UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

IONÉSIO LIMA DA COSTA JÚNIOR

PyGrid:

UMA PLATAFORMA DESCENTRALIZADA PARA COMPUTAÇÃO CONFIDENCIAL E APRENDIZAGEM DE MÁQUINA FEDERADA

CAMPINA GRANDE - PB

2021

IONÉSIO LIMA DA COSTA JÚNIOR

PyGrid:

UMA PLATAFORMA DESCENTRALIZADA PARA COMPUTAÇÃO CONFIDENCIAL E APRENDIZAGEM DE MÁQUINA FEDERADA

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Professor Dr. Leandro Balby Marinho.

CAMPINA GRANDE - PB

2021

C837t Costa Junior, Ionésio Lima da.
PyGrid: uma plataforma descentralizada para computação confidencial e aprendizagem de máquina federada. / Ionésio Lima da Costa Junior. – 2021.

10 f.

Orientador: Prof. Dr. Leonardo Balby Marinho.
Trabalho de Conclusão de Curso - Artigo (Curso de Bacharelado em Ciência da Computação) - Universidade Federal de Campina Grande; Centro de Engenharia Elétrica e Informática.

1. Computação confidencial. 2. Aprendizagem de máquina federada. 3. Privacidade de dados. 4. Desenvolvimento de plataforma open-source. 5. Dados – segurança e privacidade. 6. PySyft. 7. Treinamento federado. I. Marinho, Leonardo Balby. II. Título.

CDU:004.415.2(045)

Elaboração da Ficha Catalográfica:

Johnny Rodrigues Barbosa
Bibliotecário-Documentalista
CRB-15/626

IONÉSIO LIMA DA COSTA JÚNIOR

PyGrid:

UMA PLATAFORMA DESCENTRALIZADA PARA COMPUTAÇÃO CONFIDENCIAL E APRENDIZAGEM DE MÁQUINA FEDERADA

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Professor Dr. Leandro Balby Marinho
Orientador – UASC/CEEI/UFCG

Professor Dr. Rohit Gheyi
Examinador – UASC/CEEI/UFCG

Professor Dr. Tiago Lima Massoni
Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 25 de maio de 2021.

CAMPINA GRANDE - PB

RESUMO

A preocupação com segurança e privacidade de dados tornou-se tema recorrente em diferentes esferas sociais e tecnológicas, sendo amplamente debatida por companhias e órgãos governamentais na busca por alternativas que assegurem tais direitos. Diante disso, este trabalho tem por objetivo, contribuir na construção de soluções aplicadas à preservação e privacidade de dados, auxiliando no desenvolvimento de uma plataforma "open-source" para computação confidencial e aprendizagem de máquina federada.

PyGrid: Uma plataforma descentralizada para computação confidencial e aprendizagem de máquina federada

Ionésio Lima da Costa Junior
ionesio.junior@ccc.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil

Leandro Balby Marinho
lbmarinho@dsc.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil

RESUMO

A preocupação com segurança e privacidade de dados tornou-se tema recorrente em diferentes esferas sociais e tecnológicas, sendo amplamente debatida por companhias e órgãos governamentais na busca por alternativas que assegurem tais direitos. Diante disso, este trabalho tem por objetivo, contribuir na construção de soluções aplicadas à preservação e privacidade de dados, auxiliando no desenvolvimento de uma plataforma *"open-source"* para computação confidencial e aprendizagem de máquina federada.

ACM Reference Format:
Ionésio Lima da Costa Junior and Leandro Balby Marinho. 2021. PyGrid: Uma plataforma descentralizada para computação confidencial e aprendizagem de máquina federada. In *Proceedings of (TCC).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUÇÃO

A preocupação com segurança e privacidade de dados tornou-se tema recorrente em diferentes esferas sociais e tecnológicas, sendo amplamente debatido por companhias e órgãos governamentais na busca por alternativas que garantam tais direitos.

Contudo, embora soluções venham sendo discutidas e implementadas pela comunidade científica, a adoção de tais técnicas trazem consigo limitações, impactando diretamente os modelos de negócio vigentes. Como exemplo, temos o uso de criptografia assimétrica, que assegura a privacidade e segurança dos dados, mas inviabiliza o uso de operações lógicas e aritméticas em informações criptografadas.

