

3.1 Aprendizado Federado

3.1.1 Definição e Características

O Aprendizado Federado (Federated Learning - FL) é um paradigma de aprendizado de máquina distribuído que permite o treinamento de modelos em dados descentralizados, sem a necessidade de centralizar ou compartilhar os dados brutos. Neste modelo, múltiplos dispositivos ou servidores (denominados clientes) colaboram para treinar um modelo global, mantendo seus dados localmente armazenados e privados. Esta abordagem representa uma mudança fundamental em relação ao paradigma tradicional de aprendizado centralizado, onde todos os dados são coletados e armazenados em um único servidor para treinamento. O Aprendizado Federado surgiu como resposta crescente a preocupações com privacidade de dados, regulamentações como GDPR (General Data Protection Regulation) e LGPD (Lei Geral de Proteção de Dados), e limitações práticas de transferência de grandes volumes de dados através da rede. Além de preservar a privacidade, o FL permite o aproveitamento de dados distribuídos em dispositivos edge, como smartphones, veículos autônomos, dispositivos IoT e sistemas médicos, sem violar restrições de privacidade ou incorrer em custos proibitivos de comunicação.

O fluxo de trabalho do Aprendizado Federado segue 5 etapas ilustradas nas Figuras 1 a 5:

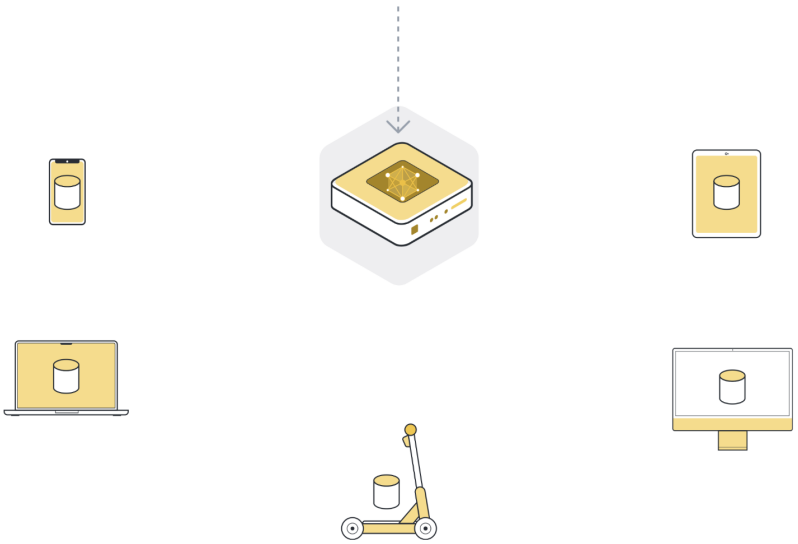

Figura 1: Inicialização do modelo global no servidor central

Na Figura 1, o servidor central inicializa um modelo global que serve como ponto de partida para o treinamento federado. Esta inicialização pode ser feita com parâmetros aleatórios, seguindo distribuições estatísticas apropriadas (como Xavier ou He initialization para redes neurais), ou pode utilizar modelos pré-treinados (transfer learning) quando disponíveis. A qualidade da inicialização pode impactar significativamente a velocidade de convergência e o desempenho final do modelo federado.

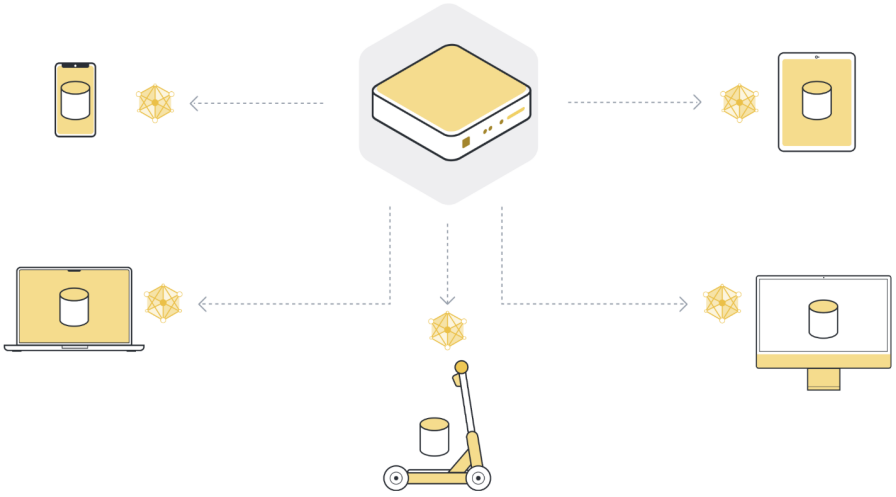

Figura 2: Distribuição do modelo global para os clientes

Na Figura 2, o modelo global é transmitido do servidor para os clientes selecionados através da serialização dos parâmetros do modelo. Este processo de broadcast envolve a conversão dos parâmetros do modelo em um formato adequado para transmissão pela rede (como arrays de bytes ou Protocol Buffers), seguido pela distribuição através de

protocolos de comunicação eficientes como gRPC. A serialização deve ser eficiente para minimizar o overhead de comunicação, especialmente em cenários com largura de banda limitada ou grande número de clientes participantes.

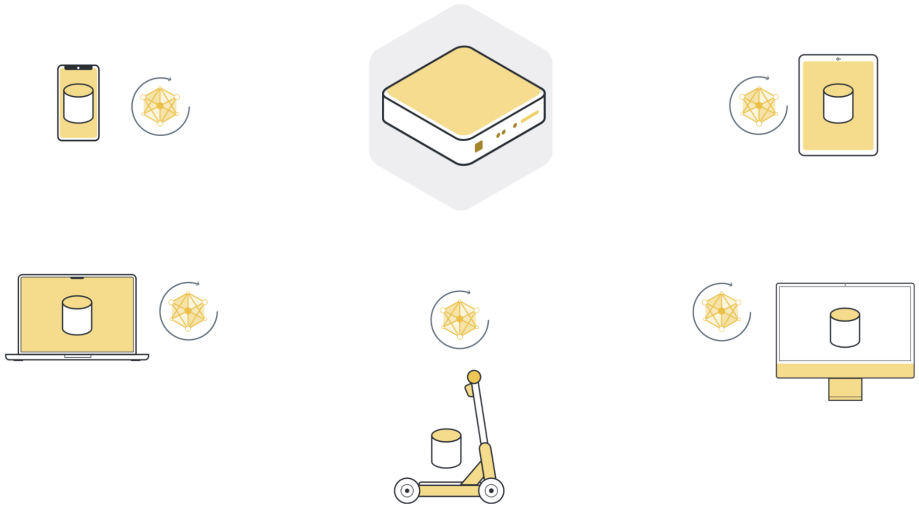

Figura 3: Treinamento local nos clientes com dados privados

Na Figura 3, cada cliente treina o modelo utilizando seus dados locais privados, sem que os dados saiam do dispositivo cliente. Durante esta fase, cada cliente executa múltiplas épocas de treinamento local usando algoritmos de otimização como SGD (Stochastic Gradient Descent), Adam ou outros otimizadores apropriados para o tipo de modelo. O número de épocas locais e o tamanho do batch são hiperparâmetros importantes que afetam tanto a qualidade do modelo atualizado quanto o custo computacional no cliente. Esta etapa é crucial para preservar a privacidade dos dados, pois nenhuma informação bruta é compartilhada com o servidor ou outros clientes.

