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

Analisi dei Dati

I dati prodotti sul nodo vengono analizzati localmente in modo da non dover inviare le informazioni raccolte attraverso la rete, in modo da ridurre il carico di informazioni sul nodo centrale e di preservare la privacy dell'utente, in quanto le unica informazione resa pubbblica sarà il modello generato, ma non i dati necessari a generare tale modello.

In particolare lo scopo delle nostre analisi è stato quello di effettuare il clustering di una serie di punti generati casualmente, e per ottenere questo risultato ci siamo affidati ad un Fizzy C-Means, in modo non solo da individuare per ciascun dato a quale cluster appartenesse, me anche il suo grado di appartenenza a tale cluster, al fine di effettuare future analisi ulteriormente più precise.

In aggiunta abbiamo inserito alcuni meccanismi atti a migliorare la creazione dei cluster, in particolare:

- E' stato realizzato un meccanismo di finestra scorrevole utile ad analizzare solo una porzione dei dati complessivi, in modo da dare un'importanza relativa (che può essere scelta dall'utente) ai valori storici rispetto a quelli appena ottenuti.
- E' stato anche inserito un controllo riguardo la posizione dei punti stessi rispetto ai precedenti centri dei cluster, in modo da accertatci prima di rigenerare un modello che i punti che si andranno ad analizzare siano validi e non delle outlier che sporcherebbero il modello finale.

5.1 Implementazione

Per realizzare queste analisi ci siamo avvalsi del linguaggio Python e di una libreria esterna che offriva un algoritmo di Fuzzy C-Means già completo e funzionante.

5.1.1 Training

Una volta generati i valori sul nodo verrà calcolato il modello come specificato nel seguente codice:

```
1     def train(self, id,coeff,window,values):
2         #Retriving the dataframe related to the generated file
3         df = pd.read_csv(BASE_DATA_PATH+id+".txt",names=["X","Y"],header=
    None,dtype={"X":float,"Y":float})
4         #Computing the effective dimension of the window
5         dim = int(int(window)*(1+float(coeff)))
6         START_WINDOW = dim * (-1);
7         if df.shape[0] == int(values):
8             result = True
9         else:
10            #Checking if the model must be computed
11            NEW_VALUES = int(values) * -1
12            newValues = df[NEW_VALUES:]
13            #Deleting from the original dataframe the new values and the
    previous window
14            df = df[:NEW_VALUES]
15            [df,result] = self.isModelNeeded(id,df,newValues)
16        if(result):
17            #Selecting only the desired window
18            if (START_WINDOW * (-1)) < df.shape[0]:
19                df = df[START_WINDOW:]
20            #Training the FCM with the array just obtained
21            points = np.array(df)
22            cntr,u_orig, _, _, _, _, _ = fuzz.cluster.cmeans(points.T,
    CLUSTERS,2,error=ERROR_THRESHOLD,maxiter = MAX_ITER)
23            #Creating the JSON with the information of the created model
24            model = {}
25            model["centers"] = cntr.tolist()
26
27            jsonToSave = {}
28            jsonToSave["points"] = points.tolist()
29            jsonToSave["centers"] = cntr.tolist()
30            save(0,int(id),"trainResult"+str(id)+"_"+str(time.time()),
    jsonToSave)
31
32
33            #Saving the JSON in the file
34            with open(BASE_MODEL_PATH+id+".json","w") as newModelFile:
35                newModelFile.write(json.dumps(model))
36
```

```
37              #Returning the OK code
38              return "Model created",201
39          else:
40              return "",204
```

5.1.2 Validazione dei punti

Come spiegato precedentemente però non sempre un modello deve essere ricalcolato, in quanto ci possono essere dei punti appena generati che sono outlier e che quindi possono sporcare il modello (nelle analisi abbiamo considerato come limite di outlier accettabile un quarto dei nuovi valori inseriti). Questo processo ovviamente non si applica alla generazione del primo modello, in quanto non avendo un base di partenza, tutti i punti vengono considerati buoni.

```
1     def isModelNeeded(self,id,df,df2):
2         if os.path.isfile(BASE_MODEL_PATH+id+".json"):
3             with open(BASE_MODEL_PATH+id+".json","r") as modelFile:
4                 #Load the centers from the model saved in the file
5                 centers = np.array(json.load(modelFile)["centers"])
6                 #Compute the distance between the new point and each center
     and find
7                 #the minimum distance for each new value
8                 minDistances = np.amin(cdist(np.array(df2.values),centers,
    metric='euclidean'),axis=1)
9                 #Finding the correct points and the outliers
10                correct = (minDistances <= DISTANCE_THRESHOLD)
11                outliers = np.invert(correct)
12                print(df2.loc[outliers].shape[0])
13                print(df2.shape[0])
14                #Creating a dataframe from that tuples
15                df = pd.concat([df,df2.loc[correct]])
16                #Writing on file the new dataframe
17                df.to_csv(BASE_DATA_PATH+id+".txt",index = None,header =
    None)
18                #checking if the number of outliers is above the threshold
19                if df2.loc[outliers].shape[0] <= int(df2.shape[0] * 0.5):
20                    return df,True
21                else:
22                    return df,False
23        else:
24            return df,True
```

5.1.3 Merging

Il sink invece ha il compito di unire tutti i modelli ricevuti in un unico generico, che sia in grado di migliorare (qualora fosse possibile) la precisione dei modelli generati singolarmenti dai nodi ai bordi della rete.