Outro problema resultante destas limitações é observado no âmbito científico. Ao longo da última década, observa-se grande avanço tecnológico em decorrência do uso de modelos de inteligência artificial e *"big data"*. Contudo, existem áreas que ainda não desfrutaram de tal progresso devido à natureza de seus dados. A área médica, por exemplo, possui inúmeros casos de uso onde modelos preditivos poderiam auxiliar no prognóstico de enfermidades. Doenças poderiam ser diagnosticadas com anos de antecedência através modelos preditivos, possibilitando tratamentos preventivos. Todavia, a baixa disponibilidade de dados médicos decorrente de políticas de segurança tem dificultado avanço de técnicas de inteligência artificial na área da saúde.

Diante disso, este trabalho visou discorrer e propor solucões a problemas de privacidade e proteção de dados no contexto de aprendizagem de máquina. Buscou-se contribuir com a criação de uma plataforma de computação confidencial e gerenciamento de dados, de forma a habilitar o uso de informações sensíveis, respeitando conceitos de privacidade e segurança.
[1]

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Aprendizagem de máquina federada

A abordagem federada tem como objetivo treinar modelos de inteligência artificial de forma descentralizada, permitindo que dados sensíveis permaneçam em suas infraestruturas de origem, inacessíveis durante todo o processo. Dessa forma, as informações sensíveis utilizadas durante o treinamento jamais transitarão pela rede.

Distinguindo-se da arquitetura centralizada, que possui execução linear e bem definida, o treinamento federado é realizado de forma iterativa, sendo necessária comunicação constante entre os componentes envolvidos.

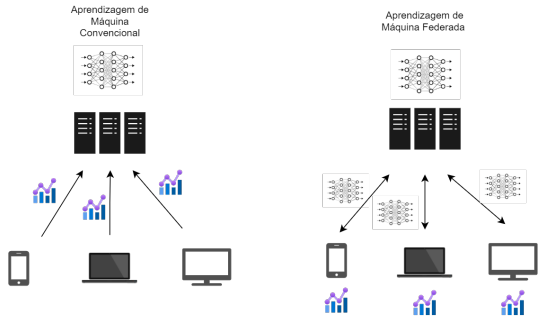

Figura 1: Esquema apresentando as diferenças de arquitetura entre treinamentos utilizando a abordagem tradicional (a esquerda) e federada (a direita).

Tal processo se dá em duas etapas: treinamento local e agregação. O treinamento local é realizado no dispositivo detentor dos

TCC, Maio 2021, Campina Grande, Paraíba, Brasil

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] "Os autores retêm os direitos, ao abrigo de uma licença Creative Commons Atribuição CC BY, sobre todo o conteúdo deste artigo (incluindo todos os elementos que possam conter, tais como figuras, desenhos, tabelas), bem como sobre todos os materiais produzidos pelos autores que estejam relacionados ao trabalho relatado e que estejam referenciados no artigo (tais como códigos fonte e bases de dados). Essa licença permite que outros distribuam, adaptem e evoluam seu trabalho, mesmo comercialmente, desde que os autores sejam creditados pela criação original."

dados, sendo gerenciado por uma entidade remota, responsável por coordenar a execução da rotina. A agregação por sua vez, consiste em diferentes técnicas que buscam a construção de um modelo preditivo global, a partir de modelos locais resultantes da primeira etapa.

2.2 PySyft

PySyft é uma biblioteca desenvolvida com o objetivo de estender técnicas de segurança e privacidade utilizadas no contexto de aprendizagem de máquina. A dependência *PySyft* tem como principal funcionalidade o suporte à aprendizagem de máquina federada. Para tanto, a biblioteca faz uso de árvores de abstração sintática como sua principal estrutura de dado, permitindo um alto nível de transparência e abstração ao executar operações remotas.