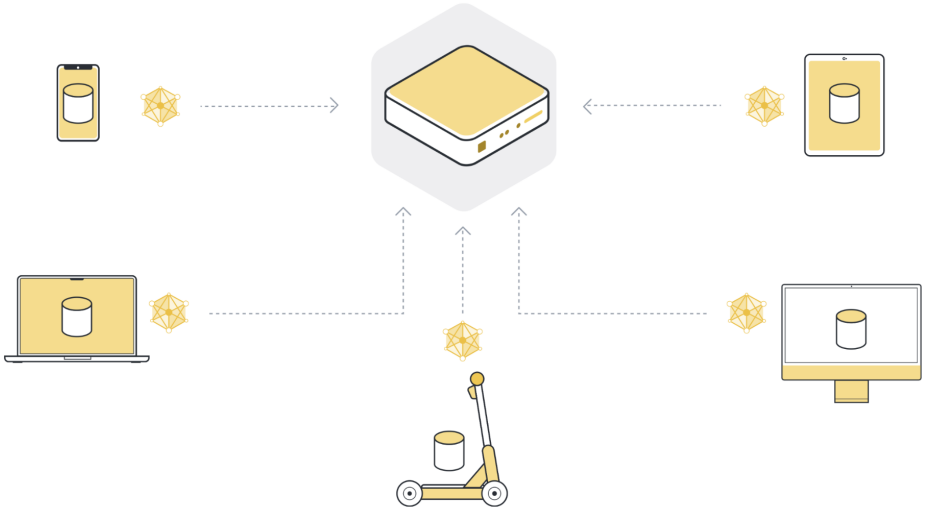

Figura 4: Envio das atualizações locais para o servidor

Na Figura 4, após o treinamento local, cada cliente envia suas atualizações de modelo (parâmetros treinados) de volta ao servidor central. As atualizações podem ser enviadas como parâmetros completos do modelo ou, em algumas implementações, como gradientes ou deltas em relação ao modelo inicial. Esta fase de upload representa um dos principais gargalos de comunicação em sistemas FL, especialmente para modelos grandes ou em ambientes com conectividade limitada. Técnicas de compressão, quantização e agregação parcial podem ser empregadas para reduzir o volume de dados transmitidos durante esta etapa.

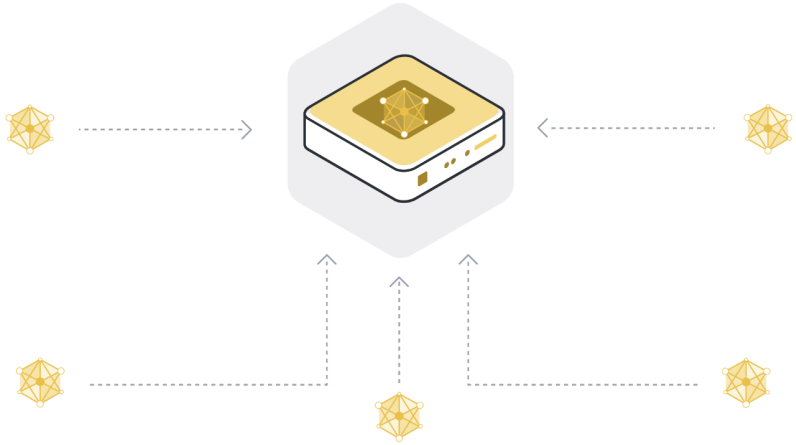

Figura 5: Agregação das atualizações no servidor

Na Figura 5, o servidor agrega as atualizações recebidas de todos os clientes para criar um novo modelo global melhorado. O ciclo se repete por múltiplas rodadas até a convergência. O processo de agregação é determinado pela estratégia escolhida (como FedAvg, FedProx, ou estratégias específicas para modelos baseados em árvore como Bagging e Cyclic), e pode envolver ponderação das atualizações baseada no número de amostras de cada cliente, qualidade dos dados, ou outros critérios. A convergência é tipicamente avaliada através de métricas como acurácia em um conjunto de validação centralizado, estabilização da função de perda, ou após um número pré-determinado de rodadas de comunicação. Este processo iterativo continua até que um critério de parada seja satisfeito, resultando no modelo global final que incorpora conhecimento de todos os clientes participantes sem ter acesso direto aos seus dados privados.

O processo básico do Aprendizado Federado pode ser descrito formalmente da seguinte forma. Considere N clientes, cada um com seu próprio conjunto de dados local $\mathcal{D}_i$, onde $i = 1, 2, \ldots, N$. O objetivo é aprender um modelo global w que minimize a função de perda agregada:

$$\min_{w} F(w) = \sum_{i=1}^{N} \frac{n_i}{n} F_i(w) \tag{1}$$

onde $F_i(w)$ é a função de perda local do cliente i, $n_i = |\mathcal{D}_i|$ é o número de amostras do cliente i, e $n = \sum_{i=1}^{N} n_i$ é o total de amostras em todos os clientes. A função de perda local é definida como:

$$F_i(w) = \frac{1}{n_i} \sum_{(x,y)\in\mathcal{D}_i} \ell(w; x, y) \tag{2}$$

onde $\ell(w; x, y)$ é a função de perda para uma amostra individual (x, y) com parâmetros do modelo w.

As 5 principais características do Aprendizado Federado que o diferenciam do aprendizado de máquina tradicional centralizado são:

Privacidade e Segurança: Os dados nunca saem dos dispositivos locais dos clientes. Apenas atualizações de modelo (gradientes ou parâmetros) são compartilhadas com o servidor central, preservando a privacidade dos dados sensíveis. Esta característica é especialmente importante em domínios como saúde, finanças e dispositivos móveis.

Comunicação Limitada: A troca de dados entre clientes e servidor é reduzida ao compartilhamento de parâmetros do modelo, ao invés de conjuntos de dados completos. Isso minimiza o overhead de comunicação, crucial em ambientes com largura de banda limitada.

Dados Não-IID: Os dados distribuídos entre os clientes frequentemente não são independentes e identicamente distribuídos (non-IID). Cada cliente pode ter uma distribuição de dados significativamente diferente dos demais, refletindo características locais específicas.

Desbalanceamento de Dados: O número de amostras pode variar drasticamente entre clientes. Alguns clientes podem ter milhares de amostras enquanto outros possuem apenas algumas dezenas.

Heterogeneidade de Dispositivos: Os clientes podem ter capacidades computacionais, de armazenamento e de rede muito diferentes. Esta heterogeneidade exige estratégias adaptativas para garantir a convergência do modelo.

O ciclo de treinamento federado típico segue 6 etapas:

1. **Inicialização:** O servidor central inicializa o modelo global $w^{(0)}$ com parâmetros aleatórios ou pré-treinados.

2. **Seleção de Clientes:** Em cada rodada t, o servidor seleciona um subconjunto $\mathcal{S}_t$ de clientes disponíveis para participar do treinamento.

3. **Broadcast:** O servidor envia os parâmetros atuais do modelo global $w^{(t)}$ para os

clientes selecionados.

4. **Treinamento Local:** Cada cliente $i \in \mathcal{S}_t$ realiza treinamento local usando seus dados $\mathcal{D}_i$, produzindo parâmetros atualizados $w_i^{(t+1)}$.

5. **Agregação:** O servidor coleta os modelos locais dos clientes e os agrega para produzir o novo modelo global $w^{(t+1)}$.

6. **Convergência:** O processo se repete até que um critério de convergência seja atingido (número de rodadas, precisão mínima, etc.).

3.1.2 Estratégias de Agregação

As estratégias de agregação são fundamentais no Aprendizado Federado, pois determinam como os modelos locais treinados pelos clientes são combinados para formar o modelo global. A escolha da estratégia de agregação impacta diretamente na convergência, precisão e robustez do sistema federado, influenciando aspectos como velocidade de convergência, qualidade do modelo final, tolerância a dados non-IID, consumo de recursos de comunicação e capacidade de lidar com clientes heterogêneos ou falhas de comunicação. Diferentes estratégias são mais adequadas para diferentes cenários de aplicação, tipos de modelos e características dos dados distribuídos. Neste trabalho, são exploradas 2 estratégias principais de agregação especialmente relevantes para modelos baseados em árvore: Bagging e Cíclica (Cyclic).

