PeerReviewFlower: un framework per il federated learning con peer review.

Mattia Passeri
mattia.passeri2@studio.unibo.it

Codice: https://github.com/passerim/peer-review-flower

Abstract

All'intersezione tra i sistemi distribuiti e il machine learning, si è sviluppato recentemente ed è in rapida ascesa il federated learning, un approccio all'apprendimento automatico distribuito che permette di addestrare un modello predittivo condiviso da un insieme di dispositivi diversi mantenendo i dati sui dispositivi stessi, facendo così a meno della necessità di raccogliere e archiviare i dati in maniera centralizzata, con lo scopo di garantire la privacy dei dati dei partecipanti. Oggigiorno esistono alcuni framework software, in vari stadi di sviluppo, che consentono di implementare nella pratica semplici sistemi di federated learning su dispositivi eterogenei. Questo lavoro nasce dall'idea di estendere Flower, uno dei più promettenti framework a disposizione, per supportare un nuovo meccanismo di federated learning che consente l'implementazione di algoritmi ispirati a una procedura comune in ambito accademico, detta di revisione tra pari o peer review in inglese, con l'obiettivo di ottenere modelli di previsione più performanti. La libreria sviluppata, chiamata PeerReviewFlower, è stata quindi utilizzata per implementare un nuovo algoritmo federato di tipo gradient boosted decision trees, chiamato FedLSBT, con cui si sono realizzati esperimenti per task di regressione e classificazione.

1 Introduzione

La recente diffusione capillare di dispositivi mobili connessi alla rete, dotati di sensori e utilizzati in maniera sempre maggiore dalle persone sia nella vita quotidiana sia sul lavoro, ha fatto sì che ora nell'ambito del machine learning e della data science si stiano sviluppando idee e applicazioni con l'obiettivo di sfruttare le capacità di questi dispositivi di raccogliere, memorizzare e rendere disponibili grandi quantità dati per addestrare modelli predittivi. Tipicamente però, gran parte di questi dati contengono informazioni personali relative agli individui che li generano e il loro utilizzo è soggetto a restrittivi vincoli per motivi di privacy. In questo caso, il paradigma tradizionale di gestione dei dati prevede costose e laboriose attività di centralizzazione e anonimizzazione delle informazioni, ponendo così importanti limiti alle applicazioni.

Il federated learning è un nuovo approccio all'apprendimento automatico distribuito che permette di addestrare un modello predittivo condiviso da un insieme di dispositivi distribuiti sfruttando i dati memorizzati localmente senza la necessità di raccoglierli in maniera centralizzata [9]. I dispositivi che partecipano all'apprendimento, detti *client*, sono coordinati da un aggregatore che prende il nome di *server*. Ogni client mantiene un modello predittivo locale e un insieme privato di dati che non vengono mai caricati sul server. Ad ogni iterazione del processo di addestramento, i client utilizzano i dati in loro possesso per produrre un aggiornamento dei parametri del proprio modello locale e inviano questo aggiornamento al server. Il compito del server è orchestrare l'addestramento distribuito nell'arco di un certo numero di iterazioni, selezionando di volta in volta i client che partecipano all'addestramento e aggregando gli aggiornamenti provenienti dai client selezionati, per produrre al termine di ogni iterazione un nuovo modello globale.
Questo progetto riguarda lo sviluppo di una nuova libreria software per permettere la sperimentazione di un'estensione del processo di federated learning ispirata alla procedura, comune in ambito accademico, di revisione tra pari o in inglese peer review. Questa variante di federated learning con peer review, detta anche peer review federated learning, prevede ad ogni iterazione del processo di addestramento l'aggiunta di una fase, chiamata *review* o *peer review*, di validazione degli aggiornamenti prodotti da ciascun client. Per lo sviluppo della libreria si sono considerati prima di tutto i framework di federated learning attualmente disponibili nel mondo open source, al fine di individuare uno di questi da utilizzare come base di partenza per essere esteso con l'implementazione di un processo flessibile e configurabile di peer review. Come parte integrante del progetto sono stati realizzati degli esempi di utilizzo della libreria sviluppata e si sono condotte sperimentazioni che hanno portato all'ideazione di un nuovo algoritmo di addestramento federato per modelli di tipo gradient boosted decision trees.

2 Background

In questa sezione si descrive che cos'è il federated learning, qual'è la sua origine, i suoi recenti sviluppi e i framework a disposizione per implementarlo nella pratica. Infine, si entra nel dettaglio di Flower, il framework scelto come punto di partenza per questo progetto.

2.1 Federated learning

Il federated learning è una tecnica di machine learning che prevede l'addestramento di un modello di previsione globale condiviso da molteplici client distribuiti che mantengono in locale i propri dati senza necessità di condividerli con l'esterno. In questo modo, il federated learning si contrappone a tutte le tecniche di machine learning tradizionali che richiedono la centralizzazione di tutti i dati utilizzati per l'addestramento in un'unica macchina o base dati, e si pre-

senta come la soluzione ideale per applicare, più in generale, gli strumenti della data science in tutti i domini applicativi in cui è cruciale preservare la privacy, tra cui: sanità, automotive, IOT e telecomunicazioni.
Il federated learning è stato per la prima volta impiegato su larga scala e reso popolare da Google tra il 2016 e il 2017, quando sono stati pubblicate le prime notizie di un sistema che, utilizzando questa tecnica, era in grado di predire le parole digitate dagli utenti sulla tastiera dello smartphone grazie a modelli neurali ricorrenti addestrati sui dispositivi stessi [5].

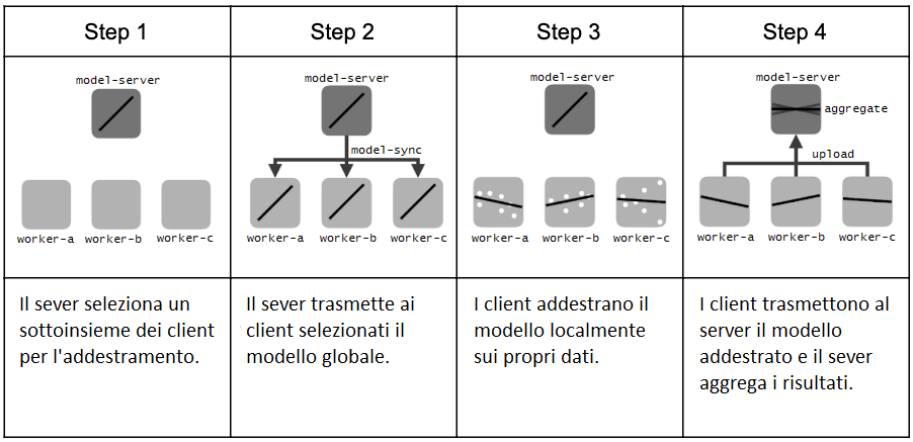

Figura 1: Schema del processo generale di federated learning.

Caratteristiche. L'idea alla base del federated learning prevede che il modello predittivo globale sia memorizzato in un server che comunica con un insieme di client distribuiti. Facendo riferimento all'immagine in Figura 1, l'addestramento procede per iterazioni in maniera sincrona, con il server che ad ogni iterazione seleziona un sottoinsieme dei client disponibili (Step 1), i quali scaricano il modello globale (Step 2), eseguono un addestramento in locale con i dati in loro possesso (Step 3) e al termine dell'addestramento inviano al server il proprio modello aggiornato a partire dal precedente modello globale (Step 4). Il server, dopo aver ricevuto il modello aggiornato di ciascuno dei client selezionati all'iterazione corrente, aggrega i modelli ricevuti per produrre un aggiornamento complessivo del modello globale. In letteratura esistono numerose varianti di questo meccanismo che prevedono accorgimenti come: aggiornamenti asincroni, tecniche di compressione dei messaggi per aumentare l'efficienza nella comunicazione, protocolli di cifratura degli aggiornamenti e privacy differenziale.
Sebbene una delle caratteristiche del federated learning sia quella di non essere vincolato all'utilizzo di un particolare algoritmo di apprendimento, l'aggregazione centralizzata degli aggiornamenti prodotti dai client fa sì che risulti naturale adottare algoritmi predittivi di tipo parametrico, mentre nel caso di algorit-

mi non parametrici l'applicazione del federated learning è ancora notevolmente limitata e oggetto di studio.
La principale differenza tra il federated learning e gli approcci di addestramento distribuito che sfruttano la parallelizzazione dei dati, tradizionalmente utilizzati per l'addestramento di modelli di apprendimento di grandi dimensioni, risiede nelle assunzioni che riguardano i dataset locali appartenenti ai client. Infatti, mentre l'addestramento distribuito con parallelizzazione dei dati richiede una fase di centralizzazione dei dati e di partizionamento del dataset in maniera indipendente e identicamente distribuita (IID), i dataset locali nel federated learning tipicamente non soddisfano questa proprietà e possono variare da client a client anche nelle dimensioni.

Formulazione del problema. Si consideri uno scenario in cui si hanno N client distribuiti e ciascun client $k = 1...N$ ha una funzione obiettivo locale L_k definita sul suo insieme di dati locale D_k. L'obiettivo dell'addestramento federato è risolvere il seguente problema di ottimizzazione di una funzione che è definita come la media pesata della funzione obiettivo su tutti i client:

$$\arg\min_{\theta} L(\theta) = \sum_{k=1}^{N} w_k L_k(D_k, \theta)$$

dove w_k è il peso associato al client k-esimo ed è generalmente proporzionale alla quantità di dati che esso possiede mentre θ rappresenta i parametri del modello predittivo globale.

Federated Averaging. FedAvg [9], Federated Averaging per esteso, è il più utilizzato algoritmo di federated learning ed è generalmente applicato e descritto utilizzando algoritmi di machine learning parametrici addestrabili mediante discesa del gradiente. Dato che in un contesto di ottimizzazione federata, i costi delle comunicazioni client-server contribuiscono in maniera preponderante sul tempo necessario all'addestramento, con l'obiettivo di aumentare l'efficienza del algoritmo federato si vuole aumentare la computazione eseguita sui client al fine di ridurre il numero di round necessari alla convergenza dell'addestramento. L'idea alla base dell'algoritmo FedAvg è quella di far si che i client eseguano tipicamente più di un'iterazione completa di discesa del gradiente, aggiustando i parametri del modello locale utilizzando i dati in proprio possesso, prima di inviare al server questi parametri per l'aggiornamento del modello globale. La logica di aggregazione lato server non fa altro che effettuare una media dei modelli ricevuti da tutti i client durante un'iterazione dell'algoritmo, eventualmente applicando pesi proporzionali al numero di campioni che ciascun client possiede.

2.2 Framework per il federated learning

Attualmente diversi framework open source supportano l'implementazione di algoritmi di federated learning, di seguito se ne introducono alcuni tra i principali e si motiva la scelta di Flower come base per questo progetto.

- **FedML** [8]: è un framework per federated learning basato su PyTorch. La sua architettura comprende un livello core che include la logica di addestramento e implementa i protocolli di comunicazione distribuita, mentre un'API di alto livello permette di implementare vari algoritmi di federated learning, lavorare con i dati e con i modelli predittivi.

- **Flower** [2]: diversamente dagli altri è agnostico rispetto al framework scelto per implementare i modelli predittivi. Flower permette il deployment sia distribuito sia in locale, sotto forma di simulazioni, di sistemi di federated learning su dispositivi eterogenei, essendo disponibili implementazioni della libreria lato client anche per piattaforme mobile (iOS, Android e IOT).