Nos trechos de código abaixo, é possível observar as diferenças de implementação entre algoritmos de aprendizagem de máquina utilizando solução nativa e solução acoplada ao *PySyft*.

```
def train():
    # Training Logic
    opt = optim.SGD(params=model.parameters(),lr=0.1)
    for iter in range(20):
        #1) erase previous gradients (if they exist)
        opt.zero_grad()

        #2) make a prediction
        pred = model(data)

        #3) calculate how much we missed
        loss = ((pred - target)**2).sum()

        4) figure out which weights caused us to miss
        loss.backward()

        #5) change those weights
        opt.step()

        #6) print our progress
        print(loss.data)
```

Quadro 1: Treinamento convencional (PyTorch nativo)

Durante a abordagem convencional, a execução de inferências, cálculos de erro e reajustes de pesos violam a privacidade dos dados utilizados durante o processo. Todo o processo ocorre de forma centralizada, nao garantindo a privacidade do detentor do dado.

Considerando cenário federado, o modelo é enviado à máquina que armazena dados privados, computando-se a inferência, erro e reajuste de pesos remotamente. Somente após isso, o modelo local é recuperado do dispositivo e agregado a um modelo global. Aqui, os dados permanecem armazenados no dispositivo detentor da informação, não tendo sua privacidade violada em nenhum momento.

```
def train():
    # Training Logic
    for iter in range(10):
        # NEW) iterate through each worker's dataset
        for data,target in datasets:
            # NEW) send model to correct worker
            model.send(data.location)
            # Call the optimizer for the worker
            # using get_optim
            opt = optims.get_optim(data.location.id)
            # 1) erase previous gradients (if they exist)
            opt.zero_grad()
            # 2) make a prediction
            pred = model(data)
            # 3) calculate how much we missed
            loss = ((pred - target)**2).sum()
            # 4) figure out which weights caused us to miss
            loss.backward()
            # 5) change those weights
            opt.step()
            # NEW) get model (with gradients)
            model.get()
            # 6) print our progress
            print(loss.get().data)
```

Quadro 2: Treinamento Federado (PyTorch + PySyft)

3 ARQUITETURA E PROJETO DA SOLUÇÃO

3.1 Visão Geral

PyGrid é uma plataforma desenvolvida com objetivo de solucionar problemas relacionados a segurança e privacidade no âmbito de ciência de dados preditiva. A solução busca habilitar o uso de informações sensíveis de forma segura e privada, viabilizando casos de uso antes impactados por limitações decorrentes de políticas de segurança convencionais. Para tal, a plataforma foi desenvolvida reunindo diversas técnicas de criptografia e anonimização, fornecendo-as como parte de seu serviço. Entre as ferramentas disponíveis, tem-se o suporte à aprendizagem de máquina federada, computação multipartidária segura, criptografia homomórfica e privacidade diferencial.

A disposição topológica da plataforma foi planejada de forma descentralizada e heterogênea, sendo composta por três aplicações: *Network, Domain e Worker*. Suas responsabilidades podem ser segmentadas a partir do seguinte esquema:

- PyGrid Network
  - Responsável por gerenciar e rotear aplicações Domain.
  - Propagar buscas por dados armazenados nos dispositivos.
- PyGrid Domain
  - Gerenciar e monitorar acesso aos dados.
  - Armazenar dados e modelos.
  - Provisionar ambientes customizados em nuvem.
- PyGrid Worker
  - Realizar computações.

Tal divisão permite uma melhor distribuição de recursos, delimitando responsabilidades entre os componentes da rede e possibilitando a elasticidade do sistema de acordo com a necessidade.

3.2 Decisões de Projeto:

3.2.1 ***Arquitetura do Sistema:*** A fim de mitigar conflitos entre aplicações *PyGrid* e bibliotecas desenvolvidas pela organização, a arquitetura adotada pela plataforma foi influenciada diretamente por padrões de projeto utilizados no desenvolvimento da biblioteca *PySyft*. Desse modo, a adição ou modificação de funcionalidades tem menor impacto na lógica inerente as aplicações, resultando em pouca ou nenhuma alteração no código fonte.

