Corso di Laurea Magistrale in
Computer Engineering

Federated Learning Architecture

Alessandro Sieni
Amedeo Pochiero
Roberto Magherini

Indice

Capitolo 1

Specifiche

1.1 Descrizione Problema

Il sistema da noi presentato ha come obiettivo quello di sfruttare il concetto di Federated Learning per un sistema distribuito composto da più nodi e un sink centrale. Lo scopo è quello di usare la potenza di calcolo disponibile ai bordi della rete per un'elaborazione iniziale dei dati che verranno usati per tecniche di Machine Learning. L'idea è quella di permettere ai nodi di creare un proprio modello a partire dai dati che ricevono dai propri sensori, in modo che sia successivamente inoltrato al sink che si occuperà di unire tutti i modelli ricevuti dai nodi, ottenendo così un modello più accurato, eventualmente inviato ai nodi.

1.2 Comunicazione Asincrona Nodi-Sink

I nodi dovranno comunicare con il sink in modo asincrono in quanto non è possibile stabilire a priori il momento esatto in cui i nodi stessi invieranno il modello. Questo è dovuto al fatto che la generazione del modello non avviene ad un ritmo costante ma dipende dal numero di dati ricevuti, i quali sono rilevati ad intervalli variabili.

1.3 Comunicazione Asincrona Sink-Nodi

Una volta che i modelli sono stati uniti sul sink con un algoritmo di merging apposito, si può decidere di comunicarlo ai nodi in modo che essi lo possano utilizzare sui dati che rileveranno in futuro. Anche questa comunicazione è di tipo asincrono, in quanto il merging dipende dall'arrivo asincrono di tutti i modelli sul sink. Pertanto i nodi

devono essere costantemente pronti a ricevere e a sostituire il proprio modello con quello del sink, considerato più accurato. Per fare ciò si prevede l'utilizzo di un meccanismo Publish&Subscribe in cui i nodi si comporteranno come subscribers che si abbonano al servizio offerto dall'unico publisher del sistema, vale a dire il sink.

1.4 Gestione concorrenza della ricezione dati sui nodi

I nodi sono in grado di ricevere dati o eventi da una o più sorgenti. In questo scenario è necessaria una corretta gestione della concorrenza che viene a crearsi tra le molteplici sorgenti, le quali andranno a scrivere sulla stessa destinazione. Una volta che il nodo ha ottenuto una quantità di dati sufficienti, si può procedere alla creazione o all'aggiornamento del modello. Il momento in cui la modifica del modello viene eseguita dipende non solo dalla quantità di dati ricevuti, ma anche dal valore stesso, in quanto alcuni valori potrebbero non influenzarlo mentre altri potrebbero essere abbastanza significativi da farlo variare.

1.5 Variazione del numero dei nodi

Il sistema dovrà essere in grado di poter variare il numero di nodi in tempo reale, mantenendo attiva la comunicazione con il sink centrale. Si prevede quindi l'utilizzo di tecnologie apposite con protocolli di tipo Publish&Subscribe, in cui i nodi recitano la parte dei publisher, in quanto generano i modelli da essere inviati, mentre il sink risulta essere l'unico subscriber del sistema dato che si occupa della raccolta dei modelli.

1.6 Soluzioni Proposte

- Rabbit MQ: è un middleware per la gestione dei messaggi di tipo asincrono che sarà utilizzato per la comunicazione asincrona tra i nodi e il sink. Questo meccanismo si basa sul concetto di produttore-consumatore in cui il consumatore avrà una coda per ogni tipologia di modello, la quale viene riempita in modo asincrono dai produttori, in questo caso i nodi. Con questo meccanismo verrà gestita anche la comunicazione in senso opposto, da sink a nodo, per la distribuzione dei modelli più accurati. In questo caso il sink sarà il produttore che inserirà in una coda i dati da inviare a tutti i consumatori.

- Per la gestione della concorrenza sulla struttura dati dei nodi sarà utilizzato un algoritmo apposito in modo da evitare Race Condition e Starvation.