Estratégia Bagging

Na estratégia de Bagging, múltiplos clientes treinam modelos independentes em paralelo usando seus dados locais. Após o treinamento, os modelos são agregados no servidor central. Para modelos baseados em árvore de decisão, a agregação por Bagging é particularmente eficaz, pois permite combinar ensembles de árvores de diferentes clientes, explorando a diversidade dos dados locais para criar um modelo global mais robusto e generalizável. Esta estratégia é análoga ao tradicional Bootstrap Aggregating (Bagging) proposto por Breiman, mas adaptado para o contexto federado onde os dados já estão naturalmente particionados entre clientes. Considere que cada cliente i treina um modelo M_i com K árvores. A agregação pode ser realizada de 2 formas distintas, cada uma com suas vantagens e características:

Agregação de Árvores: As árvores individuais de todos os modelos locais são combinadas em um único ensemble:

$$M_{global} = \bigcup_{i \in \mathcal{S}} \text{Árvores}(M_i) \tag{3}$$

Agregação de Predições: Cada modelo local realiza predições independentes, e a predição final é obtida pela média (regressão) ou votação (classificação):

$$\hat{y}_{ensemble}(x) = \frac{1}{|\mathcal{S}|} \sum_{i \in \mathcal{S}} M_i(x) \tag{4}$$

Estratégia Cíclica (Cyclic)

Na estratégia Cíclica, diferentemente do treinamento paralelo do Bagging, os clientes treinam sequencialmente. Em cada rodada t, apenas um cliente i_t é selecionado para treinar, e o modelo é passado de um cliente para o próximo em uma ordem determinada:

$$w^{(t+1)} = w_i^{(t+1)}, \quad \text{onde } i = t \mod N \tag{5}$$

O processo pode ser descrito em 4 etapas:

1. Cliente i_1 recebe o modelo inicial $w^{(0)}$ e treina localmente, produzindo $w_1^{(1)}$
2. Cliente i_2 recebe $w_1^{(1)}$, treina e produz $w_2^{(2)}$
3. O processo continua até todos os clientes participarem
4. O ciclo pode se repetir por múltiplas rodadas

Esta estratégia é vantajosa quando há limitações de recursos (memória, largura de banda) ou quando se deseja um processo de convergência mais gradual e controlado. Para modelos baseados em árvore, a estratégia cíclica permite que cada cliente adicione árvores incrementalmente ao modelo existente.

3.2 Modelos Baseados em Árvore de Decisão

Os modelos baseados em árvore de decisão são algoritmos de aprendizado de máquina que utilizam estruturas em forma de árvore para realizar predições. Estes modelos são particularmente eficazes para dados tabulares e têm sido amplamente utilizados em competições de ciência de dados devido à sua precisão, interpretabilidade e eficiência computacional.

Um modelo baseado em árvore de decisão faz predições dividindo recursivamente o espaço de características em regiões. Cada divisão é determinada por uma condição sobre uma característica específica. Para uma tarefa de regressão, a predição final para uma amostra x é dada por:

$$\hat{y}(x) = \sum_{j=1}^{J} w_j \mathbb{1}(x \in R_j) \tag{6}$$

onde J é o número de folhas na árvore, w_j é o valor de predição associado à folha j, R_j é a região definida pela folha j, e $\mathbb{1}(\cdot)$ é a função indicadora.

Os métodos de ensemble combinam múltiplas árvores de decisão para melhorar a precisão e generalização. Os 3 algoritmos de gradient boosting considerados neste trabalho (XGBoost, LightGBM e CatBoost) são baseados no princípio de boosting, onde árvores são adicionadas sequencialmente para corrigir os erros das árvores anteriores.

3.2.1 XGBoost

XGBoost (eXtreme Gradient Boosting) é um algoritmo de gradient boosting otimizado que se tornou uma das técnicas mais populares em aprendizado de máquina para dados tabulares. O algoritmo constrói um ensemble de árvores de decisão de forma aditiva, onde cada nova árvore é treinada para corrigir os erros residuais das árvores anteriores.

O modelo XGBoost para uma amostra x_i após K iterações é dado por:

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i) \tag{7}$$

onde f_k representa a k-ésima árvore de decisão. O objetivo do treinamento é minimizar a seguinte função de perda regularizada:

$$\mathcal{L}(\phi) = \sum_{i=1}^{n} \ell(\hat{y}_i, y_i) + \sum_{k=1}^{K} \Omega(f_k) \tag{8}$$

onde ℓ é uma função de perda diferenciável (ex: erro quadrático médio, log-loss), e $\Omega(f_k)$ é um termo de regularização definido por:

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \tag{9}$$

onde T é o número de folhas, w_j é o peso da folha j, γ penaliza a complexidade do modelo (número de folhas), e λ controla a regularização L2 nos pesos das folhas.

O algoritmo XGBoost utiliza uma abordagem de otimização aditiva. Na iteração t, uma nova árvore f_t é adicionada para minimizar:

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} \ell(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t) \tag{10}$$

Aplicando a expansão de Taylor de segunda ordem à função de perda:

$$\mathcal{L}^{(t)} \approx \sum_{i=1}^{n} [\ell(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) \tag{11}$$

onde $g_i = \partial_{\hat{y}_i^{(t-1)}} \ell(y_i, \hat{y}_i^{(t-1)})$ é o gradiente de primeira ordem e $h_i = \partial^2_{\hat{y}_i^{(t-1)}} \ell(y_i, \hat{y}_i^{(t-1)})$ é a Hessiana.

Removendo termos constantes e reorganizando:

$$\tilde{\mathcal{L}}^{(t)} = \sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w_j^2 \tag{12}$$

Definindo $I_j = \{i | q(x_i) = j\}$ como o conjunto de amostras atribuídas à folha j, podemos reescrever a função objetivo como:

$$\tilde{\mathcal{L}}^{(t)} = \sum_{j=1}^{T} [(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T \tag{13}$$

O peso ótimo para a folha j é obtido derivando em relação a w_j e igualando a zero:

$$w_j^* = -\frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda} \tag{14}$$

E a pontuação de qualidade correspondente para a estrutura da árvore é:

$$\tilde{\mathcal{L}}^{(t)} = -\frac{1}{2} \sum_{j=1}^{T} \frac{(\sum_{i \in I_j} g_i)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma T \tag{15}$$

Esta pontuação é usada para avaliar candidatas a divisão durante a construção da árvore. O ganho de uma divisão que separa o conjunto I em I_L (esquerda) e I_R (direita) é:

$$\text{Ganho} = \frac{1}{2} \left[\frac{(\sum_{i \in I_L} g_i)^2}{\sum_{i \in I_L} h_i + \lambda} + \frac{(\sum_{i \in I_R} g_i)^2}{\sum_{i \in I_R} h_i + \lambda} - \frac{(\sum_{i \in I} g_i)^2}{\sum_{i \in I} h_i + \lambda} \right] - \gamma \tag{16}$$

As principais características e inovações do XGBoost incluem:

Otimizações de Eficiência: XGBoost implementa diversas otimizações como column block para armazenamento paralelo e cache-aware access para melhorar a performance em hardware moderno.

Regularização: Tanto regularização L1 (α) quanto L2 (λ) podem ser aplicadas aos pesos das folhas para prevenir overfitting.

Suporte a GPU: Implementação nativa de algoritmos para treinamento em GPU, acelerando significativamente o processo.

Tratamento de Valores Ausentes: Árvores aprendem automaticamente a melhor direção para valores ausentes durante o treinamento.

3.2.2 LightGBM

LightGBM (Light Gradient Boosting Machine) é um framework de gradient boosting desenvolvido pela Microsoft que utiliza algoritmos baseados em árvores de decisão. A principal inovação do LightGBM é o uso de estratégias de crescimento de árvore mais eficientes, resultando em treinamento mais rápido e menor uso de memória.

Enquanto a maioria dos algoritmos de gradient boosting cresce árvores level-wise (nível por nível), o LightGBM utiliza uma estratégia leaf-wise (folha por folha):

Crescimento Level-wise: Divide todos os nós no mesmo nível, resultando em árvores balanceadas mas potencialmente ineficientes.