- **PySyft** [11]: si basa su PyTorch come framework per il machine learning ed è incentrato su computazione sicura distribuita e privacy differenziale, supportando un deployment sia nel concentrato sia distribuito. Tuttavia, nonostante siano forniti una serie di tutorial sul repository del progetto, manca una documentazione dettagliata e pertanto è difficile lavorarci.

- **Tensorflow Federated** [6]: sviluppato da Google come estensione di Tensorflow, la sua architettura è divisa in due livelli: un livello core che dà la possibilità di effettuare computazioni federate e un'API di più alto livello per lavorare con modelli, dataset e implementare algoritmi. La più grande limitazione di questo framework è il fatto di poter essere utilizzato solamente per simulare sistemi di federated learning in locale in quanto non supporta il deployment distribuito.

Tra queste alternative disponibili, si è scelto di adottare Flower come base per il progetto da realizzare in quanto è il framework che risponde meglio alle esigenze di essere: facilmente estendibile, applicabile sia in contesti distribuiti sia per simulazioni, disponibile su piattaforme eterogenee e agnostico rispetto alla specifica libreria impiegata per implementare algoritmi di machine learning.

2.3 Flower

Panoramica. Flower è un framework che facilita l'implementazione dei componenti fondamentali di un sistema di federated learning, è indipendente dallo specifico framework utilizzato per eseguire algoritmi di machine learning ed è principalmente sviluppato in linguaggio Python ma sono in via di sviluppo anche porting verso altri linguaggi, come Java e C++, per il deployment su piattaforme mobile. Con Flower è possibile trasportare con una certa facilità una pipeline pre-esistente di machine learning in un contesto federato. Inoltre, Flower si contraddistingue per la possibilità di implementare simulazioni di sistemi di federated learning in locale prima di passare al deployment in ambiente distribuito. Flower, infine, consente agli utilizzatori di implementare in maniera piuttosto flessibile nuovi algoritmi di federated learning.
Di seguito sono elencati gli obiettivi con cui è sviluppato Flower.

- **Supporto e sperimentazione su dispositivi eterogenei**: per dare la possibilità agli utilizzatori della libreria di eseguire Flower su dispositivi di tipologia diversa, nel repository di Flower sono presenti esempi di utilizzo su piattaforme Android, iOS e IOT. La portabilità di Flower è resa possibile dalla sua architettura essenziale e dall'impiego di gRPC come protocollo di serializzazione dei messaggi.

- **Scalabilità**: per far fronte alla caratteristica del federated learning di essere applicato a sistemi con un numero molto elevato di client, Flower dà la possibilità di eseguire sperimentazioni su una singola macchina simulando un numero elevato di client.

- **Facilità nel passaggio dalla simulazione in locale al distribuito**: al fine di comprendere l'applicabilità di nuove idee ed algoritmi, solitamente sviluppati in contesti simulati, a scenari reali in cui l'esecuzione si sposta su dispositivi distribuiti, è possibile passare senza necessità di modifiche dall'esecuzione simulata degli algoritmi al deployment distribuito.

- **Indipendenza dal framework di apprendimento**: Flower adotta come struttura dati preferenziale per la rappresentazione dei modelli predittivi gli array multidimensionali di NumPy. In questo modo Flower rimane indipendente dal tipo di framework e dal tipo di algoritmo di machine learning che si vuole utilizzare purché vi sia la possibilità di codificare i parametri come array NumPy.

Architettura. [1] Nel federated learning si possono distinguere due tipologie di computazioni: computazioni eseguite in locale sui singoli client che hanno accesso ai dati e computazioni eseguite sul server che aggregano i risultati delle computazioni dei client. L'architettura di Flower riflette questa caratteristica e spezza la logica algoritmica in due parti: una eseguita sui client e una eseguita sul server.
La logica lato server gestisce tramite una serie di chiamate da parte del server ai metodi della classe astratta `Strategy`: la selezione dei client partecipanti ad ogni round di federated learning, la configurazione sia dell'addestramento sia della validazione del modello globale e l'aggregazione dei risultati restituiti dai client. Un'implementazione della classe astratta `Strategy` definisce quindi uno specifico algoritmo di federated learning insieme alla rispettiva implementazione del client, che specifica come debba essere effettuato l'addestramento e la validazione del modello predittivo sul dataset locale. Un altro componente fondamentale lato server è la classe `ClientManager` che gestisce un insieme di oggetti `ClientProxy`, ognuno dei quali rappresenta un client connesso al server e consente al server di inviare e ricevere messaggi con il client stesso. La classe `Server` implementa il ciclo di addestramento federato e chiama le callback di una specifica implementazione della classe astratta `Strategy`, definita dall'utilizzatore della libreria, eseguendo un ciclo che in ordine:

(1) chiede alla strategia di selezionare i client che parteciperanno al prossimo round di addestramento e di associare ad ogni client un messaggio di

configurazione dell'addestramento che contiene il modello globale e un dizionario di coppie chiave-valore per specificare eventuali iperparametri dell'addestramento;

(2) invia ai client selezionati le relative configurazioni e attende i risultati;

(3) riceve dai client gli aggiornamenti prodotti come risultato dell'addestramento oppure messaggi di errore nel caso in cui falliscano alcune delle procedure di addestramento sui client;

(4) delega l'aggregazione dei risultati alla strategia;

(5) se come risultato dell'aggregazione eseguita dalla strategia si ha un nuovo insieme di parametri globali allora il server aggiorna il modello globale, altrimenti passa al round successivo;

(6) eventualmente, è possibile eseguire ad ogni iterazione un ulteriore round di validazione delle prestazioni del modello globale corrente, che prevede una fase di configurazione della validazione e una successiva fase di aggregazione dei risultati.

La logica lato client è puramente reattiva, infatti i client, che sono istanze di classi che estendono la classe astratta `Client` (o in alternativa `NumPyClient`), attendono la ricezione di messaggi da parte del server ed eseguono l'handler associato al tipo di messaggio ricevuto.
I tipi di messaggi previsti dal protocollo di Flower e i relativi handler sono:

- `GetPropertiesIns`: il messaggio contiene un dizionario chiave-valore, la sua ricezione scatena l'esecuzione della callback `get_properties` il cui risultato deve essere un messaggio di tipo `GetPropertiesRes` contenente a sua volta un dizionario chiave-valore e può essere utilizzato per inviare e ricevere informazioni tra client e server;

- `GetParametersIns`: la ricezione di un messaggio di questo tipo scatena l'esecuzione della callback `get_parameters` il cui risultato deve essere un messaggio di tipo `GetParametersRes` contenente i parametri del modello del client;

- `FitIns`: questo messaggio trasporta il modello globale inviato dal server al client e un dizionario di configurazione dell'addestramento, alla ricezione del messaggio il client esegue la callback `fit` il cui risultato deve essere un messaggio di tipo `FitRes` contenente: il modello aggiornato, il numero di campioni utilizzati per l'addestramento e un dizionario che può contenere qualsiasi altra informazione sia necessario restituire al server;

- `EvalIns`: è il messaggio con cui il server invia al client il modello globale per effettuarne la validazione sul set di test locale, il client esegue la callback `evaluate` che restituisce come risultato un messaggio di tipo `EvalRes` contenente le metriche di validazione desiderate.

Flower prevede che tutti i messaggi scambiati tra client e server siano serializzati in formato binario. I client ricevono ogni messaggio come array di byte, deserializzano il messaggio per ottenere l'istruzione inviata dal server ed eseguono l'handler associato al tipo di istruzione ricevuta. Il risultato di ogni handler dovrà essere un messaggio che rappresenta il risultato, che viene serializzato in un array di byte e inviato al server. Poiché il federated learning richiede un protocollo di comunicazione stabile ed efficiente, Flower utilizza gli stream bidirezionali in formato binario di gRPC [7]. Attraverso la sintassi di gRPC sono definiti i tipi dei messaggi del protocollo e l'implementazione del codice per la serializzazione e la deserializzazione viene generata in maniera automatica.
Flower prevede anche un meccanismo denominato Virtual Client Engine (VCE) per consentire simulazioni in locale, anche con numerosi client, in maniera efficiente. Il VCE crea un oggetto `ClientProxy` per ogni client, ma rimanda l'istanziazione vera e propria dell'oggetto client (compresi i dati e il modello locale) al momento in cui il server deve contattare il client per eseguire una computazione. Questo consente di risparmiare risorse, soprattutto per quanto riguarda la necessità di mantenere in memoria dati e modelli per ogni client. Il VCE è implementato sfruttando un framework per la computazione distribuita in Python chiamato Ray [10].

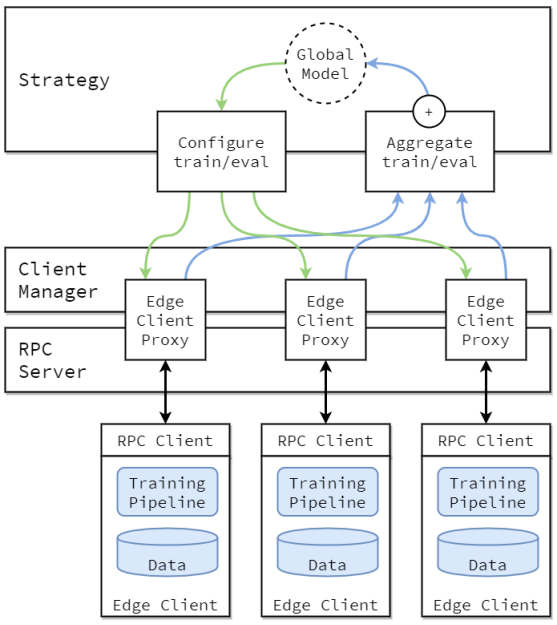

Figura 2: Architettura di Flower in un deployment distribuito.

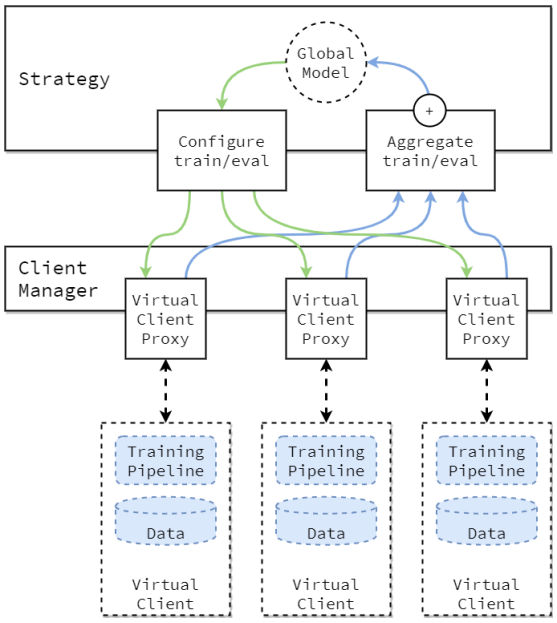

Figura 3: Architettura di Flower in un deployment simulato, da notare che la comunicazione tra client e server non avviene tramite RPC.