3.2.2 ***Sobrescrita do módulo de armazenamento:*** O *PySyft* utiliza-se da interface *"ObjectStore"* para armazenar e gerenciar dados. Dada a finalidade da biblioteca, a abordagem convencional utiliza memória primária como recurso de armazenamento. Para adicionar persistência à plataforma, optou-se por estender a interface da estrutura *"ObjectStore"*, preservando sua assinatura e modificando seu comportamento.

```
class MemoryStore(ObjectStore):
    # [...]

    def __getitem__(self, key: UID) -> StorableObject:
        try:
            return self._objects[key]
        except Exception as e:
            critical(f"{
            type(self)
            } __getitem__ error {key} {e}")
            traceback_and_raise(e)

    # [...]

class DiskObjectStore(ObjectStore):
    # [...]
    def __getitem__(self, key: UID) -> StorableObject:
        bin_obj = self.db.session.query(
        BinObject
        ).filter_by(
        id=str(key.value)
        ).first()
        obj_metadata = (
            self.db.session.query(
            ObjectMetadata).filter_by(
            obj=str(key.value)).first()
        )
        if not bin_obj or not obj_metadata:
            raise Exception("Object not found!")

        read_permissions = {
            VerifyKey(
            key.encode("utf-8"),
            encoder=HexEncoder): value
            for key, value in
            obj_metadata.read_permissions.items()
        }

        obj = StorableObject(
            id=UID.from_string(bin_obj.id),
            data=bin_obj.object,
            description=obj_metadata.description,
            tags=obj_metadata.tags,
            read_permissions=read_permissions,
            search_permissions=syft.lib.python.Dict(
            {VERIFYALL: None}),
        )
        return obj
    # [...]
```

Quadro 3: Trecho de código retirado da classe DiskObjectStore

3.2.3 ***Extensão de Serviços PySyft:*** As funcionalidades da biblioteca são desenvolvidas fazendo uso de uma estrutura baseada em serviços. Estes por sua vez, são executados através de um mapeamento de mensagens. Desse modo, a adição de novas funcionalidades ocorre através da definição de um novo serviço. Uma abordagem semelhante foi utilizada durante o desenvolvimento das aplicações da plataforma, tornando-a apta a suportar novas funcionalidades desenvolvidas pela dependência e adicionar suas próprias funcionalidades à lista de serviços suportados. A preservação da arquitetura proposta pela biblioteca, possibilitou a adição de novas funcionalidades através da modularização da lógica de serviços e mensagens, permitindo a coexistência de serviços *PySyft* e *PyGrid* em uma mesma estrutura de dados.

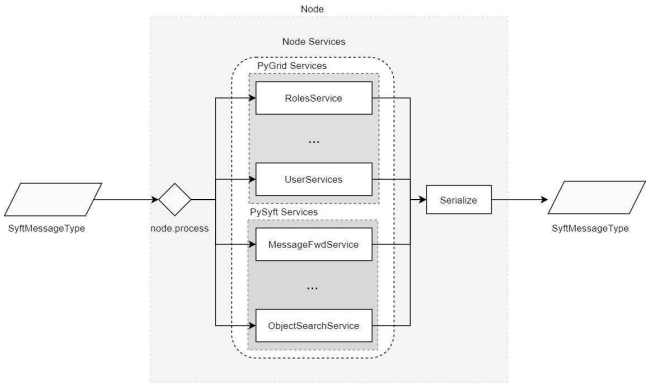

Figura 2: Representação do mapeamento e execução de serviços em uma aplicação Domain.

3.3 Tecnologias Utilizadas

O desenvolvimento da plataforma envolveu diversas tecnologias, sendo válido mencionar:

- **Python** - Linguagem de programação
- **Flask** - Framework utilizado para criar aplicações com interface de rede.
- **SQLAlchemy** - Biblioteca utilizada na comunicação com banco de dados.
- **Postgres** - Banco de dados.
- **Flask-Sockets** - Uso de protocolo websocket em aplicações Flask.
- **PyTorch** - Biblioteca para aprendizagem de máquina.
- **PyDP** - Biblioteca para uso de privacidade diferencial
- **TenSEAL** - Biblioteca para uso de criptografia Homomórfica.
- **SyMPC** - Biblioteca para uso de criptografia multi-partidária segura.
- **PySyft** - Biblioteca utilizada para gerenciamento e execução de computações remotas.

3.4 Contribuições

Durante a realização deste trabalho, inúmeras contribuições à organização *OpenMined* foram produzidas, destacando-se as seguintes funcionalidades:

- Idealização e criação de uma estrutura de dados para representar tensores privados.
- Adição de criptografia e autenticação na comunicação entre aplicações utilizando protocolo TLS.
- Aplicação do protocolo WebRTC para tradução de endereços de rede, viabilizando o envio e recebimento de dados através de ferramentas como *"Jupyter Notebook"* e *"Google Colab""*.
- Idealização e construção da plataforma PyGrid.