Per effettuare questa unione è stato di deciso di riapplicare un Fuzzy C-Means avendo come dati in input i centri stessi ricevuti, al fine di ottenere dei nuovi centroidi che siano migliori, sfruttando le informazioni ricevute. In aggiunta per garantire un'accuretezza migliore nei confronti dei nodi, dopo aver generato questo nuovo modello, si andrà a calcolare quando differisce da ciascun modello di partenza, in quanto ci possono essere dei modelli che sono totalmente differenti dalla maggior parte, in quanto i valori possono essere stati generati in diverse condizioni. Per garantire anche a questi nodi di aver il miglior modello possibile viene inviato, insieme al modello unito, anche un peso per ciascun nodo - che varia tra 0 e 1 - indicante la rilevanza del modello unito rispetto a quello di partenza.

```
1     def merge(self,nodes):
2         #Obtaining the centers
3         with open(BASE_MODEL_SINK_PATH+"1.json","r") as model:
4             centers = np.array(json.load(model)["centers"],dtype=float)
5         for i in range(2,nodes+1):
6             #Opening the file and concatenating the centers
7             with open(BASE_MODEL_SINK_PATH+str(i)+".json","r") as model:
8                 nodeCntrs = np.array(json.load(model)["centers"],dtype=
    float)
9                 centers = np.vstack((centers,nodeCntrs))
10
11        cntr,u_orig, _, _, _, _, _ = fuzz.cluster.cmeans(centers.T,CLUSTERS
    ,2,error=ERROR_THRESHOLD,maxiter = MAX_ITER)
12        mergedModel = {}
13        mergedModel["centers"] = cntr.tolist()
14        #Computing the mean Minumum distance for the new centers from the
    old centers
15        for i in range(1,nodes+1):
16            with open(BASE_MODEL_SINK_PATH+str(i)+".json","r") as model:
17                oldcntrs = np.array(json.load(model)["centers"])
18
19            indexes = np.argmin(cdist(cntr,oldcntrs,metric='euclidean'),
    axis=1)
20            #Calculating the minimum value along the row
21            minDistances = np.amin(cdist(cntr,oldcntrs,metric='euclidean'),
    axis=1)
```

```
22              minDistancesIndex =  np.argmin(cdist(cntr,oldcntrs,metric='
    euclidean'),axis=1)
23             #Check if there are at least a repetition
24              if(np.unique(minDistancesIndex).shape[0] == minDistancesIndex.
    shape[0]):
25                  #Compute the mean of the distances
26                  meanDistance = np.mean(minDistances)
27                  mergedModel[str(i)] = self.associate(meanDistance)
28              else:
29                  mergedModel[str(i)] = 0
30
31          jsonToSave = {}
32          jsonToSave["newcenters"] = cntr.tolist()
33          jsonToSave["oldcenters"] = centers.tolist()
34          save(1,0,"MergedModel_"+str(time.time()),jsonToSave)
35
36
37          #Saving the new Model
38          with open(MERGED_MODEL_PATH,"w") as mergedModelFile:
39              json.dump(mergedModel,mergedModelFile)
40          return "Models merged",201
```

5.1.4 Updating

Infine, una volta che ciascun nodo ha ricevuto il modello unito dal sink, avrà il compito di aggiornare il proprio modello in base alle informazioni appena ricevute, in particolare la discrimante è il peso che il sink stesso ha associato al suo modello che indica:

- 0: Il nodo deve usare il proprio modello invece che quello unito, in quanto è troppo differente
- 0.5: Il nodo deve usare un modello che è la media aritmetica tra il proprio e quello ricevuto
- 0.75: Il nodo deve usare un media pesata tra i modelli - il suo modello ha peso 1 mentre quello ricevuto ha peso 3 - .
- 1 : Il nodo deve usare solo il modello ricevuto dal sink, e scartare il proprio.

```
1     def update(self,id):
2         with open(BASE_MODEL_PATH+str(id)+".json","r") as oldModelFile:
```

```
3             oldModel = np.array(json.load(oldModelFile)['centers'])
4         with open(MERGED_MODE_NODE_PATH,"r") as mergedModelFile:
5             dict = json.load(mergedModelFile)
6             mergedModel = np.array(dict['centers'])
7             weight = float(dict[str(id)])
8         distances = cdist(mergedModel,oldModel,metric="euclidean")
9         minDistancesIndex = np.argmin(distances,axis=1)
10        updatedPoint = []
11        for i in range (0,mergedModel.shape[0]):
12            IncrementX = float(mergedModel[i,0])*weight + float(oldModel[
    minDistancesIndex[i],0])*(1-weight)
13            IncrementY = float(mergedModel[i,1])*weight + float(oldModel[
    minDistancesIndex[i],1])*(1-weight)
14            updatedPoint.append([IncrementX,IncrementY])
15
16        jsonToSave = {}
17        jsonToSave["oldModel"] = oldModel.tolist()
18        jsonToSave["mergedModel"] = mergedModel.tolist()
19        jsonToSave["updatedPoint"] = updatedPoint
20        save(0,int(id),"updatedPoint"+str(id)+"_"+str(time.time()),
    jsonToSave)
21
22        dict = {}
23        dict["centers"] = updatedPoint
24        with open(BASE_MODEL_PATH+str(id)+".json","w") as updatedModelFile:
25            json.dump(dict,updatedModelFile)
26        return "Model updated",200
```

Capitolo 6

Testing