Utilizzo. Come anticipato, con Flower è possibile implementare sistemi di federated learning in pochi passi. In un sistema di federated learning implementato con Flower, il server interagisce con i client inviando messaggi che mettono in esecuzione i rispettivi handler previsti dalla classe astratta `Client`. Quando il server seleziona un particolare client per l'addestramento o la validazione, invia ad esso un messaggio contenente i parametri del modello globale e un dizionario chiave-valore che può contenere gli iperparametri della procedura da eseguire. Ogni client riceve quindi le istruzioni dal server ed esegue la procedura associata. Flower fornisce anche una classe astratta `NumPyClient` che facilita l'implementazione della classe astratta `Client` quando si utilizzano modelli parametrici, come nel caso delle reti neurali, che possono essere rappresentati come una lista di array NumPy.
Per realizzare un client in Flower è sufficiente implementare i seguenti metodi della classe astratta `NumPyClient`:

- `get_parameters`: restituisce i parametri del modello locale del client rappresentati come una lista di array NumPy;
- `get_properties`: restituisce un dizionario di coppie chiave-valore arbitrariamente costruito dal client, eventualmente in base alle informazioni ricevute dal server;

- `fit`: riceve dal server il modello globale rappresentato come una lista di array NumPy, imposta i parametri del modello locale, addestra il modello e restituisce i parametri aggiornati del modello sotto forma di una lista di array NumPy;
- `evaluate`: riceve dal server il modello globale, testa il modello globale sul dataset locale di validazione e restituisce i risultati ottenuti.

Di seguito si fornisce l'esempio di una semplice implementazione di un client che utilizza il framework PyTorch per addestrare un modello di rete neurale su un set di dati locale che è suddiviso in due sottoinsiemi di addestramento e di validazione.

```
class CifarClient(fl.client.NumPyClient):

    ...

    def get_parameters(self, config):
        return [v.cpu().numpy() for k, v in self.net.state_dict().items()]

    def get_properties(self, config):
        return {}

    def set_parameters(self, parameters):
        params_dict = zip(self.net.state_dict().keys(), parameters)
        state_dict = OrderedDict(
            {k: torch.tensor(v) for k, v in params_dict}
        )
        self.model.load_state_dict(state_dict, strict=True)

    def fit(self, parameters, config):
        self.set_parameters(parameters)
        train(self.net, self.trainloader, epochs=config["num_epochs"])
        return self.get_parameters(), len(self.trainloader.dataset), {}

    def evaluate(self, parameters, config):
        self.set_parameters(parameters)
        loss, _ = test(self.net, self.testloader)
        return float(loss), len(self.testloader.dataset), {}
```

Figura 4: Implementazione di un semplice client di esempio utilizzando PyTorch.

Un algoritmo di federated learning può essere implementato in Flower implementando opportunamente la classe astratta `Strategy`, che rappresenta l'astrazione di un algoritmo di apprendimento federato e prevede che siano implementati alcuni metodi che il server chiama ad ogni iterazione del ciclo di apprendimento nell'ordine mostrato dal diagramma di sequenza seguente.

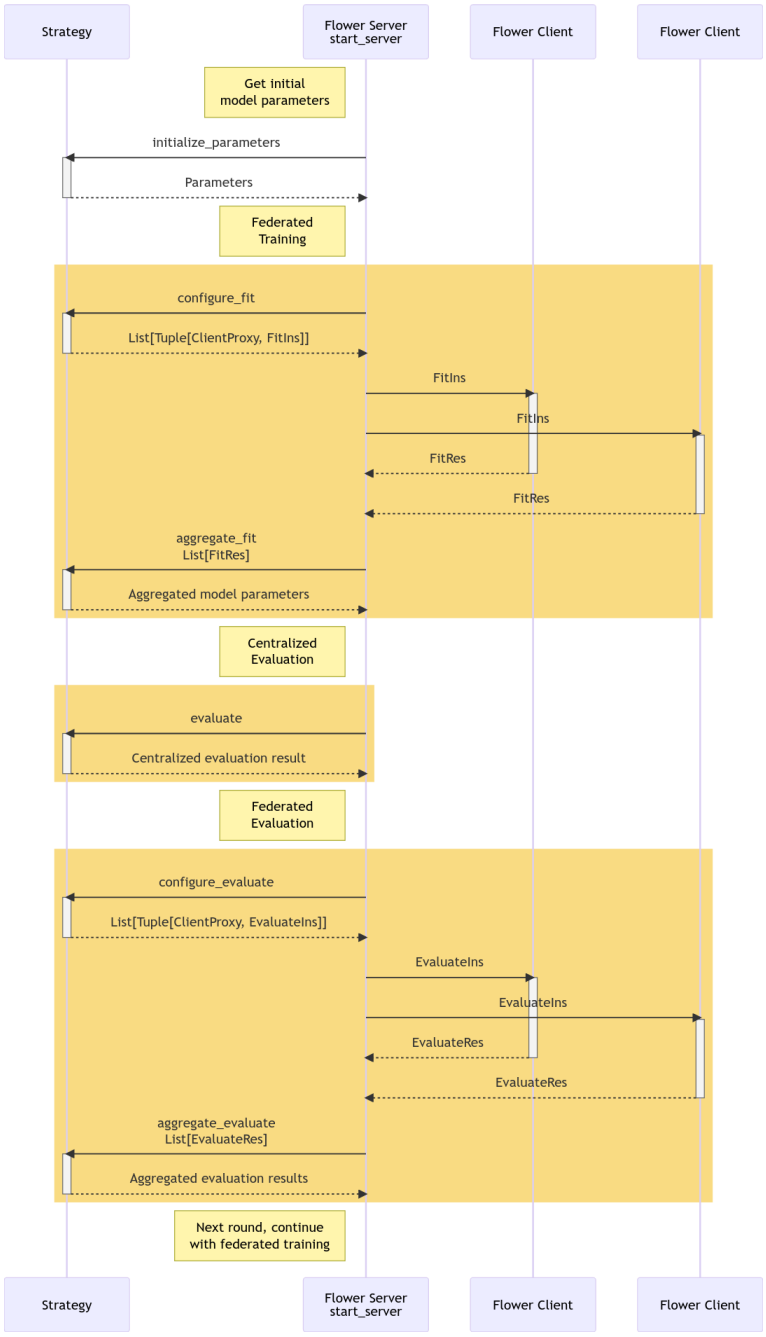

Figura 5: Diagramma di sequenza del meccanismo di federated learning implementato da Flower.

Di seguito si descrive brevemente il ruolo di ciascun metodo della classe astratta `Strategy`:

- `initialize_parameters`: è chiamato solo una volta, all'inizio dell'processo di federated learning e serve a fornire al server i parametri del modello globale iniziale in forma serializzata come oggetto di tipo `Parmeters`;
- `configure_fit`: permette di configurare il successivo round di addestramento selezionando i client e preparando le istruzioni da inviare loro;
- `aggregate_fit`: serve per aggregare i risultati restituiti dai client selezionati durante il round di addestramento corrente;
- `configure_evaluate`: permette di configurare il successivo round di validazione distribuita del modello globale;
- `aggregate_evaluate`: serve per aggregare i risultati restituiti dai client selezionati durante il round di validazione corrente;
- `evaluate`: metodo per validare le performance del modello globale in maniera centralizzata lato server.

Sul repository del progetto si fornisce l'implementazione completa, sotto forma di notebook Jupyter, di una simulazione di addestramento federato di un modello CNN sul dataset CIFAR-10, utilizzando Flower e l'algoritmo FedAvg.

3 Obiettivo

L'obiettivo di questo progetto è la realizzazione di un framework software che consenta di sperimentare un nuovo approccio al federated learning chiamato peer review federated learning.

3.1 Peer Review Federated Learning

Il peer review federated learning è un'estensione del processo di federated learning tradizionale ispirata alla procedura, comune in ambito accademico, di revisione tra pari. Questa variante di federated learning, detta anche federated learning con peer review, prevede l'aggiunta di una fase di validazione degli aggiornamenti prodotti da ciascuno dei client durante l'addestramento.
In particolare, ad ogni iterazione di addestramento federato, l'aggiornamento dei parametri del modello globale viene prodotto in due fasi che prendono il nome di fase di *training* e fase di *review*. La fase di training corrisponde a un singolo round di addestramento del federated learning tradizionale in cui viene selezionato un certo numero di client per produrre un insieme di aggiornamenti provvisori dei parametri del modello globale. La fase di review, che non è presente nel federated learning tradizionale, è una procedura, che può essere composta da molteplici round di comunicazione da server a client e da client a server, con cui l'aggiornamento provvisorio prodotto da ciascun client viene

sottoposto a uno o più cicli di validazione da parte degli altri client per valutarne la bontà prima che possa essere aggregato al modello globale.
Un round di review prevede quindi:

- la selezione da parte del server di un sottoinsieme dei client che parteciperanno al round di review;
- l'invio da parte del server ai client selezionati di uno o più aggiornamenti provvisori del modello globale, eventualmente insieme al modello globale stesso;
- la validazione da parte dei client che partecipano al round di review degli aggiornamenti ricevuti dagli altri client;
- una fase di aggregazione lato server dei risultati restituiti dai client ed eventualmente la produzione di un nuovo insieme di aggiornamenti provvisori del modello globale.

Al termine della fase di review il server aggrega gli aggiornamenti provvisori risultanti per ottenere un nuovo insieme di parametri del modello globale da sostituire ai parametri del modello corrente.
Di seguito si riporta in pseudocodice la descrizione del comportamento di un server che implementa il ciclo di apprendimento federato con peer review.

```
Input: num_rounds         # Number of FL rounds to perform
       max_review_rounds  # Max review rounds to perform each FL round

for n = 1...num_rounds:
  # training step
  select N clients
  send global model to selected clients along with training instructions
  receive model updates and aggregate them
  # review
  for m = 1...max_review_rounds:
    # review step
    select M clients
    send global model and candidate model parameters to selected clients
    receive parameters reviews and aggregate them
    if stop_review:
      break
  # candidate model parameters aggregation after multiple review steps
  aggregate candidate models parameters
```

Figura 6: Pseudocodice del comportamento di un server che implementa il ciclo di federated learning con peer review.

3.2 Caratteristiche del framework

Le caratteristiche desiderate per il framework che si vuole realizzare sono:

- **flessibilità**: cioè dare la possibilità di utilizzare il framework per implementare algoritmi diversi;

- **semplicità**: fare sì che gli utenti della libreria possano implementare nuovi algoritmi concentrandosi prevalentemente sulla logica algoritmica, ignorando aspetti relativi alla distribuzione e ai protocolli di comunicazione;
- **supporto alla simulazione**: affinché gli algoritmi implementati possano essere testati in contesti di prova per valutarne le performance;
- **supporto al deployment distribuito**: per fare sì che gli algoritmi sviluppati e testati in contesti simulati possano essere sperimentati anche in ambiente distribuito e in contesti reali.

Come punto di partenza si è individuata la libreria Flower, di cui si è discusso in precedenza, che si vuole estendere in maniera tale da fare sì che possa supportare l'implementazione di nuovi algoritmi di federated learning con peer review. Quello che si vuole realizzare è quindi una nuova libreria che estenda e faccia uso di Flower per implementare algoritmi di federated learning con peer review e ne consenta il deployment in ambiente sia simulato sia distribuito. Questo permetterà agli utenti di sperimentare una nuova variante di federated learning al fine di valutarne vantaggi e svantaggi rispetto alla forma tradizionale.