Crescimento Leaf-wise: Divide a folha que maximiza a redução de perda, mesmo que isso resulte em árvores desbalanceadas. Esta abordagem geralmente converge mais rapidamente:

$$\text{folha\_dividir} = \arg\max_{\text{folha}} \text{Ganho}(\text{folha}) \tag{17}$$

O LightGBM introduz 2 técnicas principais para melhorar a eficiência:

Gradient-based One-Side Sampling (GOSS)

GOSS reduz o número de amostras usadas para estimar o ganho de informação, mantendo aquelas com gradientes grandes (erros grandes) e amostrando aleatoriamente aquelas com gradientes pequenos. Para um conjunto de amostras com gradientes ordenados, GOSS:

1. Mantém as top-$a \times 100\%$ amostras com maiores gradientes (conjunto A)
2. Amostra aleatoriamente $b \times 100\%$ das amostras restantes (conjunto B)
3. Calcula o ganho de informação usando:

$$\text{Ganho} = \frac{1}{n}\left(\frac{(\sum_{x_i \in A} g_i + \frac{1-a}{b}\sum_{x_i \in B} g_i)^2}{n_l} + \frac{(\sum_{x_i \in A} g_i + \frac{1-a}{b}\sum_{x_i \in B} g_i)^2}{n_r}\right) \tag{18}$$

onde $\frac{1-a}{b}$ é um fator de correção para compensar a subamostragem.

Exclusive Feature Bundling (EFB)

EFB reduz a dimensionalidade agrupando características mutuamente exclusivas (features que raramente assumem valores não-zero simultaneamente). O problema de encontrar o agrupamento ótimo é NP-difícil, mas pode ser aproximado como um problema de coloração de grafos:

1. Construir um grafo onde cada característica é um vértice
2. Arestas conectam características com conflitos significativos
3. Aplicar algoritmo de coloração de grafos (greedy) para agrupar características

O número efetivo de características após EFB é:

$$d_{efetivo} = d_{original} - \sum_{i=1}^{k}(|B_i| - 1) \tag{19}$$

onde k é o número de bundles e $|B_i|$ é o tamanho do bundle i.

Discretização de Características

LightGBM utiliza histogramas para discretizar características contínuas em bins discretos. Para uma característica $x^{(j)}$, os valores são mapeados para b bins:

$$\text{bin}(x_i^{(j)}) = \min\{k : x_i^{(j)} \leq \text{threshold}_k\}, \quad k = 1, \ldots, b \tag{20}$$

Os gradientes são então acumulados por bin:

$$G_k = \sum_{i:\text{bin}(x_i^{(j)})=k} g_i \tag{21}$$

$$H_k = \sum_{i:\text{bin}(x_i^{(j)})=k} h_i \tag{22}$$

O ganho para uma divisão no threshold entre bins k e $k+1$ é calculado eficientemente como:

$$\text{Ganho}_k = \frac{(\sum_{i=1}^{k} G_i)^2}{\sum_{i=1}^{k} H_i} + \frac{(\sum_{i=k+1}^{b} G_i)^2}{\sum_{i=k+1}^{b} H_i} - \frac{(\sum_{i=1}^{b} G_i)^2}{\sum_{i=1}^{b} H_i} \tag{23}$$

Esta abordagem reduz a complexidade de encontrar a melhor divisão de $O(n \times d)$ para $O(b \times d)$, onde $b \ll n$.

As vantagens do LightGBM incluem velocidade de treinamento superior (especialmente em datasets grandes), menor uso de memória através de histogramas, e suporte nativo a características categóricas sem necessidade de one-hot encoding.

3.2.3 CatBoost

CatBoost (Categorical Boosting) é um algoritmo de gradient boosting desenvolvido pela Yandex que se diferencia por seu tratamento superior de características categóricas e por mecanismos que reduzem overfitting. O nome reflete sua principal inovação: o tratamento eficiente de variáveis categóricas sem a necessidade de pré-processamento extensivo.

Ordered Boosting

Uma limitação dos algoritmos tradicionais de gradient boosting é o chamado "prediction shift": o modelo usado para calcular gradientes é treinado nos mesmos dados, levando a um viés. CatBoost resolve isso com Ordered Boosting, que utiliza diferentes modelos para diferentes amostras.

Para cada amostra x_i, o CatBoost mantém um modelo M_i treinado apenas nas amostras que precedem x_i em uma permutação aleatória. O gradiente para x_i é calculado usando M_i:

$$g_i = \nabla_{\hat{y}} \ell(y_i, M_i(x_i)) \tag{24}$$

Na prática, manter n modelos separados é computacionalmente proibitivo. CatBoost utiliza uma aproximação onde múltiplas amostras compartilham o mesmo modelo, mas os modelos são atualizados sequencialmente seguindo a permutação:

$$M_i = M_{i-1} + \alpha \cdot \text{TreeUpdate}(x_{\sigma(i-1)}, g_{\sigma(i-1)}) \tag{25}$$

onde σ é uma permutação aleatória dos índices das amostras.

Target Statistics para Características Categóricas

CatBoost codifica características categóricas usando target statistics, calculando estatísticas do target baseadas em valores categóricos. Para uma característica categórica

c com valor v, a codificação é:

$$\text{TS}(c = v) = \frac{\sum_{i:c_i=v, i<j} y_i + a \cdot P}{\sum_{i:c_i=v, i<j} 1 + a} \tag{26}$$

onde:

- A soma considera apenas amostras que precedem a amostra atual j na permutação (evitando data leakage)
- P é uma estimativa prior do target (geralmente a média global)
- $a > 0$ é um parâmetro de suavização que controla o peso do prior

Para múltiplas permutações, CatBoost calcula a média das target statistics:

$$\text{TS}_{final}(c = v) = \frac{1}{p} \sum_{k=1}^{p} \text{TS}_k(c = v) \tag{27}$$

onde p é o número de permutações.

Oblivious Trees

CatBoost utiliza árvores simétricas (oblivious trees) como modelos base. Nesta estrutura, a mesma condição de divisão é aplicada em todos os nós do mesmo nível. Uma árvore oblivious de profundidade d pode ser representada como um vetor de d divisões:

$$T = (s_1, s_2, \ldots, s_d) \tag{28}$$

onde cada s_i é uma divisão binária. O número de folhas é 2^d, e cada caminho da raiz até uma folha é determinado unicamente por um vetor binário de decisões.

As vantagens das árvores oblivious incluem:

- **Redução de Overfitting:** A estrutura simétrica age como uma forma de regularização
- **Eficiência Computacional:** Predições podem ser calculadas muito rapidamente usando operações binárias
- **Facilidade de Interpretação:** A estrutura balanceada facilita a compreensão do modelo

Regularização e Penalidades

CatBoost implementa múltiplas formas de regularização:

Penalidade de Complexidade: Controla o tamanho das árvores:

$$\mathcal{L}_{reg} = \mathcal{L}_{loss} + \lambda \sum_{t=1}^{T} \Omega(f_t) \tag{29}$$

Bagging: Subamostragem de dados com reposição (Bayesian bootstrap):

$$P(x_i \text{ selecionado}) = 1 - (1 - p)^s \tag{30}$$

onde p é a probabilidade de subsample e s é o tamanho da amostra.

Random Strength: Adiciona ruído aleatório às pontuações de divisão para reduzir overfitting:

$$\text{score}_{modificado} = \text{score}_{original} - \text{random\_strength} \cdot \mathcal{N}(0, 1) \tag{31}$$

O CatBoost também suporta diferentes modos de crescimento de árvore (Ordered, Plain) e oferece predições robustas através do uso de múltiplas permutações durante o treinamento.

3.3 Redes Definidas por Software (SDN)

3.3.1 Arquitetura SDN

3.3.2 OpenFlow

3.3.3 Aplicações em Aprendizado de Máquina

3.4 Framework Flower

Figura 6: Arquitetura do Flower: framework agnóstico que suporta diversos tipos de clientes e algoritmos de ML

Flower (Federated Learning over the Wire) é um framework open-source para Aprendizado Federado que facilita a implementação e experimentação de sistemas FL. Desenvolvido para ser agnóstico em relação ao framework de machine learning subjacente (PyTorch, TensorFlow, scikit-learn, XGBoost, etc.), Flower oferece uma arquitetura flexível e escalável para pesquisa e aplicações em produção, conforme ilustrado na Figura 6.