Capitolo 2

Data Collector

Capitolo 3

Comunicazione tra i Nodi e il Sink

3.1 Implementazione

3.1.1 abstract class CommunicationModelHandler

Il software creato si appoggia sulle Java API di *RabbitMQ* che permette una facile gestione di message queueing e per via delle somiglianza tra le azioni da svolgere sia sul Sink che sui Nodi, è stata implementata una classe astratta *CommunicationModelHandler* 3.1 che mantiene delle informazioni utilizzati nelle interazioni con il server di RabbitMQ e i nomi delle *Queue*, comuni a tutti i nodi. Inoltre, il costruttore inizializza una connessione che il Server di RabbitMQ configurato sulla porta di default 5672 e richiama le 3 funzioni per l'inizializzazione delle strutture su cui verrano scambiati i dati:

- *initRPC()*
- *initSinkToNode()*
- *initNodeToSink()*

Ognuna di queste verrà definita dal Nodo e dal Sink in modo da rispettare il proprio ruolo rispetto alla struttura in questione.

```
1 package it.unipi.cds.federatedLearning;
2
3 import com.rabbitmq.client.Channel;
4 import com.rabbitmq.client.ConnectionFactory;
5
```

```
6  public abstract class CommunicationModelHandler {
7
8      protected final String RPC_NODE_TO_SINK_QUEUE_NAME = "RPC_QUEUE";
9      protected final String NODE_TO_SINK_QUEUE_NAME = "MODELS_QUEUE";
10     protected final String SINK_TO_NODE_EXCHANGE_NAME = "NEW_MODEL_QUEUE";
11
12     protected ConnectionFactory factory;
13     protected Channel channelNodeSink;
14     protected Channel channelSinkNode;
15     protected Channel channelRPC;
16     protected ModelReceiver receiver;
17
18     public CommunicationModelHandler(String hostname) {
19         this.factory = new ConnectionFactory();
20         factory.setHost(hostname);
21         factory.setVirtualHost("cds/");
22         factory.setUsername("cdsAdmin");
23         factory.setPassword("cds");
24
25         initRPC();
26         initSinkToNode();
27         initNodeToSink();
28     }
29     protected abstract void initNodeToSink();
30     protected abstract void initSinkToNode();
31     protected abstract void initRPC();
32     public abstract void receiveModel(Model deliveredModel);
33     public abstract void sendModel();
34 }
```

Listing 3.1: CommunicationModelHandler

3.1.2 NodeCommunicationModelHandler

Questa classe è l'implementazione della *CommunicationModelHandler* utilizzata su ogni Nodo per la gestione della comunicazione con il Sink. Ai campi membri della super classe viene aggiunto un intero *NodeID* che contiene l'identificativo del nodo. La classe definisce le funzioni astratte nel seguente modo:

3.1.2.1 initRPC()

Questa funzione crea un canale per comunicare con il Server RPC presente sul Sink attraverso cui si chiama la Remote Procedure per «registrare» il Nodo nel sistema, il

quale ottiene un identificativo univoco come risposta. Tale ID è usato per distinguere i nodi tra di loro e i loro relativi modelli.

```
1     @Override
2     protected void initRPC() {
3         try {
4             Connection connectionRPC = factory.newConnection();
5             channelRPC = connectionRPC.createChannel();
6             Log.info("NodeCommunicationHandler", "Creating RPC Client");
7
8             nodeID = Integer.parseInt(callFunction("Registration"));
9         } catch (IOException | TimeoutException | InterruptedException e) {
10            Log.error("Node", e.toString());
11        }
12    }
```

Listing 3.2: NodeCommunicationModelHandler.initRPC()

3.1.2.2 initNodeToSink()

Capitolo 4

REST Server

Per integrare gli algoritmi di clustering realizzati in python con il core del progetto realizzato invece in Java è stato di scelto di far comunicare i due linguaggi mediante l'utilizzo di un serve REST in grado di fornire le funzioni realizzate in python tramite messaggi REST appositamente realizzati.

4.1 Implementazione

Dal punto di vista implementativo è stato scelto di realizzare il server REST mediante l'utilizzo della libreria Flask, in quanto offre un servizio completamente funzionante e modificabile seguendo le preferenze del programmatore.

4.1.1 Gestione delle richieste

Per poter chiamare i metodi messi a disposizione dal server REST vengono effettuate delle richieste REST nella quale si specifica, mediante un messaggio json, il metodo da richiamare ed gli argomenti necessari, rimanendo che il risultato venga processato e che un codice corrispondente allo stato d'esecuzione del metodo venga rispedito al mittente

```
1 from flask import Flask,request, jsonify
2 from flask_restful import Api,Resource,reqparse
3 from FCM import FCM
4 from Utils import removeOldFiles
5
6 class Server(Resource):
7     def post(self):
8         if(request.json['command'] == "Train"):
```

```
9               return FCM().train(request.json['ID'],request.json["Coeff"],
    request.json["Window"],request.json['values'])
10          elif (request.json['command'] == "Merge"):
11              return FCM().merge(int(request.json['nodes']))
12          elif (request.json["command"] == "Update"):
13              return FCM().update(int(request.json["ID"]))
14          else:
15              return "Command not available",200
16
17  removeOldFiles()
18  app = Flask(__name__)
19  api = Api(app)
20  api.add_resource(Server, '/server')
21  app.run(debug=True)
```

Capitolo 5

Testing