3.3 Qualità ed efficacia

Per assicurare la buona riuscita del progetto si individuano due aspetti del risultato finale da considerare: la qualità del software prodotto e l'efficacia del risultato. Dal punto di vista della qualità del software si prevede un testing esaustivo di tutti i componenti della libreria, eseguito sia in forma manuale sia automatizzato tramite continuous integration. L'efficacia del lavoro svolto dipenderà invece dalla effettiva possibilità di implementare in maniera semplice nuovi algoritmi di federated learning con peer review grazie all'utilizzo della libreria. Come fattore comune ad entrambi gli aspetti si considera l'aderenza di questa estensione di Flower alla libreria originale, con cui deve essere mantenuta la massima compatibilità possibile sia in termini di interfacce sia in termini di similarità di utilizzo.

4 Analisi dei requisiti

L'obiettivo di questo progetto richiede lo sviluppo di una libreria che consenta di implementare ed eseguire algoritmi di federated learning con peer review in ambiente sia distribuito sia concentrato sotto forma di simulazioni. Attualmente, come già discusso precedentemente, esistono già alcune soluzioni per lavorare con il federated learning e si tratta per lo più di framework in via di sviluppo che utilizzano il linguaggio Python per via della sua popolarità in ambito data science. Tra le librerie a disposizione si è individuata Flower, da utilizzare come punto di partenza per essere estesa in quanto la sua flessibilità e la sua relativa semplicità fanno sì che sia possibile estendere il comportamento di alcuni suoi componenti secondo le necessità individuate mantenendo la compatibilità

sia con gli altri componenti sia con il protocollo di comunicazione che la libreria utilizza per realizzare la comunicazione client-server. Queste caratteristiche fanno si che la scelta di appoggiarsi su Flower risulti particolarmente conveniente e preferibile all'implementazione da zero di una nuova libreria per il federated learning.
Come requisito funzionale del framework da realizzare si ha l'estensione del meccanismo di federated learning tradizionale, basato su un singolo round di comunicazione server-client-server, a una logica che preveda più round, dei quali uno è eseguito obbligatoriamente, il round di training, mentre gli altri sono un numero a piacere di round di review.
Di fondamentale importanza per lo sviluppo futuro di questo progetto sarà mantenere la massima compatibilità con Flower, sia in termini di componenti sia di interfacce. Dato che Flower è in fase di sviluppo attivo, è auspicabile che i cambiamenti che la libreria subisce nel tempo possano essere integrati con uno sforzo relativamente limitato nell'estensione che si realizzerà. Per favorire questo requisito di carattere non funzionale si vuole rimanere aderenti al design di Flower che prevede una logica predefinita nel server che richiama varie callback personalizzabili dall'utente che implementano una particolare strategia di addestramento.
Inoltre, essendo i componenti che supportano la peer review delle estensioni dei componenti di Flower, è bene realizzare una gerarchia di specializzazione tra i componenti di Flower e i componenti con peer review, con quest'ultimi che specializzano i corrispondenti componenti della libreria originale e risultano ad essi sostituibili se necessario.
Infine, si richiede che sia realizzato un esempio di utilizzo della libreria al fine di mostrarne il funzionamento nella pratica. Qual'ora la libreria dovesse risultare efficace nell'implementazione di algoritmi di federated learning con peer review, sarà parte integrante del lavoro anche la realizzazione di sperimentazioni del federated learning con peer review che sfruttino la libreria sviluppata.

5 Design

La libreria è sviluppata come un'estensione di Flower e come tale i componenti realizzati sono pensati soprattutto come specializzazioni dei componenti di Flower. In questo modo è possibile mantenere la massima compatibilità sia in termini di interfacce sia per quanto riguarda l'utilizzo dei componenti che non è necessario modificare (il concetto di interfaccia non è presente in Python ma in questo caso il termine interfaccia è utilizzato per fare riferimento alle *abstract base classes* che modellano i componenti fondamentali di Flower).
Il punto di riferimento per il design della libreria è il comportamento del server, descritto in forma di pseudocodice in Figura 6 e formalizzato dal diagramma di attività riportato in seguito, in base al quale sono sviluppate le estensioni dei componenti di Flower che modellano il server, il client e la strategia di apprendimento.
Tra le diverse alternative implementative, al fine di sfruttare la libreria di partenza e preservare la compatibilità con essa, si sceglie di basarsi il più possibile

sul protocollo e sui componenti di Flower aggiungendo le funzionalità necessarie ispirandosi al design delle classi già esistenti.

5.1 Struttura

Al fine di ottenere il meccanismo desiderato di federated learning con peer review si devono sviluppare alcuni nuovi componenti che estendano il comportamento dei componenti già presenti in Flower. In particolare, non è necessario modellare entità completamente nuove, tuttavia è necessario specializzare in maniera opportuna alcune delle entità esistenti.
Si deve quindi implementare:

- un server `PeerReviewServer` che estenda la classe `Server` di Flower e sia in grado di eseguire molteplici round di review, il cui numero possa essere stabilito anche dinamicamente;
- un client `PeerReviewClient` che alla ricezione di un apposito messaggio sia in grado di eseguire, oltre a una procedura per l'addestramento del modello predittivo sul dataset locale, anche una procedura per effettuare la review dei parametri ricevuti dal server;
- un client `PeerReviewNumPyClient` che supporti il processo di review e sia l'equivalente della classe `NumPyClient` di Flower, ovvero offra un supporto immediato per lavorare con liste di array multidimensionali come formato di rappresentazione di un modello predittivo;
- una strategia `PeerReviewStrategy` che estenda le callback previste dalla classe astratta `Strategy` di Flower aggiungendo opportuni metodi per: configurare un round di review e aggregarne i risultati in maniera simile a quanto previsto da Flower per il round di addestramento, aggregare al termine della fase di review i risultati ottenuti nel corso dei round effettuati ed infine un metodo che verifichi al termine di ciascun round di review se continuare con un altro round di review o terminare l'iterazione corrente del ciclo di apprendimento del server;
- opportuni messaggi `TrainIns`/`TrainRes` e `ReviewIns`/`ReviewRes` che segnalino ai client l'esecuzione di un round di training o di review. Nella sezione successiva si tratta in maniera dettagliata l'implementazione dei messaggi di training e review.

Si riporta in Figura 7 la modellazione con un diagramma della classi UML dei principali componenti di PeerReviewFlower.

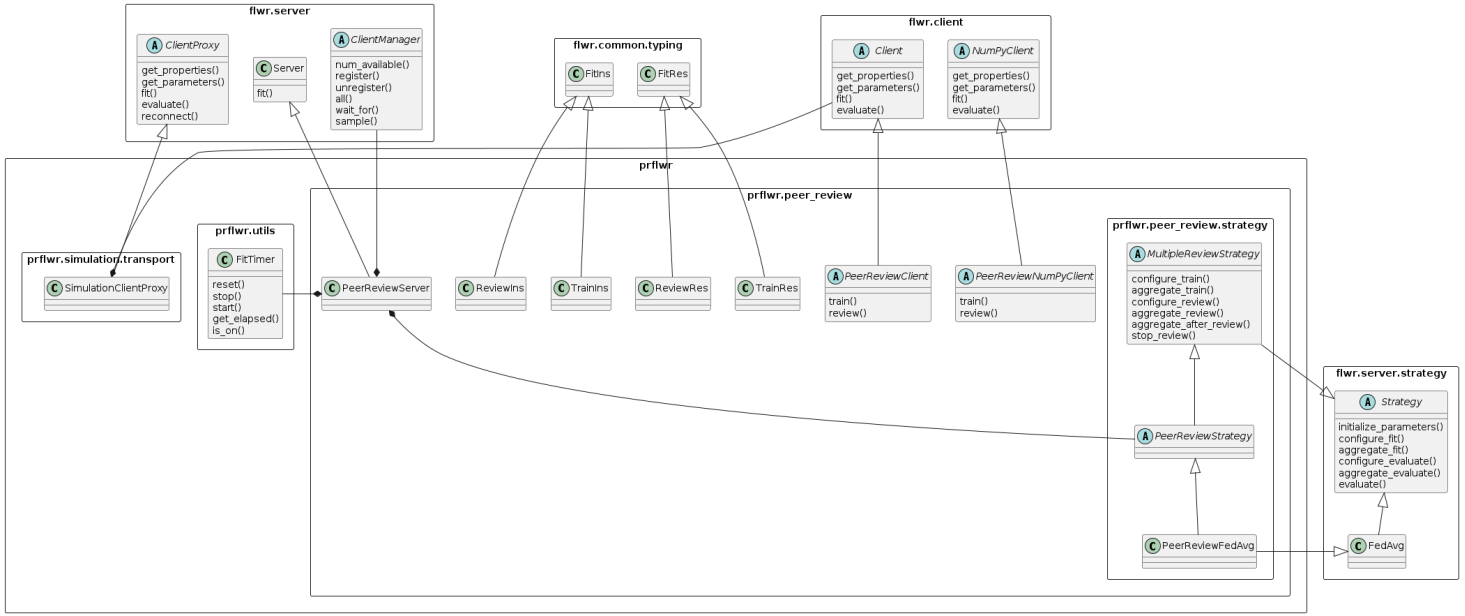

Figura 7: Diagramma delle classi UML che rappresenta i principali componenti di PeerReviewFlower.

Deployment. Il deployment distribuito di un sistema di federated learning con peer review utilizzando PeerReviewFlower prevede un insieme di nodi client e un solo nodo server. Un client è un nodo dotato di un ambiente Python in cui è in esecuzione un'istanza di una classe, implementata dall'utente della libreria, che estende `PeerReviewClient` oppure `PeerReviewNumPyClient`. Il server è invece un nodo provvisto di un ambiente Python in cui esegue un'istanza della classe `PeerReviewServer`. In ambiente distribuito il server comunica con i client tramite gRPC utilizzando il protocollo e i messaggi previsti da Flower. Infine, è necessario fornire al server l'implementazione di una strategia di addestramento federato estendendo la classe `PeerReviewStrategy`, che mette a disposizione della classe `PeerReviewServer` le callback chiamate per configurare la fase di training, di review e di validazione del modello che si vuole addestrare.
Si riporta in Figura 8 il diagramma di deployment del sistema descritto.

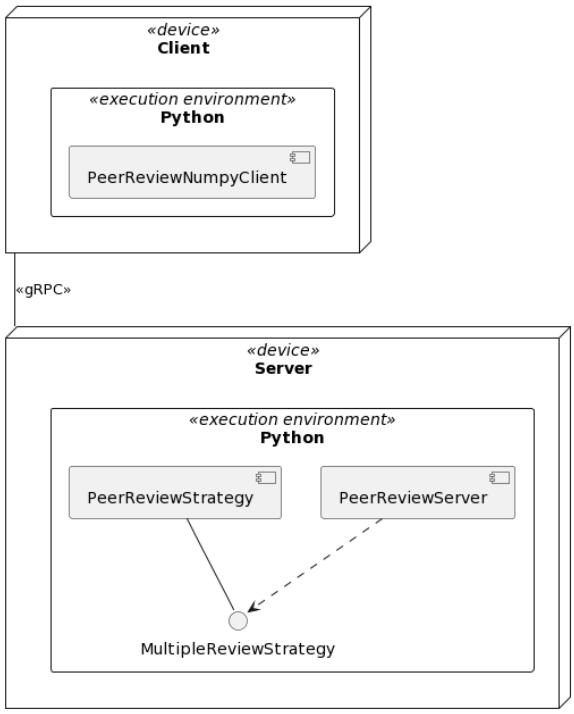

Figura 8: Diagramma di deployment UML che schematizza l'architettura di un sistema di federated learning con peer review distribuito con PeerReviewFlower, viene rappresentato un solo client.