O design do Flower é fundamentado em 3 princípios principais:

Flexibilidade: Suporte a diferentes frameworks de ML, estratégias de agregação e topologias de rede, permitindo experimentação com diversos algoritmos e configurações.

Extensibilidade: Arquitetura modular que permite fácil customização de componentes individuais (clientes, servidor, estratégias) sem modificar o núcleo do framework.

Simplicidade: API minimalista e intuitiva que reduz a complexidade de implementação de sistemas FL, tornando-os acessíveis a pesquisadores e desenvolvedores.

3.4.1 Arquitetura do Flower

A arquitetura do Flower segue um modelo cliente-servidor com comunicação baseada em gRPC (Google Remote Procedure Call), permitindo comunicação eficiente entre clientes e servidor através de diferentes redes e plataformas.

Componentes Principais

A arquitetura do Flower consiste em 3 componentes fundamentais:

Client (Cliente): Representa um participante do treinamento federado que possui dados locais. Cada cliente implementa a interface `Client` que define 2 métodos principais:

- `fit(parameters, config)`: Recebe parâmetros do modelo global, realiza treinamento local, e retorna parâmetros atualizados junto com métricas
- `evaluate(parameters, config)`: Avalia o modelo global usando dados locais e retorna métricas de desempenho

Formalmente, um cliente i executa a seguinte computação durante o treinamento:

$$w_i^{(t+1)} = \texttt{fit}(w^{(t)}, \mathcal{D}_i, \text{config}) \tag{32}$$

$$\text{métricas}_i = \texttt{evaluate}(w^{(t+1)}, \mathcal{D}_i^{val}) \tag{33}$$

Server (Servidor): Coordena o processo de treinamento federado, gerenciando clientes, distribuindo parâmetros e agregando atualizações. O servidor mantém o modelo global $w^{(t)}$ e orquestra as rodadas de treinamento:

[H] [1] Inicializar modelo global $w^{(0)}$ rodada $t = 1$ até T Selecionar clientes $\mathcal{S}_t \subseteq \mathcal{C}$ usando `configure_fit()` Enviar $w^{(t)}$ e configuração para clientes em $\mathcal{S}_t$ Receber atualizações $\{w_i^{(t+1)}\}_{i \in \mathcal{S}_t}$ dos clientes $w^{(t+1)} \leftarrow$ `aggregate_fit()`$(\{w_i^{(t+1)}\}_{i \in \mathcal{S}_t})$ (Opcional) Avaliar $w^{(t+1)}$ usando `evaluate()` **Retornar** $w^{(T)}$

Strategy (Estratégia): Define como os modelos locais são agregados no servidor e como os clientes são selecionados. As principais funções de uma estratégia são:

- `configure_fit(server_round, parameters, client_manager)`: Seleciona clientes para participar da rodada de treinamento e prepara configurações
- `aggregate_fit(server_round, results, failures)`: Agrega os resultados de treinamento dos clientes para produzir o modelo global
- `configure_evaluate(server_round, parameters, client_manager)`: Configura a avaliação distribuída
- `aggregate_evaluate(server_round, results, failures)`: Agrega métricas de avaliação dos clientes

Protocolo de Comunicação

Flower utiliza Protocol Buffers e gRPC para comunicação eficiente entre clientes e servidor. O protocolo define mensagens estruturadas para troca de parâmetros e metadados:

- `Parameters`: Encapsula os parâmetros do modelo como arrays de bytes
- `FitIns`: Instrução de treinamento enviada do servidor para o cliente
- `FitRes`: Resultado do treinamento enviado do cliente para o servidor
- `EvaluateIns`: Instrução de avaliação
- `EvaluateRes`: Resultado da avaliação

A serialização de parâmetros segue o padrão:

$$\text{bytes} = \text{serialize}(w) \rightarrow \text{Parameters}(\text{tensors} = [\text{bytes}]) \tag{34}$$

$$w = \text{deserialize}(\text{Parameters.tensors}) \tag{35}$$

Simulação e Deployment

Flower oferece 2 modos de operação:

Simulation Mode: Simula múltiplos clientes em uma única máquina usando Ray para processamento distribuído. Útil para pesquisa e experimentação rápida:

$$\text{num\_supernodes} \times \text{num\_clients\_per\_supernode} = \text{total\_clients} \tag{36}$$

Deployment Mode: Clientes e servidor executam em processos ou máquinas separadas, comunicando-se através de rede. Adequado para ambientes de produção e cenários cross-device reais.

Ciclo de Vida de uma Rodada

O ciclo completo de uma rodada de treinamento no Flower envolve 7 etapas:

1. **Seleção:** Servidor executa `configure_fit()` da estratégia para selecionar clientes
2. **Broadcast:** Servidor envia `FitIns` contendo parâmetros globais para clientes selecionados
3. **Treinamento Local:** Cada cliente executa `fit()` e treina o modelo localmente
4. **Upload:** Clientes enviam `FitRes` contendo parâmetros atualizados e métricas
5. **Agregação:** Servidor executa `aggregate_fit()` para combinar atualizações
6. **Atualização:** Modelo global é atualizado com parâmetros agregados
7. **Avaliação:** Servidor ou clientes avaliam o novo modelo global

4 METODOLOGIA

4.1 Visão Geral da Solução Proposta

4.2 Arquitetura do Sistema

4.2.1 Componentes do Sistema

4.2.2 Fluxo de Comunicação

4.3 Implementação dos Modelos Federados

A implementação dos modelos federados segue rigorosamente o padrão de cliente-servidor do framework Flower [**?**], onde cada algoritmo (XGBoost [**?**], LightGBM [**?**] e CatBoost [**?**]) possui um cliente customizado que implementa a interface abstrata `Client` do Flower, definindo métodos específicos para treinamento e avaliação local. O fluxo de trabalho do Aprendizado Federado segue as 5 etapas sequenciais ilustradas nas Figuras 1 a 5 (Seção 3.1), onde o servidor inicializa o modelo global com parâmetros apropriados, distribui o modelo serializado para os clientes selecionados através de gRPC, aguarda que cada cliente realize treinamento local em seus dados privados, coleta as atualizações de modelo (parâmetros ou árvores) de todos os clientes participantes, e finalmente agrega essas atualizações usando a estratégia de agregação escolhida (Bagging ou Cyclic) para produzir um novo modelo global melhorado. Este ciclo se repete por múltiplas rodadas até atingir convergência ou um número máximo de iterações pré-definido.

Todos os algoritmos implementados neste trabalho compartilham uma estrutura comum de 2 métodos obrigatórios conforme especificado na Seção 3.4 sobre o framework Flower e ilustrado na arquitetura modular do Flower (Figura 6). Esta uniformidade na interface facilita a comparação direta entre algoritmos e garante compatibilidade com o ecossistema do Flower:

- `fit(ins: FitIns) -> FitRes`: Realiza treinamento local e retorna modelo atualizado
- `evaluate(ins: EvaluateIns) -> EvaluateRes`: Avalia modelo global com dados locais

4.3.1 Adaptação de XGBoost para FL

A implementação do cliente XGBoost para Aprendizado Federado está localizada no arquivo `algorithms/xgboost/client.py`, contendo a classe `XGBoostClient` que herda de `flwr.client.Client`. O cliente utiliza a estrutura de dados `DMatrix` nativa

do XGBoost [?] para otimizar o desempenho do treinamento, aproveitando o armazenamento otimizado em formato column-oriented e cache-aware access que reduz significativamente o uso de memória e acelera operações de divisão de nós durante a construção das árvores. Esta implementação foi especialmente adaptada para funcionar eficientemente em ambientes federados, lidando com serialização de modelos, treinamento incremental e comunicação através do protocolo Flower.