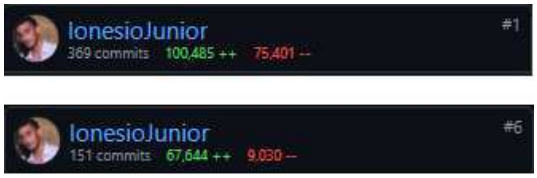

Figura 3: Estatísticas de contribuição nos repositórios PyGrid e PySyft respectivamente.

4 PROVA DE CONCEITO

Como prova de conceito deste trabalho, temos a execução de uma rotina de treinamento federado centrado em modelos. Esta técnica parte da premissa de que o treinamento será coordenado por uma aplicação responsável por sincronizar os dispositivos de rede.

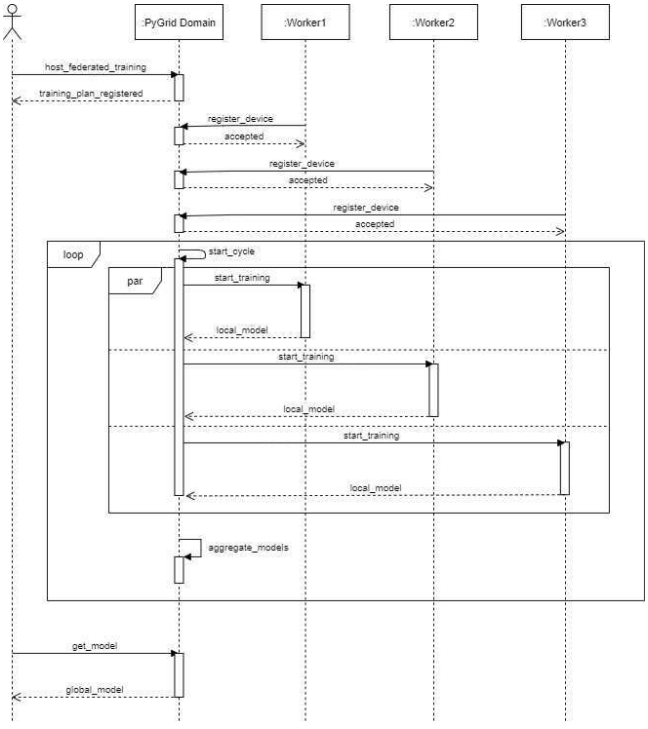

Figura 4: Diagrama de sequência simplificado representando o fluxo de treinamento federado na abordagem centrada em modelos.

Considerando a arquitetura adotada pela plataforma PyGrid, a aplicação *Domain* assume a responsabilidade de coordenador do treinamento, gerenciando e sincronizando os dispositivos (aplicações *Worker*) envolvidos no processo. Ao final dos ciclos, modelos locais são retornados a aplicação *Domain* para a construção de um modelo global.

4.1 Cientista de dados

Analisando o caso de uso sob a ótica de um indivíduo ou entidade interessado em treinar modelos preditivos, pode-se dividir a execução do treinamento a partir das seguintes etapas:

4.1.1 ***Declaração do modelo e rotina de treinamento****:* Primeiramente, define-se a arquitetura do modelo a ser treinado. Em seguida, é definido o algoritmo de treinamento através da construção de um plano, estrutura de dado que permite o registro e serialização de instruções de máquina de forma segura. Desse modo, instruções definidas nesta estrutura podem ser serializadas, enviadas e manipuladas remotamente.

```
# 1 - Definir arquitetura do modelo
local_model = MLP(th)

# 2 - Algoritmo de treinamento a ser
# executado pelos dispositivos.
@make_plan
def training_plan(
    xs=th.randn(bs, 28 * 28),
    ys=th.nn.functional.one_hot(
        th.randint(
            0,
            classes_num,
            [bs]), classes_num),
    batch_size=th.tensor([bs]),
    lr=th.tensor([0.1]),
    params=model_params_zeros,
):
    model = local_model.send(
        ROOT_CLIENT,
        send_parameters=False)

    # Substituição de parâmetro locais
    # por suas versões remotas.
    set_remote_model_params(model.modules, params)

    # forward
    logits = model(xs)

    # loss
    loss,loss_grad = model.softmax_cross_entropy_with_logits(
        logits, ys, batch_size
    )
    # backward
    grads = model.backward(xs, loss_grad)

    # SGD step
    updated_params = tuple(
        param - lr * grad
        for param, grad in zip(
        model.parameters(),
        grads)
    )
    # accuracy
    acc = model.accuracy(logits, ys, batch_size)
    return (loss, acc, *updated_params)
```