Sul repository del progetto si fornisce un esempio, all'interno della directory `experiments/fedlsbt-dockerized`, di implementazione di un sistema di federated learning con peer review che fa uso di PeerReviewFlower e di Docker per il deployment delle immagini di server e client.

5.2 Comportamento

Il comportamento del sistema durante un processo di addestramento federato con peer review è determinato dall'interazione tra server e client.
Il server ha un comportamento proattivo, esegue il loop di addestramento alternando una fase di training, una fase di review e una fase di validazione delle performance del modello globale che può essere eseguita in maniera sia centralizzata sia federata.
Di seguito si riporta il diagramma di attività UML che rappresenta il ciclo di addestramento federato eseguito dal server.

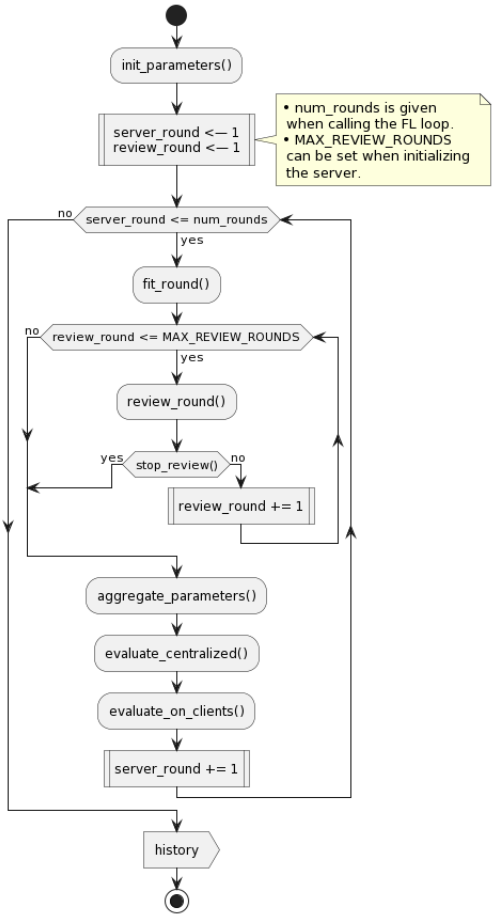

Figura 9: Diagramma di attività UML che formalizza il comportamento del server nell'esecuzione del loop di addestramento federato con peer review.

I client hanno un comportamento reattivo, una volta messi in esecuzione attendono la ricezione di un messaggio, in base alla tipologia del messaggio ricevuto

eseguono il relativo handler e rispondono al server inviando un messaggio che contiene il risultato della computazione eseguita.
Di seguito si riporta il diagramma di stato UML che rappresenta il comportamento reattivo di un'istanza della classe `PeerReviewNumPyClient` che modella un client.

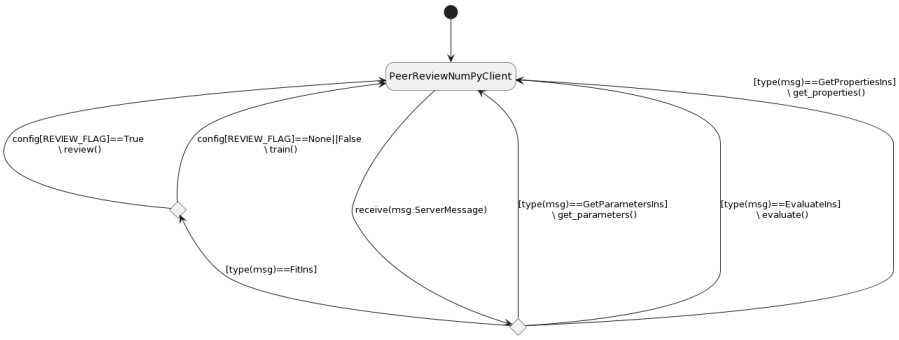

Figura 10: Diagramma di stato UML che formalizza il comportamento di un client, un'entità puramente reattiva.

Anche la strategia di apprendimento, che implementa uno specifico algoritmo di federated learning con peer review come estensione della classe astratta `PeerReviewStrategy`, è un'entità reattiva che fornisce una serie di callback che sono invocate direttamente dal server.

5.3 Interazione

Poiché i componenti di PeerReviewFlower sono specializzazioni dei componenti di Flower le modalità di interazione tra essi non cambiano, quello che cambia è invece la complessità dell'interazione tra entità, che aumenta per consentire l'esecuzione della fase di review.
In Figura 11 si mostra il diagramma di sequenza che rappresenta sia la fase di training sia la fase di review di un'iterazione del ciclo di apprendimento federato con peer review implementato dal server. Nel diagramma sono evidenziate le interazioni tra le entità discusse fino a questo momento: server, client, strategia, messaggi e inoltre si rappresenta anche il `ClientManager` di Flower, un componente che non richiede di essere adattato per supportare il processo di review.
Con riferimento al diagramma di sequenza in Figura 5, che corrisponde alla prima parte del diagramma in Figura 11, si può notare come l'aggiunta della fase di review richieda più interazioni tra server, strategia e client.
Per quanto riguarda invece i componenti di Flower che non richiedono modifiche, come il `ClientManager`, questi mantengono con i nuovi componenti le stesse modalità di interazione che sono previste con i rispettivi componenti della libreria originale.

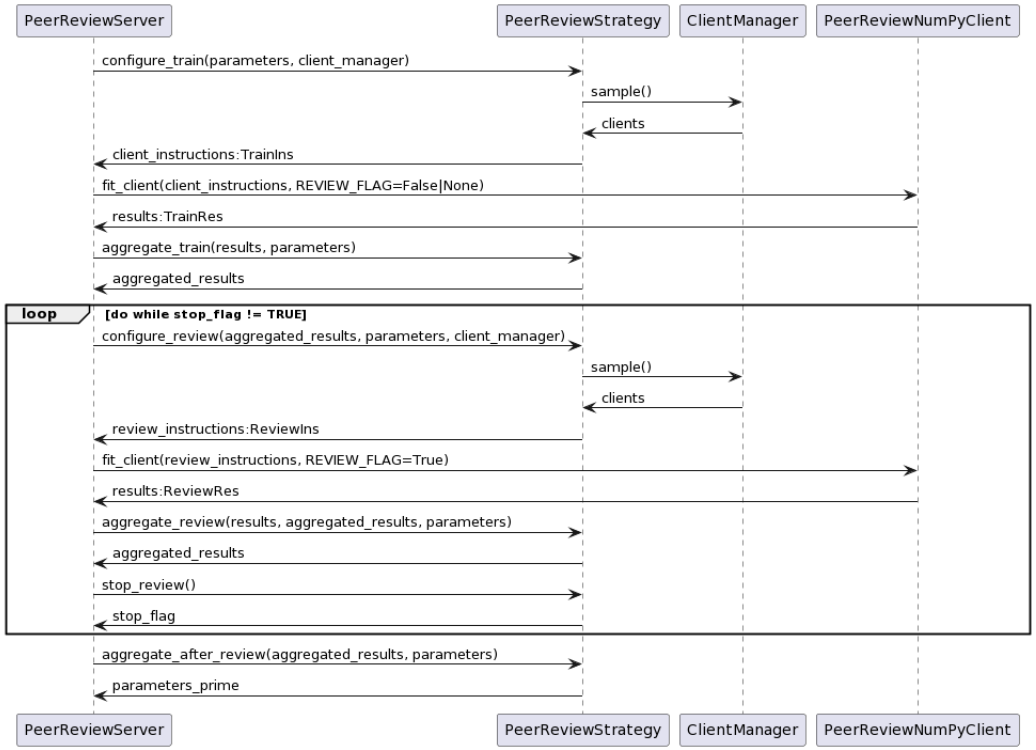

Figura 11: Diagramma di sequenza UML che formalizza l'interazione tra server e client durante un'iterazione del ciclo di addestramento federato con peer review in PeerReviewFlower.

6 Implementazione

In questa sezione si discutono alcuni dettagli implementativi che è utile approfondire anche dal punto di vista degli utilizzatori della libreria.

Messaggi di Training e di Review. Per estendere il comportamento di Flower con una fase di review dei modelli addestrati dai client è opportuno definire una coppia di nuovi messaggi di review `ReviewIns`/`ReviewRes`, simili ai già presenti messaggi `FitIns`/`FitRes` e `EvaluateIns`/`EvaluateRes`, e aggiungere i metodi per la gestione di questi messaggi sia lato server, attraverso l'utilizzo di una strategia con peer review, sia lato client.
L'architettura di Flower dà la possibilità ai programmatori di definire nuovi tipi di messaggi e di integrarli nel proprio protocollo, tuttavia l'implementazione di un nuovi messaggi, sebbene non sia troppo difficoltosa dal punto di vista tecnico, richiede estese modifiche a tutto il codice della libreria originale e rompe la compatibilità con le nuove versioni di Flower.
Un messaggio di tipo `ReviewIns` dovrebbe trasportare i parametri di uno o più aggiornamenti del modello globale in formato binario e un dizionario di configurazione, in questo modo è possibile fornire ai client tutte le informazioni di cui hanno bisogno per la review. Allo stesso modo, un messaggio di tipo `ReviewRes` deve poter trasportare dai client al server un nuovo aggiornamento per il modello globale se l'algoritmo lo richiede e un dizionario contenente varie metriche. Entrambi questi requisiti sono soddisfatti dai messaggi di tipo `FitIns`/`FitRes`, da qui l'idea di implementare i messaggi di tipo `ReviewIns`/`ReviewRes` come sottotipi dei messaggi `FitIns`/`FitRes` nel cui dizionario di configurazione è presente una coppia chiave-valore (`PrConfig.REVIEW_FLAG: True`) che funge da flag per indicare che il messaggio è utilizzato per la review. Per simmetria, anche per quanto riguarda la fase training si definisce una nuova coppia di messaggi `TrainIns`/`TrainRes` che ereditano rispettivamente da `FitIns`/`FitRes` e si differenziano dai messaggi di review in quanto nel dizionario di configurazione la chiave `PrConfig.REVIEW_FLAG` è assente (o tale che `PrConfig.REVIEW_FLAG: False`). Questa scelta implementativa consente di utilizzare il protocollo di Flower per lo scambio di messaggi di tipo `FitIns`/`FitRes` tra client e server, ciascun messaggio poi subirà un routing nei client e nel server verso gli handler opportuni a seconda della presenza e del valore del campo `PrConfig.REVIEW_FLAG` nel dizionario di configurazione.

Strategy Exceptions. Quando si implementa una qualsiasi callback della classe astratta `Strategy` di Flower si potrebbe incorrere in situazioni di errore nelle quali è necessario segnalare al server che qualcosa non è andato a buon fine e l'attuale fase di addestramento o validazione del modello globale è da interrompere. Fare questo in Flower è possibile ma richiede allo sviluppatore di andare a verificare nel codice del server per ogni metodo della classe astratta `Strategy` quali siano i valori di ritorno che il server interpreta come segnale di una situazione di errore. Per questa ragione nell'implementazione di `PeerReviewStrategy` si è voluta includere una serie di eccezioni, un'eccezione per ogni metodo della

classe astratta, che possono essere lanciate dagli utilizzatori della libreria e che vengono intercettate all'interno della classe astratta stessa per essere automaticamente convertite nei valori di ritorno che il server si aspetta come segnale di una situazione di errore.