Serialização do Modelo

O XGBoost permite serialização direta e eficiente em formato JSON utilizando o método `save_raw()`, que converte a estrutura completa do modelo (árvores, parâmetros, features e metadados) em uma representação textual JSON compacta que pode ser facilmente transmitida através da rede. Esta abordagem é superior a métodos baseados em pickle, pois produz representações independentes de versão da biblioteca e compatíveis entre diferentes plataformas e linguagens de programação:

```
# Serialização (cliente -> servidor)
local_model = bst.save_raw("json")
local_model_bytes = bytes(local_model)

# Deserialização (servidor -> cliente)
global_model = bytearray(parameters.tensors[0])
global_bst = xgb.Booster(params=params)
global_bst.load_model(global_model)
```

Esta abordagem permite transferência eficiente de modelos através da rede sem necessidade de arquivos intermediários temporários no sistema de arquivos, reduzindo overhead de I/O e eliminando potenciais problemas de concorrência ou vazamento de arquivos temporários. O formato JSON também facilita debugging e inspeção manual do modelo quando necessário durante o desenvolvimento.

Treinamento Incremental

O treinamento federado no XGBoost suporta 2 modos distintos de operação, dependendo da rodada de comunicação atual, permitindo inicialização limpa na primeira rodada e refinamento incremental nas rodadas subsequentes:

Modo Inicial (Rodada 1): Treina modelo do zero usando parâmetros inicializados aleatoriamente:

```
bst = xgb.train(
    params,
    train_dmatrix,
```

```
    num_boost_round=num_local_round,
    evals=[(valid_dmatrix, "validate"), (train_dmatrix, "train")]
)
```

Modo Incremental (Rodadas 2+): Continua treinamento a partir do modelo global recebido:

```
bst = xgb.train(
    params,
    train_dmatrix,
    num_boost_round=num_local_round,
    xgb_model=global_bst,  # Modelo global como ponto inicial
    evals=[(valid_dmatrix, "validate"), (train_dmatrix, "train")]
)
```

O parâmetro `xgb_model` permite que o treinamento local adicione novas árvores ao modelo global existente, implementando efetivamente a estratégia de boosting distribuído.

Estratégias de Agregação

Para XGBoost, foram utilizadas as estratégias nativas do Flower implementadas em `flwr.server.strategy`:

- `FedXgbBagging`: Implementa agregação por Bagging, onde árvores de múltiplos clientes são combinadas em um único ensemble
- `FedXgbCyclic`: Implementa agregação Cíclica, onde clientes treinam sequencialmente adicionando árvores ao modelo

Avaliação de Métricas

O cliente calcula métricas abrangentes após cada rodada de treinamento utilizando o módulo `common/metrics.py`:

```
y_pred_proba = bst.predict(valid_dmatrix)
metrics = calculate_comprehensive_metrics(y_valid, y_pred_proba)
```

As métricas calculadas incluem: acurácia, precisão, recall, F1-score, AUC e matriz de confusão, com detecção automática para classificação binária ou multi-classe (conforme descrito na Seção 5.3.4).

4.3.2 Adaptação de LightGBM para FL

A implementação do cliente LightGBM está em `algorithms/lightgbm/client.py`. Diferentemente do XGBoost, o LightGBM [**?**] requer serialização baseada em arquivos temporários devido às limitações da API.

Serialização Baseada em Arquivos

O LightGBM não suporta serialização direta em memória, necessitando arquivos intermediários:

```
# Serialização (cliente -> servidor)
temp_save_path = f"/tmp/lgb_local_model_{client_id}_{round}.txt"
bst.save_model(temp_save_path)
with open(temp_save_path, 'rb') as f:
    local_model_bytes = f.read()
os.remove(temp_save_path)

# Deserialização (servidor -> cliente)
temp_model_path = f"/tmp/lgb_global_model_{client_id}_{round}.txt"
with open(temp_model_path, 'wb') as f:
    f.write(global_model_bytes)
bst = lgb.train(params, train_data, init_model=temp_model_path)
os.remove(temp_model_path)
```

Os arquivos temporários são imediatamente removidos após leitura/escrita para evitar acúmulo de dados no sistema.

Treinamento Incremental

Similar ao XGBoost, o LightGBM suporta treinamento incremental através do parâmetro `init_model`:

```
bst = lgb.train(
    params,
    train_data,
    num_boost_round=num_local_round,
    init_model=temp_model_path,  # Modelo global como base
    valid_sets=[valid_data],
    valid_names=['valid']
)
```

Estratégias Customizadas

Devido à falta de estratégias nativas no Flower para LightGBM, foram implementadas estratégias customizadas em `algorithms/lightgbm/server.py`:

- `LightGBMBaggingStrategy`: Herda de `FedAvg` e implementa agregação por Bagging
- `LightGBMCyclicStrategy`: Implementa seleção cíclica de clientes (1 cliente por rodada)

Ambas estratégias implementam os métodos `configure_fit()`, `aggregate_fit()` e `configure_evaluate()` para gerenciar o ciclo de treinamento federado.

Callback de Progresso

Foi implementado um callback customizado para visualização do progresso:

```
class VerboseCallback:
    def __call__(self, env):
        iteration = env.iteration
        metric_value = env.evaluation_result_list[0][2]
        print(f"[Cliente {client_id}] Round {round} | "
              f"Época {iteration+1}/{num_rounds} | Valid: {metric_value:.4f}")
```

4.3.3 Adaptação de CatBoost para FL

A implementação do cliente CatBoost está em `algorithms/catboost/client.py`. O CatBoost [**?**] utiliza a estrutura `Pool` para armazenamento eficiente de dados e também requer serialização baseada em arquivos.

Serialização em Formato CBM

O CatBoost utiliza formato binário proprietário CBM (CatBoost Model):

```
# Serialização (cliente -> servidor)
temp_save_path = f"/tmp/catboost_local_model_{client_id}_{round}.cbm"
model.save_model(temp_save_path, format='cbm')
with open(temp_save_path, 'rb') as f:
    local_model_bytes = f.read()
os.remove(temp_save_path)

# Deserialização (servidor -> cliente)
temp_model_path = f"/tmp/catboost_global_model_{client_id}_{round}.cbm"
with open(temp_model_path, 'wb') as f:
```

```
    f.write(global_model_bytes)
model = CatBoost(params)
model.load_model(temp_model_path)
os.remove(temp_model_path)
```

Treinamento Incremental

O CatBoost implementa treinamento incremental através do parâmetro `init_model`:

```
model = CatBoost(params)
model.load_model(temp_model_path)  # Carregar modelo global
model.fit(
    train_pool,
    eval_set=valid_pool,
    init_model=model,  # Continuar do modelo global
    verbose=10
)
```

Estratégias Customizadas

Similar ao LightGBM, foram implementadas estratégias customizadas em `algorithms/catboo`

- `CatBoostBaggingStrategy`: Implementa agregação por Bagging
- `CatBoostCyclicStrategy`: Implementa agregação Cíclica sequencial

Tratamento de Probabilidades

O CatBoost requer tratamento especial para predições de probabilidade em classificação binária:

```
y_pred_proba = model.predict(valid_pool, prediction_type='Probability')
# Para classificação binária, extrair probabilidade da classe positiva
if len(y_pred_proba.shape) == 2 and y_pred_proba.shape[1] == 2:
    y_pred_proba = y_pred_proba[:, 1]
```

Este tratamento garante compatibilidade com as funções de cálculo de métricas que esperam probabilidades unidimensionais para problemas binários.