Quadro 4: Declaração do modelo a ser treinado e algoritmo de treino a ser executado remotamente

Os parâmetros do modelo de interesse são substituídos por estruturas remotas e análogas, garantindo que toda e qualquer operação feita sob o modelo durante a execução do treinamento, seja computada remotamente.

4.1.2 ***Declaração da rotina de agregação****:* O próximo passo é definir o algoritmo de agregação utilizado para gerar a versão global do modelo treinado. As técnicas utilizadas para agregação de modelos variam de acordo com o contexto do problema ao qual o modelo se propõe a resolver. Por razões didáticas, será utilizada a técnica de agregação baseada no cálculo da média aritmética dos pesos pertencentes aos modelos locais.

```
@make_plan
def avg_plan(
    avg=List(local_model.parameters()),
    item=List(local_model.parameters()), num=Int(0)
):
    new_avg = []
    for i, param in enumerate(avg):
        new_avg.append(
        (avg[i] * num + item[i]) / (num + 1)
        )
    return new_avg
```

Quadro 5: Declaração da rotina de agregação a ser utilizada.

4.1.3 ***Configuração de parâmetro gerais****:* Também se faz necessária a configuração de parâmetros gerais. Tendo finalidades diversas, estes atributos são anexados ao algoritmo de treinamento a ser registrado. No trecho de código abaixo, temos as configurações gerais de uma rotina, bem como suas respectivas definições.

```
name = "mnist"

version = "1.0"

# Configurações gerais de dispositivos (workers)
client_config = {
    # Nome da rotina de treino
    "name": name,
    # Versão do modelo
    "version": version,
    # Tamanho do batch
    "batch_size": 64,
    # Taxa de aprendizagem
    "lr": 0.01,
    # Limite de atualizações de modelo por worker
    "max_updates": 100,
}

# Configurações gerais de servidor (domain)
server_config = {
    # Número total de ciclos
    "num_cycles": 30,
    # Limiar de tempo de resposta para cada ciclo.
    "cycle_length": 60*60*24,
    # Ciclos executados por atualização do modelo global.
    "max_diffs": 1,
    # Limiar de aceitação (taxa de upload)
    "minimum_upload_speed": 0,
    # Limiar de aceitação (taxa de download)
    "minimum_download_speed": 0,
    # Execução assíncrona / iterativa
    "iterative_plan": True,
}
```

Quadro 6: Configurações gerais de dispositivos e servidor agregador

4.1.4 ***Registro de rotina de treinamento****:* Por fim, como última instrução a ser executada sob a ótica de entidades interessadas na produção de modelos preditivos, tem-se o registro da rotina de treinamento na plataforma. Uma vez registrado, a aplicação começará a processar requisições de participação por parte dispositivos, adicionando-os aos ciclos de treino.

```
# Conexão com a Aplicação "Domain"
grid = ModelCentricFLClient(address=grid_address, secure=False)
grid.connect()

# Registro da rotina de treinamento.
response = grid.host_federated_training(
    model=local_model,
    client_plans={
        "training_plan": training_plan,
    },
    client_protocols={},
    server_averaging_plan=avg_plan,
    client_config=client_config,
    server_config=server_config,
)
```

4.2 Detentor dos dados

Sob a perspectiva de entidades detentoras de dados sensíveis, o processo de treinamento federado segmenta-se da seguinte forma:

4.2.1 ***Declaração de eventos assíncronos****:* Dispositivos utilizados durante um treinamento federado utilizam-se de rotinas assíncronas para delegar a responsabilidade de sua execução à aplicação coordenadora (*Domain*). Dessa forma, a lógica adotada por estas aplicações precisa considerar a natureza assíncrona dos eventos.

```
def on_accepted(job: FLJob):
    print(f"Accepted into cycle {len(cycles_log) + 1}!")
    # Recupera configurações de treinamento predefinido
    cycle_params = job.client_config
    batch_size = cycle_params["batch_size"]
    max_updates = cycle_params["max_updates"]
    # Recupera algoritmo de treinamento e peso do modelo global atual.
    training_plan, model_params = job.plans["training_plan"], job.model
    losses, accuracies = [], []
    # Prepara dados locais para treinamento
    train_loader = th.utils.data.DataLoader(
        train_set, batch_size=batch_size, drop_last=True, shuffle=True
    )
    # Executa algoritmo de treinamento predefinido.
    for batch_idx, (x, y) in enumerate(train_loader):
        y = th.nn.functional.one_hot(y, 10)
        (model_params,) = training_plan(xs=x, ys=y, params=model_params)

        if batch_idx >= max_updates - 1:
            break
    # Retorna os pesos do modelo local ao final do ciclo.
    job.report(model_params)
    # Salva métricas locais
    cycles_log.append((losses, accuracies))

# Evento executado quando recusado em um ciclo de treinamento.
def on_rejected(job: FLJob, timeout):
    if timeout is None:
        print(f"FL training is done)")
    else:
        print(f"Rejected from cycle with timeout: {timeout}")
    status["ended"] = True

# Evento executado para tratamento de erros.
def on_error(job: FLJob, error: Exception):
    print(f"Error: {error}")
    status["ended"] = True
```

Quadro 7: Declaração de eventos assíncronos utilizados por um worker.

No trecho de código acima, são declarados respectivamente os eventos a serem executados em caso de aceitação, rejeição e erro de um ciclo.

4.2.2 ***Execução de treinamento federado utilizando dados locais****:* Uma vez definidos os eventos, dispositivos interessados em participar de treinamentos podem se autenticar na plataforma. Em caso de aceitação, os dispositivos receberão da plataforma todas as estruturas e configurações necessárias para a execução do processo.

```
def create_client_and_run_cycle():
    # Conexão com a aplicação Domain
    client = FLClient(
        url=gridAddress,
        auth_token=auth_token,
        secure=False
    )
    # Autenticação de dispositivo
    client.worker_id = client.grid_worker.authenticate(
        client.auth_token, model_name, model_version
    )["data"]["worker_id"]

    job = client.new_job(model_name, model_version)

    # Configura eventos assíncronos.
    job.add_listener(job.EVENT_ACCEPTED, on_accepted)
    job.add_listener(job.EVENT_REJECTED, on_rejected)
    job.add_listener(job.EVENT_ERROR, on_error)

    # Shoot!
    job.start()

# Execução do treinamento
while not status["ended"]:
    create_client_and_run_cycle()
    time.sleep(1)
```

Quadro 8: Encapsulamento e execução de treinamento federado utilizando dados locais.

5 EXPERIÊNCIA E LIÇÕES APRENDIDAS

A realização deste trabalho proporcionou uma reflexão sobre os impactos da adoção de políticas de preservação e privacidade no âmbito da ciência de dados preditiva. A partir do estudo de caso proposto, explorou-se diferentes técnicas e de arquiteturas desenvolvidas para previnir a violação de informações confidenciais. Como proposta de solução ao problema discutido, desenvolveu-se uma plataforma de código aberto focada no uso seguro e confidencial de dados sensíveis.

A plataforma vem sido discutida e utilizada pela comunidade científica, através de publicações de artigos científicos em revistas e conferências renomadas e projetos de pesquisa realizados em parceria com universidades internacionais.

Contudo, devido ao uso da linguagem Python, as aplicações possuem limitações no que tange a performance e gerenciamento de memória, não sendo capazes de executar tarefas que utilizem um grande volume de dados. Além disso, suas solução federada é restrita ao uso de modelos baseados em uma arquitetura de rede neural. Por fim, embora a solução busque garantir segurança e privacidade, as aplicações ainda são suscetíveis a ataques realizados por usuários com de privilégio na infraestrutura.

Como trabalhos futuros, busca-se a otimização de performance e gerenciamento de memória por meio da reescrita de sua lógica em linguagem *Rust* e o suporte a tecnologias de execução em ambientes confiáveis, permitindo autenticação de software e hardware e inviabilizando ataques executados por usuários com privilégio *sudo*.