Simulazione Concorrente. La libreria Flower consente di eseguire in locale simulazioni di procedure di apprendimento federato, utilizzando anche un numero elevato di client simulati, grazie all'utilizzo del framework di computazione concorrente Ray. Le simulazioni eseguite in questo modo però, per via della loro implementazione, presentano un alto overhead di risorse computazionali nel caso in cui si vogliano eseguire simulazioni con un numero non troppo elevato di client e possono portare a problematiche di sovrautilizzo di memoria e disco. Per ovviare a questi problemi, si è inserita in PeerReviewFlower la possibilità di eseguire simulazioni concorrenti che non utilizzano Ray e quindi risultano notevolmente più pratiche con un numero di client contenuto.

7 Utilizzo

In questa sezione si mostra come utilizzare PeerReviewFlower per implementare algoritmi di federated learning con peer review. A titolo di esempio, si considera un'estensione dell'algoritmo FedAvg che ad ogni iterazione sfrutta la fase di review per effettuare una line search utile a determinare l'entità ottimale dell'aggiornamento dei parametri del modello globale, nel seguito si chiamerà questa procedura Federated Line Search (FedLS). In particolare, l'algoritmo proposto modifica la fase dell'aggiornamento del modello globale nel seguente modo: al termine di ogni round di addestramento si aggregano i singoli aggiornamenti prodotti dai client nella stessa maniera in cui procede FedAvg, tuttavia, al contrario di quest'ultimo, l'aggiornamento così prodotto non viene direttamente inglobato nel modello globale ma viene ad esso sommato dopo essere stato scalato per una quantità determinata tramite una procedura di line search eseguita dai client selezionati per la fase di review. Per implementare questo algoritmo in PeerReviewFlower è necessario definire la logica di training e review lato client in una classe che estende `PeerReviewNumPyClient`, dato che come modello predittivo si farà uso di una rete neurale che può essere naturalmente rappresentata come una lista di array NumPy multidimensionali. La logica lato server dovrà essere invece contenuta in una strategia definita in una classe che estende la classe astratta `PeerReviewStrategy`, pertanto dovranno essere implementati sia i metodi definiti nella classe astratta `Strategy` di Flower sia quelli definiti nella classe astratta `MultiplteReviewStrategy` di PeerReviewFlower.
Nel caso dell'esempio realizzato, la classe `FedLS` implementa la strategia lato server. Poiché l'implementazione di alcuni metodi della classe non cambia rispetto all'algoritmo FedAvg, si fa uso dell'implementazione di questi metodi fornita dalla classe `FedAvg` di Flower tramite pattern adapter, mentre per gli altri metodi viene realizzata un'implementazione ad hoc, riportata di seguito.

```
class FedLS(PeerReviewStrategy):

    ...

    def configure_review(
        self, server_round: int, review_round: int,
        parameters: Parameters, client_manager: ClientManager,
        parameters_aggregated: List[Optional[Parameters]],
        metrics_aggregated: List[Dict[str, Scalar]],
    ) -> List[Tuple[ClientProxy, FitIns]]:
        if self.fraction_review == 0.0:
            raise ConfigureReviewException
        config = {}
        if self.on_review_config_fn is not None:
            config = self.on_review_config_fn(server_round, review_round)
        # Prepare review instructions
        config.setdefault("server_round", server_round)
        config.setdefault("eta0", self.max_step_size)
        config.setdefault("gamma", self.step_size_decay)
        config.setdefault("max_evaluations", self.max_evaluations)
        review_ins = ReviewIns(parameters_aggregated[0], config)
        # Sample clients
        sample_size, min_num_clients = self.num_review_clients(
            client_manager.num_available())
        clients = client_manager.sample(
            num_clients=sample_size, min_num_clients=min_num_clients)
        # Return client/config pairs
        return [(client, review_ins) for client in clients]

    def aggregate_review(self, server_round: int, review_round: int,
        results: List[Tuple[ClientProxy, ReviewRes]],
        failures: List[Union[Tuple[ClientProxy, ReviewRes], BaseException]],
        parameters: Parameters,
        parameters_aggregated: List[Optional[Parameters]],
        metrics_aggregated: List[Dict[str, Scalar]],
    ) -> List[Tuple[Optional[Parameters], Dict[str, Scalar]]]:
        if not results:
            raise AggregateReviewException
        if not self.fedavg.accept_failures and failures:
            raise AggregateReviewException
        # Aggregate results
        losses = []
        for i in range(self.max_evaluations):
            aggregated_loss = weighted_loss_avg([
                (review_res.num_examples, review_res.metrics[i])
                for _, review_res in results])
            losses.append(aggregated_loss)
        metrics_aggregated[0].setdefault("step_size",
            self.max_step_size * (self.step_size_decay ** np.argmin(losses)))
        # Return parameters update and step size
        return list(zip(parameters_aggregated, metrics_aggregated))

    def aggregate_after_review(
        self, server_round: int, parameters: Optional[Parameters],
        parameters_aggregated: List[Optional[Parameters]],
        metrics_aggregated: List[Dict[str, Scalar]]
    ) -> Optional[Parameters]:
        weights = parameters_to_ndarrays(parameters)
        gradient = parameters_to_ndarrays(parameters_aggregated[0])
        # Compute update
        step_size = max(self.min_step_size, metrics_aggregated[0]["step_size"])
        for j, tensor in enumerate(gradient):
            weights[j] += step_size * gradient[j]
        # Return new global parameters
        parameters_prime = ndarrays_to_parameters(weights)
        return parameters_prime

    def stop_review(
        self, server_round: int, review_round: int,
        parameters: Parameters, client_manager: ClientManager,
        parameters_aggregated: List[Optional[Parameters]],
        metrics_aggregated: List[Dict[str, Scalar]]
    ) -> bool:
        return True
```

Figura 12: Implementazione della strategia FedLS con PeerReviewFlower.

Per quanto riguarda la logica lato client, la classe che nell'esempio corrente estende `PeerReviewNumPyClient` è chiamata `LSClient` e, rispetto a un client che supporta l'algoritmo FedAvg, prevede l'aggiunta del metodo di review per effettuare un numero prefissato di passi di line search, in cui ad ogni passo si memorizza il valore della loss misurata sul validation set locale e infine si invia al server un dizionario contenente tutte le loss registrate.

```
class FedLSClient(PeerReviewNumPyClient):
    def __init__(self, cid, net, trainloader, valloader):
        self.cid = cid
        self.net = net
        self.trainloader = trainloader
        self.valloader = valloader

    def get_parameters(self, config):
        return get_parameters(self.net)

    def train(self, parameters, config):
        current_round = config["server_round"]
        local_epochs = config["local_epochs"]
        set_parameters(self.net, parameters)
        optimizer = torch.optim.Adam(self.net.parameters())
        train(self.net, self.trainloader, optimizer, local_epochs)
        return get_parameters(self.net), len(self.trainloader), {}

    def review(self, parameters, config):
        current_round = config["server_round"]
        max_evaluations = config["max_evaluations"]
        gamma = config["gamma"]
        eta = config["eta0"]
        losses = []
        for i in range(max_evaluations):
            loss, num_examples, _ = self.evaluate(
                [eta * param for param in parameters], {}
            )
            losses.append(loss)
            eta = eta * gamma
        review_metrics = dict(enumerate(losses))
        return [], num_examples, review_metrics

    def evaluate(self, parameters, config):
        set_parameters(self.net, parameters)
        loss, accuracy = test(self.net, self.valloader)
        return float(loss), len(self.valloader), {}
```

Figura 13: Implementazione del client FedLS con PeerReviewFlower.

Una volta implementata la strategia di apprendimento e la classe che modella i client, è possibile eseguire un intero processo di apprendimento federato in maniera simulata su una singola macchina oppure in maniera distribuita grazie al supporto dato da Flower. Inoltre è possibile personalizzare diversi parametri della procedura come: numero di client partecipanti e numero di client utilizzati durante la fase di addestramento e di review.
Il codice sopra riportato, insieme alle parti omesse per questioni di spazio, si trova in un notebook Jupyter sul repository della libreria, all'interno della directory `"experiments/fedls"`.

8 Sperimentazione

Per dimostrare le possibilità fornite dalla libreria PeerReviewFlower e soprattutto dal federated learning con peer review, si mostra un'applicazione sperimentale di questo approccio che consente di implementare l'addestramento federato di modelli di apprendimento di tipo gradient boosted decision trees. Per questo tipo di modelli di apprendimento, che sono allo stato dell'arte nel machine learning centralizzato su dati tabulari, sono stati proposti in letteratura alcuni algoritmi di apprendimento federato, tuttavia risultano tutti ancora poco diffusi e oggetto di ricerca. L'algoritmo proposto di seguito realizza l'addestramento federato di modelli GBDT sfruttando la fase di review del federated learning con peer review.
Il codice per questo esperimento si trova sotto forma di notebook Jupyter sul repository del progetto, all'interno della directory `experiments/fedlsbt`.

Gradient Boosting. Il gradient boosting è un algoritmo di machine learning utilizzato per task sia di regressione sia di classificazione che costruisce un modello predittivo come un ensemble di modelli più semplici detti weak learners. Tipicamente, la scelta di questi modelli ricade sugli alberi di decisione e l'algoritmo prende il nome di gradient boosted decision trees (GBDT). Un modello GBDT viene costruito con un procedimento iterativo che combina ad ogni passo uno o più alberi addestrati in maniera tale da consentire l'ottimizzazione di arbitrarie funzioni obiettivo purché differenziabili [3]. In particolare, il modello appreso da un algoritmo GBDT ha la seguente forma:

$$F(x) = \gamma_0 + \sum_{m=1}^{M} \gamma_m h_m(x)$$

dove γ_0 è una costante, mentre $h_m(x)$ è un albero di decisione e il suo contributo alla predizione del modello è pesato dalla costante γ_m. Per addestrare un modello GBDT, dato un training set $D = \{(x_i, y_i), i = 1...N\}$ e una funzione obiettivo differenziabile $L(y, F(x))$, ad ogni iterazione $m = 1...M$ si costruisce un modello $F_m(x)$ con un approccio greedy:

$$F_0(x) = \gamma_0 = \arg\min_{\gamma} \sum_{i=1}^{N} L(y_i, \gamma)$$
$$F_m(x) = F_{m-1}(x) + \arg\min_{h} \sum_{i=1}^{N} L(y_i, F_{m-1}(x_i) + h(x_i))$$

Approssimando al primo ordine la funzione obiettivo si ha:

$$h_m(x) = \arg\min_{h} \sum_{i=1}^{N} h(x) \frac{\partial L(y_i, F_{m-1}(x_i))}{\partial F_{m-1}(x_i)}$$

$$F_m(x) = F_{m-1}(x) - \gamma_m h_m(x)$$

quindi, a meno di una costante moltiplicativa:

$$h_m(x_i) \approx -\frac{\partial L(y_i, F_{m-1}(x_i))}{\partial F_{m-1}(x_i)}$$

Pertanto, ad ogni iterazione si calcolano i gradienti di tutti i campioni del training set rispetto alla funzione obiettivo e si addestra il modello $h_m(x)$ a predire l'opposto di tali valori. In questo modo, l'algoritmo esegue una sorta di discesa del gradiente dove la costante γ_m rappresenta il learning rate.
Di seguito si riporta l'algoritmo generale di addestramento di un modello gradient boosting.