4.4 Integração com SDN

4.4.1 Configuração da Rede SDN

4.4.2 Monitoramento de Tráfego

4.5 Tecnologias Utilizadas

Componente	**Tecnologia**
Framework FL	Flower 1.6+
Modelos	XGBoost, LightGBM, CatBoost
Linguagem	Python 3.9+
SDN Controller	[A definir]
Comunicação	gRPC

Tabela 1: Tecnologias utilizadas no projeto

5 DATASET E PREPARAÇÃO DOS DADOS

5.1 Origem e Geração do Dataset

Para avaliar o desempenho dos algoritmos de Aprendizado Federado neste trabalho, foi utilizado um dataset de comportamento de veículos gerado através de simulação de tráfego veicular realista. O dataset original foi criado utilizando um simulador de tráfego especializado que implementa o modelo de mobilidade IDM (Intelligent Driver Model), um modelo amplamente reconhecido na literatura de simulação de tráfego que captura comportamentos realistas de condução, incluindo aceleração, desaceleração, mudanças de faixa e interações entre veículos. A escolha de dados sintéticos gerados por simulação, ao invés de dados reais de tráfego, permite controle preciso sobre parâmetros experimentais, reprodutibilidade dos experimentos através de seeds aleatórias fixas, e evita questões de privacidade e regulamentações associadas a dados reais de veículos. O dataset completo está armazenado em:

`dataset_fl/dataset/dataset_K400_seed42/`

5.1.1 Configuração da Simulação

A simulação que gerou o dataset foi configurada com um conjunto detalhado de parâmetros que definem o ambiente de tráfego, características dos veículos e duração da observação. Estes parâmetros foram armazenados em um arquivo de configuração `simulation_config.json` para garantir reprodutibilidade e documentação completa dos experimentos. A configuração escolhida visa simular um cenário de rodovia realista com densidade moderada de tráfego e condições operacionais típicas:

- **Número de veículos (K)**: 400 veículos simulados
- **Rodovia**: 88 km de extensão com 3 faixas
- **Tempo de simulação**: 3.600 segundos (1 hora)
- **Intervalo de amostragem**: 1 segundo
- **Seed aleatória**: 42 (para reprodutibilidade)
- **Modelo de mobilidade**: IDM com parâmetros:
  - Velocidade desejada: 25 m/s (90 km/h)
  - Distância de segurança: 2 m
  - Tempo de reação: 1,5 s

- Aceleração máxima: 1,0 m/s^2
- Desaceleração confortável: 1,5 m/s^2

5.1.2 Estrutura Original dos Dados (JSON)

Os dados brutos da simulação foram salvos em formato JSON estruturado, com 1 arquivo independente por veículo. Esta organização facilita o particionamento natural dos dados por veículo no contexto do Aprendizado Federado, onde cada arquivo representa os dados coletados por um cliente individual (veículo). O formato JSON foi escolhido por sua legibilidade, facilidade de parsing e flexibilidade para armazenar estruturas de dados heterogêneas. A estrutura de diretórios é a seguinte:

```
private_datasets/
 veh_0.json
 veh_1.json
 veh_2.json
...
 veh_399.json
```

Cada arquivo JSON contém uma lista de observações temporais do veículo, onde cada observação possui:

- **features**: Array com 22 valores numéricos (estado do veículo e vizinhança)
- **label**: Classe de comportamento (0, 1 ou 2)

Exemplo de estrutura JSON:

```
[
  {
    "features": [21.18, 0, 21.23, 1, 385.34, 20.86, ...],
    "label": 1
  },
  {
    "features": [23.25, 0, 23.35, 1, 262.89, 25.26, ...],
    "label": 2
  },
  ...
]
```

5.2 Transformação para Formato Tabular

Os dados JSON foram transformados em formato CSV tabular para facilitar o treinamento dos modelos baseados em árvore de decisão (XGBoost, LightGBM e CatBoost), que operam de forma mais eficiente com estruturas tabulares do que com formatos hierárquicos como JSON. Este processo de transformação consolidou todos os 400 arquivos JSON individuais em 2 arquivos CSV estruturados (treino e validação), criando uma representação matricial dos dados onde cada linha corresponde a uma observação temporal de um veículo e cada coluna representa uma feature numérica. A transformação preserva todas as informações relevantes para o treinamento, incluindo metadados como identificadores de veículos necessários para o particionamento federado:

Característica	**Valor**
Nome	Dataset de Comportamento de Veículos
Formato original	400 arquivos JSON (1 por veículo)
Formato processado	2 arquivos CSV tabulares
Amostras de treino	115.511 observações
Amostras de validação	28.878 observações
Veículos	400 veículos únicos
Número de features	22 features numéricas
Classes	3 classes de comportamento (0, 1, 2)
Tipo de problema	Classificação multi-classe
Simulador	IDM (Intelligent Driver Model)
Seed	42 (para reprodutibilidade)

Tabela 2: Metadados do dataset de veículos utilizado nos experimentos

O processo de transformação JSON $\rightarrow$ CSV consolidou os dados de múltiplos arquivos JSON em 2 arquivos CSV estruturados:

- **Treino**: `dataset_all_vehicles.csv` (115.511 amostras de 400 veículos)
- **Validação**: `dataset_validation_all_vehicles.csv` (28.878 amostras)

Processo de conversão:

1. Leitura de cada arquivo JSON (`veh_*.json`) contendo observações temporais
2. Extração dos arrays `features` e valores `label` de cada observação
3. Adição de coluna `vehicle_id` para identificar a origem dos dados
4. Concatenação de todas as observações em uma única estrutura tabular
5. Adição de cabeçalhos descritivos para cada feature

6. Divisão em conjuntos de treino e validação
7. Exportação para formato CSV com codificação UTF-8

5.3 Descrição das Features

O dataset contém 22 features numéricas cuidadosamente selecionadas que descrevem de forma abrangente o estado de um veículo (ego-vehicle) e sua vizinhança imediata no ambiente de tráfego simulado. Estas features capturam informações espaciais (posições e distâncias relativas), cinemáticas (velocidades absolutas e relativas), e contextuais (presença de veículos vizinhos, faixa de rodagem) que são essenciais para caracterizar o comportamento do veículo e permitir classificação precisa. As features foram organizadas em categorias lógicas baseadas em sua natureza e na região espacial que descrevem em relação ao ego-vehicle, facilitando a interpretação e análise posterior:

Feature	Descrição	Tipo
Estado do Ego-Vehicle		
ego_speed	Velocidade do veículo principal	float32
ego_lane	Faixa atual do veículo (0, 1, 2)	float32
ego_desired_speed	Velocidade desejada pelo veículo	float32
Veículo à Frente		
front_exists	Indica se existe veículo à frente (0/1)	float32
front_distance	Distância para o veículo à frente	float32
front_speed	Velocidade do veículo à frente	float32
Veículo à Esquerda (Frente)		
left_front_exists	Indica se existe veículo na frente-esquerda	float32
left_front_distance	Distância para o veículo frente-esquerda	float32
left_front_speed	Velocidade do veículo frente-esquerda	float32
Veículo à Esquerda (Trás)		
left_rear_exists	Indica se existe veículo atrás-esquerda	float32
left_rear_distance	Distância para o veículo atrás-esquerda	float32
left_rear_speed	Velocidade do veículo atrás-esquerda	float32
Veículo à Direita (Frente)		
right_front_exists	Indica se existe veículo na frente-direita	float32
right_front_distance	Distância para o veículo frente-direita	float32
right_front_speed	Velocidade do veículo frente-direita	float32
Veículo à Direita (Trás)		
right_rear_exists	Indica se existe veículo atrás-direita	float32
right_rear_distance	Distância para o veículo atrás-direita	float32
right_rear_speed	Velocidade do veículo atrás-direita	float32
Features Derivadas		
speed_diff_front	Diferença de velocidade com veículo à frente	float32
speed_diff_left_front	Diferença de velocidade com veículo frente-esquerda	float32
speed_diff_right_front	Diferença de velocidade com veículo frente-direita	float32
speed_diff_left_rear	Diferença de velocidade com veículo atrás-esquerda	float32
Metadados		
vehicle_id	Identificador único do veículo	int
label	Classe alvo (0, 1 ou 2)	int

Tabela 3: Features do dataset de comportamento de veículos

As features capturam informações espaciais (distâncias), cinemáticas (velocidades) e relacionais (diferenças de velocidade) que são essenciais para a classificação do comportamento veicular.

5.4 Pré-processamento

O pré-processamento dos dados foi implementado de forma modular e reutilizável na classe `DataProcessor` do módulo `common/data_processing.py`, seguindo um pipeline estruturado de transformações que prepara os dados brutos para o treinamento fede-

rado. Este pipeline inclui etapas de carregamento, limpeza, normalização, detecção de metadados e particionamento, todas executadas de forma automática e configurável. A implementação segue boas práticas de machine learning, como separação clara entre conjuntos de treino e validação, normalização consistente aplicada antes do particionamento, e preservação de informações necessárias para o contexto federado (como identificadores de veículos).

5.4.1 Carregamento e Leitura dos Dados

Os arquivos CSV são carregados utilizando `pandas.read_csv()`, que realiza:

- Leitura do arquivo CSV com cabeçalho
- Parsing automático dos tipos de dados (float32 para features, int para labels)
- Carregamento em memória como DataFrame do pandas

5.4.2 Tratamento de Metadados

O dataset contém 2 colunas especiais que não são features preditivas, mas sim metadados essenciais para o Aprendizado Federado, e portanto recebem tratamento diferenciado durante o pré-processamento:

- **vehicle_id**: Extraída e armazenada separadamente para particionamento por veículo no Aprendizado Federado, removida da matriz de features antes do treinamento
- **label**: Coluna alvo com as classes (0, 1, 2), separada das features e convertida para tipo `int`

Implementação (`common/data_processing.py`, método `_load_csv`):

```
if "vehicle_id" in df.columns:
    vehicle_ids = df["vehicle_id"].values
    cols_to_drop.append("vehicle_id")

if "label" in df.columns:
    y = df["label"].values.astype(int)
    cols_to_drop.append("label")

X = df.drop(columns=cols_to_drop).values.astype(np.float32)
```

5.4.3 Normalização e Padronização

Todos os dados numéricos foram normalizados utilizando `StandardScaler` do scikit-learn, um transformador que aplica padronização Z-score (também conhecida como standardization), convertendo cada feature para ter média zero e variância unitária. Esta normalização é crucial para modelos baseados em árvore quando as features possuem escalas muito diferentes, e garante que nenhuma feature domine a métrica de divisão apenas por ter valores absolutos maiores. A transformação Z-score é definida matematicamente como:

$$z = \frac{x - \mu}{\sigma} \tag{37}$$

onde μ é a média e σ é o desvio padrão das features.

O procedimento de normalização seguiu rigorosamente 3 boas práticas fundamentais de machine learning para evitar data leakage e garantir avaliação justa do modelo:

1. **Fit no treino**: O scaler foi ajustado (`fit`) apenas nos dados de treino
2. **Transform em ambos**: A transformação foi aplicada tanto no treino quanto na validação
3. **Antes do particionamento**: A normalização ocorreu antes da distribuição dos dados entre clientes

Implementação no código (`common/data_processing.py`, método `load_and_prepare_data`):

```
# Inicializar scaler
self.scaler = StandardScaler()

# Fit apenas nos dados de treino
self.scaler.fit(X_train_all)

# Transform em treino e validação
X_train_all = self.scaler.transform(X_train_all)
X_test_all = self.scaler.transform(X_test_all)
```

5.4.4 Detecção de Número de Classes

O número de classes no problema de classificação é detectado automaticamente através da análise dos valores únicos presentes na coluna `label` do conjunto de treinamento, utilizando a função `np.unique()` do NumPy que retorna um array com todos os valores distintos encontrados:

```
num_classes = len(np.unique(y_train_all))
```

Esta detecção automática permite que os algoritmos (XGBoost, LightGBM, CatBoost) configurem corretamente:

- Função objetivo (binária vs. multi-classe)
- Número de classes para softmax
- Métricas de avaliação apropriadas

Para o dataset de veículos, o número detectado é 3 classes, configurando automaticamente os modelos para classificação multi-classe com função objetivo `multi:softprob` (XGBoost) ou equivalentes.

5.5 Particionamento dos Dados para Federated Learning

5.5.1 Particionamento por Veículo

A estratégia principal de particionamento utilizada neste trabalho é o **particionamento por veículo** (vehicle-based partitioning), onde todos os dados coletados de um mesmo veículo durante a simulação são alocados exclusivamente a um único cliente, mantendo a integridade e localidade dos dados por dispositivo. Esta abordagem é significativamente mais realista e apropriada para cenários de Aprendizado Federado veicular do que particionamentos aleatórios (IID), pois melhor representa a natureza distribuída e descentralizada dos dados em aplicações reais, onde cada veículo (cliente) naturalmente coleta e mantém suas próprias observações localmente. As principais vantagens desta estratégia incluem:

- Simula dados reais onde cada veículo (cliente) coleta suas próprias observações
- Cria naturalmente partições non-IID, já que diferentes veículos podem ter comportamentos distintos
- Preserva a correlação temporal entre observações do mesmo veículo

Algoritmo de particionamento (5 etapas):

1. Identificar todos os veículos únicos no dataset (coluna `vehicle_id`)
2. Embaralhar aleatoriamente os IDs dos veículos (seed=42)
3. Dividir os veículos em N grupos de tamanho aproximadamente igual

4. Atribuir cada grupo de veículos a um cliente
5. Cada cliente recebe todas as amostras dos veículos do seu grupo

Implementação (`common/data_processing.py`):

```
unique_vehicles = np.unique(vehicle_ids_train)
shuffled_vehicles = np.random.permutation(unique_vehicles)
vehicles_per_client = num_vehicles // self.num_clients

for client_id in range(self.num_clients):
    client_vehicle_ids = shuffled_vehicles[start_idx:end_idx]
    mask = np.isin(vehicle_ids_train, client_vehicle_ids)
    client_X = X_train_all[mask]
    client_y = y_train_all[mask]
```

Características da distribuição:

- **Total de dados**: 115.511 amostras distribuídas entre clientes
- **Distribuição de amostras**: Variável por cliente (depende do número de observações por veículo)
- **Distribuição de classes**: Naturalmente non-IID, refletindo comportamentos específicos de cada veículo
- **Validação**: Conjunto centralizado de 28.878 amostras utilizado pelo servidor

5.5.2 Particionamento IID (Fallback)

Como estratégia alternativa de fallback, o sistema também implementa particionamento IID (Independent and Identically Distributed) tradicional em 3 etapas sequenciais, utilizado automaticamente quando a coluna `vehicle_id` não está disponível no dataset ou quando deseja-se avaliar o desempenho em cenários ideais com distribuição uniforme de dados:

1. Embaralhar todas as amostras aleatoriamente
2. Dividir o dataset em N partes iguais
3. Atribuir cada parte a um cliente

Esta estratégia garante distribuição uniforme de classes entre clientes, mas não é utilizada nos experimentos principais deste trabalho, pois o dataset possui informação de `vehicle_id`.

6 CONFIGURAÇÃO EXPERIMENTAL

6.1 Ambiente Computacional

Componente	**Especificação**
Processador	[Modelo e frequência]
Memória RAM	[Quantidade] GB
GPU	[Modelo] ou N/A
Sistema Operacional	[Nome e versão]
Python	[Versão]

Tabela 4: Especificações do ambiente computacional

6.2 Hiperparâmetros dos Modelos

6.3 Configuração do Aprendizado Federado

6.4 Experimentos Realizados

6.5 Métricas de Avaliação

7 RESULTADOS

7.1 Resultados em Cenário IID

7.1.1 Comparação de Modelos

7.1.2 Comparação de Estratégias

7.2 Resultados em Cenário non-IID

7.2.1 Comparação de Modelos

7.2.2 Impacto da Heterogeneidade

7.3 Análise de Convergência

7.4 Impacto da Integração SDN

7.5 Análise Estatística

7.5.1 Teste de Friedman

7.5.2 Teste Post-hoc de Nemenyi

8 DISCUSSÃO

8.1 Interpretação dos Resultados

8.2 Comparação com Estado da Arte

8.3 Limitações do Trabalho

8.4 Implicações Práticas

9 CONCLUSÕES E TRABALHOS FUTUROS

9.1 Conclusões

9.2 Contribuições

9.3 Trabalhos Futuros

REFERÊNCIAS

[1] AUTOR, A.B.; COAUTOR, C.D. Título do artigo. **Nome da Revista**, v.10, n.2, p.45-62, 2020.

[2] AUTOR, Nome. **Título do Livro**. Editora, Cidade, Ano.

[3] AUTOR, A.; COAUTOR, B. Título do artigo. **Nome da Conferência**, p.100-110, 2021.