Algorithm 1 Gradient Boosting

Input: training set $D = \{(x_i, y_i)\}_{i=1}^{N}$, number of iterations M, differentiable loss function $L(y, F(x))$.
Output: learned model $F_M(x)$.
Initialize model with a constant value:
$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{N} L(y_i, \gamma)$$
for $m \leftarrow 1$ to M **do**
Compute pseudo-residuals:
$$r_i = -\left[\frac{\partial L(y_i, F_{m-1}(x_i))}{\partial F_{m-1}(x_i)}\right], i = 1...N$$
Fit a weak learner $h_m(x)$ using the training set $\{(x_i, r_i)\}_{i=1}^{N}$:
$$h_m(x) = \arg\min_{h} \sum_{i=1}^{N} [r_i - h(x_i)]^2$$
Compute learning rate γ_m with line search:
$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{N} L(y_i, F_{m-1}(x_i) + \gamma h_m(x_i))$$
Update model: $F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$
end for

Federated Least Squares Boosted Trees (FedLSBT). Partendo dalla formulazione originale dell'algoritmo di gradient boosting, si è sviluppato un algoritmo, che prende il nome di Federated Least Squares Boosted Trees (FedLSBT), adatto all'applicazione in un contesto federato e che richiede necessariamente una fase di review degli aggiornamenti del modello globale prodotti dai client. Questo algoritmo consente di addestrare un modello GBDT globale condiviso da un insieme di client senza che questi si scambino i dati durante il training.
Come weak learners vengono sfruttati alberi di decisione addestrati con un procedimento randomizzato, chiamati Extremely Randomized Trees (ExtraTrees) [4]. L'utilizzo degli ExtraTrees è dovuto al fatto che la scelta randomizzata dei valori delle feature su cui effettuare gli split dei nodi contribuisce ulteriormente a garantire il mantenimento della privacy dei dati dei client.
Si consideri uno scenario in cui si hanno K client, un dataset D partizionato in K partizioni D_k ciascuna memorizzata in un client diverso e una funzione obiettivo differenziabile $L(y, F(x))$, l'obiettivo del federated learning è addestrare un modello globale $F(x)$ ottimizzando una funzione definita come la media della funzione obiettivo valutata su tutti i client:
$$F^*(x) = \arg\min_{F} \frac{1}{K} \sum_{k=1}^{K} \frac{1}{|D_k|} \sum_{(x_i, y_i) \in D_k} L(y_i, F(x_i))$$

Come GBDT, anche FedLSBT costruisce il modello predittivo $F(x)$ come un ensemble additivo di T alberi $h_t(x)$ in cui il contributo di ciascun albero è pesato da una costante γ_t:

$$F(x) = \sum_{t=1}^{T} \gamma_t h_t(x)$$

Ad ogni iterazione $m = 1...M$ di addestramento federato viene eseguito un round di training nel quale viene campionato un sottoinsieme S_T di client e un solo round di review in cui è coinvolto un sottoinsieme S_R di client (è possibile prendere $S_R = S_T$ oppure effettuare un nuovo campionamento dei client).
L'idea alla base dell'algoritmo consiste nell'aggiungere al modello globale per ogni ciclo di addestramento $|S_T|$ alberi appresi durante il round di training, ciascuno pesato da una costante determinata durante la fase di review.

$$F_m(x) = F_{m-1}(x) + \sum_{k=1}^{|S_T|} \gamma_k h_k(x)$$

Ogni client k che partecipa al round di training addestra un albero $h_k(x)$ sul training set locale per predire l'opposto del gradiente della funzione obiettivo rispetto alla predizione del modello globale e invia al server l'aggiornamento così prodotto.
Gli aggiornamenti ricevuti dal server al termine della fase di training vengono poi utilizzati nella fase di review per approssimare i gradienti della funzione obiettivo rispetto alla predizione del modello globale considerando tutti i campioni presenti nei dataset privati $D_{k'}$ dei client selezionati per la review, risolvendo il seguente problema ai minimi quadrati rispetto al valore delle costanti γ_k:

$$\min_{\gamma_0,\gamma_1,...,\gamma_{|S_T|}} \left[\sum_{(x,y)\in D_R} \left(\frac{\partial L(y,F_{m-1}(x))}{\partial F_{m-1}(x)} - \frac{1}{|S_T|} \sum_{k=1}^{|S_T|} \gamma_k h_k(x) \right)^2 \right]$$
$$\text{con } D_R = \bigcup_{k'\in S_R} D_{k'}$$

Un problema di questo tipo può essere risolto efficientemente in un contesto di federated learning con peer review per mezzo dell'algoritmo sotto riportato.
In maniera simile all'algoritmo di gradient boosting, la fase di review consente quindi di effettuare una line search utile a determinare il peso ottimale del contributo di ciascun weak learner alla predizione del modello globale. Tuttavia, questi valori sono determinati considerando generalmente solo un sottoinsieme di tutti i client disponibili, per questa ragione è possibile aggiungere un fattore di regolarizzazione al modello scalando ognuna delle costanti γ_k apprese all'iterazione m per un fattore moltiplicativo η che prende il nome di learning rate (o fattore di shrinkage).

Algorithm 2 Federated Least Squares Boosted Trees (FedLSBT)

Input: clients set S indexed $1...K$, local training set for each client $D_k = \{(x_i, y_i)\}_{i=1}^{N_k}$, differentiable loss function $L(y, F(x))$, number of iterations M, fraction of client participating in the training round f_T, fraction of client participating in the review round f_R, learning rate η.

Output: learned global model $F_M(x)$.

procedure FEDLSBT(S, M, f_T, f_R)
  Initialize model with a constant value: $F_0(x) = 0$
  for $m \leftarrow 1$ to M **do**
    Sample a subset of clients $S_T \subseteq S : |S_T| = \lfloor f_T \cdot |S| \rfloor$
    Initialize set of learned updates to the global model: $H_m \leftarrow \emptyset$
    for $k \in S_T$ **do**
      Send global model $F_{m-1}(x)$ to client k
      Update H_m: $H_m \leftarrow H_m \cup \text{CLIENTTRAIN}(k, F_{m-1}(x))$
    end for
    Sample a subset of clients $S_R \subseteq S : |S_R| = \lfloor f_R \cdot |S| \rfloor$
    Initialize: $C \leftarrow 0 \in \mathbb{R}^{|S_T| \times |S_T|}$, $R \leftarrow 0 \in \mathbb{R}^{|S_T|}$
    for $k' \in S_R$ **do**
      Send $F_{m-1}(x)$ and H_m to client k'
      $R_{k'}, C_{k'} \leftarrow \text{CLIENTREVIEW}(k', F_{m-1}(x), H_m)$
      Update R and C: $R \leftarrow R + R_{k'}$, $C \leftarrow C + C_{k'}$
    end for
    Compute learning rates: $\gamma = \frac{1}{|S_T|}(C^T C)^{-1} R$, $\gamma \in \mathbb{R}^{|S_T|}$
    Update model: $F_m(x) = F_{m-1}(x) + \eta \sum_{i=1}^{|S_T|} \gamma_i h_i(x)$, $h_i \in H_m$
  end for
end procedure

procedure CLIENTTRAIN(k, $F(x)$)
  Compute pseudo-residuals: $r_i = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right], (x_i, y_i) \in D_k$
  Fit a tree $h_k(x)$ using the training set $\{(x_i, r_i)\}_{i=1}^{N_k}, x_i \in D_k$
  Send $h_k(x)$ to the server
end procedure

procedure CLIENTREVIEW(k, $F(x)$, H)
  Compute pseudo-residuals: $r_i = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right], (x_i, y_i) \in D_k$
  for $h_j(x) \in H$ **do**
    Compute predicted residuals on D_k: $\hat{r}_j = [h_j(x_0), h_j(x_1), ..., h_j(x_{N_k})]^T$
  end for
  Initialize matrix: $P \in \mathbb{R}^{N_k \times |S_T|} \leftarrow \begin{bmatrix} \hat{r}_0 & \hat{r}_1 & ... & \hat{r}_{|H|} \end{bmatrix}$
  Compute: $C_k \leftarrow P^T P$
  Compute: $R_k \leftarrow P^T r$
  Send R_k and C_k to the server
end procedure

Regressione. Per dimostrare il funzionamento dell'algoritmo FedLSBT nella pratica si è realizzato un esperimento con un task di regressione utilizzando il celebre dataset California Housing Prices [12]. L'obiettivo del task è la predizione di una variabile continua, il prezzo medio di un'abitazione per ogni distretto della California, a partire da otto feature che riguardano le caratteristiche delle abitazioni nel distretto. Per l'addestramento del modello si è scelta una funzione obiettivo di tipo Mean Squared Error (MSE) e per simulare al meglio un contesto reale di federated learning si è realizzato un partizionamento non IID dei dati sulla base del valore degli attributi di latitudine e longitudine dei campioni. L'esperimento è stato eseguito sotto forma di simulazione impostando i seguenti parametri:

- Numero di client: **30**
- Numero di iterazioni di apprendimento federato: **50**
- Frazione di client per ogni round di training: $\frac{1}{3}$
- Frazione di client per ogni round di review: $\frac{1}{3}$

Dal punto di vista implementativo, si è fatto uso degli alberi di regressione forniti dalla libreria scikit-learn, la cui gestione nel framework di federated learning sviluppato richiede più lavoro rispetto ai modelli parametrici tradizionalmente impiegati. Come detto in precedenza infatti, con Flower, e di conseguenza con la sua estensione PeerReviewFlower, è possibile lavorare facilmente con modelli predittivi che possano essere naturalmente rappresentati come array multidimensionali. Nel caso degli alberi di decisione però questa rappresentazione non è immediata. Fortunatamente, il server non gestisce i modelli esplicitamente come array multidimensionali ma gestisce la loro forma serializzata come oggetti della classe `Parameters` e anche per quanto riguarda i client la classe `PeerReviewClient` gestisce i modelli nella loro forma serializzata. Pertanto, per gestire modelli della classe `ExtraTreeRegressor` di scikit-learn è stato sufficiente implementare le relative funzioni di serializzazione e deserializzazione che vengono utilizzate sia dai client sia dal server.
Al fine di valutare le performance del modello addestrato con algoritmo FedLSBT rispetto al modello GBDT implementato nella libreria scikit-learn e addestrato in maniera centralizzata, si è valutato l'errore commesso su un dataset di test composto da un sottoinsieme dei campioni del dataset originale non utilizzati in fase di addestramento. Il modello di gradient boosting centralizzato è stato addestrato in maniera tale da rendere i due algoritmi confrontabili, si è quindi impostato: un numero di stimatori pari al numero di alberi costruiti dall'algoritmo FedLSBT, un numero di campioni utilizzati per l'addestramento di ciascun albero pari alla media dei campioni presenti nel dataset di ciascun client, la funzione obiettivo MSE e inizializzazione del modello a zero.
Di seguito si riportano i risultati ottenuti:

	r^2 score	MSE
Baseline	-	0.93
GradientBoostingRegressor (sklearn)	0.72	0.26
FedLSBT	0.63	0.34

In tabella sono riportate due metriche di valutazione delle performance: r^2 score, una metrica significativa soprattutto per problemi di regressione dove la dipendenza tra variabile target e features è di tipo lineare, e MSE. Dai risultati ottenuti si vede che il modello centralizzato risulta più performante rispetto al modello federato FedLSBT che riesce comunque a raggiungere una performance pari al 88% del modello centralizzato.
La differenza nelle prestazioni tra i due modelli è del tutto attesa in un contesto di federated learning ed è prevalentemente da attribuire alla caratteristica distribuzione non IID dei dati dei client che partecipano all'addestramento federato.

Classificazione. Per dimostrare l'applicabilità generale dell'algoritmo proposto anche nel caso di un task di classificazione, quindi con una differente funzione obiettivo, si è realizzato un esempio in cui il precedente problema di regressione viene trasformato in un problema di classificazione binario assegnando ad ogni campione (x_i, y_i) una classe in base alla seguente funzione:

$$C(x_i) = \begin{cases} +1 & \text{if } y_i \geq median(Y) \\ -1 & \text{if } y_i < median(Y) \end{cases}$$

La funzione obiettivo utilizzata per il task di classificazione è la Binary Cross Entropy e il resto dell'esperimento è condotto in maniera analoga al task di regressione per quanto riguarda il partizionamento dei dati e i parametri dell'addestramento federato.
Per il confronto con un algoritmo di gradient boosting centralizzato si è utilizzato un classificatore GBDT implementato nella libreria sckit-learn con i seguenti iperparametri: un numero di stimatori pari al numero di alberi costruiti dall'algoritmo FedLSBT, un numero di campioni utilizzati per l'addestramento di ciascun albero pari alla media dei campioni presenti nel dataset di ciascun client, BCE come funzione obiettivo e inizializzazione a zero.
Di seguito si riportano i risultati ottenuti:

	accuratezza	BCE
Baseline	0.50	-
GradientBoostingClassifier (sklearn)	0.81	2.47
FedLSBT	0.80	0.50

In questo caso, si osserva che l'accuratezza di classificazione della versione centralizzata dell'algoritmo GBDT è pari al modello FedLBST che riesce anche ad ottenere un valore della loss BCE decisamente migliore.

Conclusioni sull'esperimento. L'algoritmo proposto consente di addestrare un modello di gradient boosting federato rispettando la privacy dei dati memorizzati sui client grazie al fatto che i dati non lasciano mai i dispositivi in cui sono salvati.
Le performance misurate nei test effettuati sono risultate molto soddisfacenti, pari o di poco inferiori ai modelli addestrati in maniera centralizzata impostando gli iperparametri in modo tale da rendere confrontabili le due versioni. È possibile che una ricerca degli iperparametri ottimi con cross-validation porti a risultati degli algoritmi centralizzati ancora migliori rispetto alle versioni federate, tuttavia i risultati ottenuti sono un punto di partenza molto significativo per un ulteriore studio del gradient boosting federato.

9 Validazione

In fase di definizione degli obiettivi del progetto si erano individuati due criteri secondo cui valutare il lavoro finito, la qualità del codice prodotto e l'efficacia della libreria nel fornire uno strumento utile a sperimentare il federated learning con peer review.
Per quanto riguarda la qualità del codice, si considera rispettato il mantenimento della compatibilità della libreria sviluppata con la libreria utilizzata come base di partenza, compatibilità garantita dal design di PeerReviewFlower, che favorisce l'uso congiunto dei suoi componenti con i componenti di Flower. Questo ha effettivamente facilitato lo sviluppo in quanto è stato possibile integrare con poco sforzo le modifiche che Flower ha subito nel corso del tempo con le funzionalità aggiuntive introdotte da questo lavoro.
Per quanto riguarda invece l'efficacia della libreria sviluppata, questa è confermata dal fatto che si è mantenuta la semplicità, la flessibilità e le modalità d'utilizzo di Flower, caratteristiche che hanno permesso la realizzazione con buoni risultati delle sperimentazioni discusse in precedenza.

9.1 Testing

Il testing riveste un ruolo fondamentale in questo progetto in quanto, utilizzando come base di partenza una libreria in fase di veloce sviluppo, è stato necessario verificare che l'estensione realizzata non andasse in conflitto con Flower e al tempo stesso stesse al passo con i cambiamenti che Flower ha subito nel corso del tempo. I test sono stati organizzati in test d'integrazione e test d'unità.
I test di integrazione hanno lo scopo di verificare che semplici test di utilizzo sia di Flower sia di PeerReviewFlower funzionino correttamente e producano i risultati attesi, quindi si è realizzato:

- un test di addestramento centralizzato di un modello di rete neurale con PyTorch da utilizzare come esempio di una corretta pipeline di machine learning,

- un test di addestramento federato con Flower basato sulla pipeline centralizzata,
- un test di addestramento federato con peer review che utilizza un'implementazione con peer review dell'algoritmo FedAvg e pertanto è del tutto equivalente nei risultati al test precedente,
- un test che verifica l'equivalenza dei risultati dei due precedenti test,
- un test che esegue una simulazione del test federato con peer review descritto sopra, utilizzando il supporto alla simulazione fornito da Flower.

I test d'unità hanno lo scopo di verificare che i singoli componenti software sviluppati, isolati dal resto del sistema, si comportino nella maniera desiderata a fronte di un certo numero di scenari di esecuzione. Si è costruito uno unit test per ciascuno dei componenti più importanti di PeerReviewFlower, tra cui: server, client, strategia e simulazione concorrente.
Si è testata la coverage dei test utilizzando la libreria `coverage.py` per Python, di seguito sono riportati i risultati relativi ai soli file contenuti nei sotto-package del package `prflwr`:

```
Name                                              Stmts   Miss  Cover
---------------------------------------------------------------------
prflwr/__init__.py                                    0      0   100%
prflwr/peer_review/__init__.py                        6      0   100%
prflwr/peer_review/client.py                         25      0   100%
prflwr/peer_review/config.py                          4      0   100%
prflwr/peer_review/numpy_client.py                   20      0   100%
prflwr/peer_review/server.py                        187     15    91%
prflwr/peer_review/strategy/__init__.py               4      0   100%
prflwr/peer_review/strategy/exceptions.py            22      0   100%
prflwr/peer_review/strategy/fedavg.py                61     10    84%
prflwr/peer_review/strategy/prmultrev.py             20      0   100%
prflwr/peer_review/strategy/strategy.py              50      0   100%
prflwr/peer_review/typing.py                         34      6    82%
prflwr/simulation/__init__.py                         2      0   100%
prflwr/simulation/app.py                             27      6    78%
prflwr/simulation/transport/__init__.py               0      0   100%
prflwr/simulation/transport/client_proxy.py          41     15    63%
prflwr/utils/__init__.py                              3      0   100%
prflwr/utils/dataset.py                              29      1    97%
prflwr/utils/timer.py                                28      0   100%
---------------------------------------------------------------------
TOTAL                                               563     53    91%
```

Figura 14: Report della coverage dei test relativamente ai componenti più rilevanti di PeerReviewFlower.

Continuous Integration. Al fine di automatizzare l'esecuzione dei test si è impostata una pipeline di continuous integration sul repository GitHub del progetto utilizzando GitHub Actions. La pipeline di continuous integration esegue tre gruppi di test chiamati workflow: uno comprende i test d'integrazione e gli unit test dei singoli componenti, un'altro ha l'obiettivo di verificare la correttezza dell'installazione del framework realizzato in un ambiente virtuale pulito con

Python ed infine un'ultimo workflow testa la build della documentazione della libreria.

10 Conclusioni

Nel corso di questo progetto si è realizzato un framework per la sperimentazione di un nuovo approccio al federated learning, presentato e discusso nella presente relazione, chiamato Peer Review Federated Learning. Per questo scopo si è partiti da un promettente framework di federated learning già esistente chiamato Flower, che si è esteso in maniera tale da implementare il meccanismo di peer review. Insieme all'implementazione del framework si sono prodotti anche i test relativi ai singoli componenti software e alla loro integrazione. I test sono quindi stati utilizzati per impostare una pipeline di continuous integration sul repository GitHub del framework in maniera tale da migliorare le garanzie di qualità del codice prodotto. Per valutare l'efficacia del framework nell'implementazione di sistemi di federated learning con peer review e al fine di mostrare la conversione di una pipeline tradizionale di machine learning in una pipeline federata con peer review, si è realizzata a una serie di notebook Jupyter che comprendono: un esempio di addestramento centralizzato, un esempio di addestramento federato con Flower e un esempio di federated learning con peer review utilizzando PeerReviewFlower con la strategia di apprendimento FedLS. Infine, la sperimentazione del federated learning con peer review ha portato all'ideazione di un nuovo algoritmo gradient boosted decision trees federato, denominato FedLSBT, che richiede necessariamente una fase di review e, sotto determinate condizioni relative alla scelta degli iperparametri, permette di ottenere performance pari ai modelli GBDT centralizzati.

10.1 Lavori futuri

Dato che il framework realizzato risulta adatto alla sperimentazione e il federated learning con peer review ha offerto spunti molto interessanti permettendo l'implementazione dell'algoritmo FedLSBT, il lavoro svolto per questo progetto può offrire diverse nuove prospettive di sperimentazione, sia per quanto riguarda l'applicazione di peer review federated learning all'addestramento di reti neurali e altri modelli parametrici sia per quanto riguarda l'ottimizzazione e l'estensione dell'algoritmo FedLSBT.

Riferimenti bibliografici

[1] Adap. Flower documentation. `https://flower.dev/docs/`.

[2] Daniel J. Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Javier Fernandez-Marques, Yan Gao, Lorenzo Sani, Kwing Hei Li, Titouan Parcollet, Pedro Porto Buarque de Gusmão, and Nicholas D. Lane. Flower: A friendly federated learning research framework. 2020.

[3] Jerome H. Friedman. Stochastic gradient boosting. 1999.

[4] P. Geurts, D. Ernst., and L. Wehenkel. Extremely randomized trees. 2006.

[5] Google. Federated learning: Collaborative machine learning without centralized training data. `https://www.tensorflow.org/federated`.

[6] Google. Tensorflow federated. `https://ai.googleblog.com/2017/04/federated-learning-collaborative.html`.

[7] gRPC Authors. grpc: A high performance, open source universal rpc framework. `https://grpc.io/`, 2022.

[8] Chaoyang He, Songze Li, Jinhyun So, Xiao Zeng, Mi Zhang, Hongyi Wang, Xiaoyang Wang, Praneeth Vepakomma, Abhishek Singh, Hang Qiu, Xinghua Zhu, Jianzong Wang, Li Shen, Peilin Zhao, Yan Kang, Yang Liu, Ramesh Raskar, Qiang Yang, Murali Annavaram, and Salman Avestimehr. Fedml: A research library and benchmark for federated machine learning. 2020.

[9] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. 2016.

[10] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging ai applications. 2018.

[11] Theo Ryffel, Andrew Trask, Morten Dahl, Bobby Wagner, Jason Mancuso, Daniel Rueckert, and Jonathan Passerat-Palmbach. A generic framework for privacy preserving deep learning. 2018.

[12] Scikit-learn. California housing dataset. `https://scikit-learn.org/stable/datasets/real_world.html#california-housing-dataset`.