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

Gestione dei Dati

2.1 Concorrenza in Scrittura

I nodi ricevono dati da una o più sorgenti, andandoli a salvare all'interno della stessa *Repository*. Questo aspetto ha come problema l'accesso concorrente in scrittura.
Per risolvere questo problema è stato deciso di utilizzare un meccanismo di locking, in particolare è stato usato il Fair Lock, dato che permette di concedere il lock con una politica di tipo FIFO ai thread che devono inserire il valore generato dal sensore, in questo modo, utilizziamo i dati con lo stesso ordine in cui arrivano e inoltre evitiamo la *Starvation* dei Thread.

2.2 Implementazione

Per avere una migliore gestione della concorrenza dei Thread è stato deciso di utilizzare come linguaggio implementativo Java

2.2.1 Data Collector

Modulo usato per avviare i Thread che simulano la generazione dei dati dei sensori e avvia anche il modulo che si occupa del protocollo di comunicazione con il Sink. Si occupa anche di creare un'istanza di **RepositoryHandler**, la quale si occupa della gestione della Repository.

```
1 package it.unipi.cds.federatedLearning.node;
2
3 import it.unipi.cds.federatedLearning.Config;
4 import it.unipi.cds.federatedLearning.Log;
```

```
5
6  import java.io.IOException;
7  import java.util.ArrayList;
8
9  /**
10  * This class is used to start a simulated connection with simulated sensor
        nodes
11  * in order to collect data and to start the process that communicates with
        the sink
12  * using RabbitMQ.
13  * Here are stored some constants used to decide how many values will be
       stored, how many thread
14  * (simulating the sensors) will start and the various threshold used to
       decide if the machine
15  * learning algorithm, that generates the neural network, will be started
16  *
17  */
18 public class DataCollector {
19
20   /*
21    * Read Data variables
22    */
23   public final static int THRESHOLD = Config.SIZE_WINDOW;
24   /*
25    * Testing variables
26    */
27   public final static int numberOfThreads = 100;
28   public final static int numberOfWrites = 1000;
29
30   public static boolean aModelIsBeingGeneratedNow = false;
31   public static NodeCommunicationModelHandler nodeCommunicationHandler;
32
33   public static void main(String[] args) throws InterruptedException {
34     try {
35       nodeCommunicationHandler = new NodeCommunicationModelHandler(args[0])
       ;
36     } catch (ArrayIndexOutOfBoundsException e) {
37       Log.error("Sink", "Provide RabbitMQ Server's ip address as argument")
       ;
38     }
39     ArrayList<Thread> threads = new ArrayList<>();
40     RepositoryHandler repository = new RepositoryHandler(THRESHOLD);
41     for(int i = 0; i < numberOfThreads; i++) {
42       /*
```

```
43          * the last parameter is used for to specify if there is an infinite
    number of writes or not
44          */
45          Runnable r = new DataGenerator(repository, numberOfWrites, false);
46          Thread t = new Thread(r);
47          t.start();
48          threads.add(t);
49        }
50        /*
51         * USED ONLY IF THE NUMBER OF WRITES IS NOT INFINITE
52         */
53        for ( Thread t : threads )
54          t.join();
55
56        try {
57          DataCollector.nodeCommunicationHandler.callFunction("Leave");
58        } catch (IOException | InterruptedException e) {
59          e.printStackTrace();
60        }
61        System.exit(0);
62      }
63
64  }
```

2.2.2 Data Generator

Componente con il compito di generare i dati e di scriverli all'interno della Repository, simulando il comportamento dei dati generati dai sensori. Per fare questo vengono simulate sia la frequenza di arrivo dei vari valori, ossia ogni quanto viene rilevato e ricevuto da un sensore un valore, che il valore che può assumere la risorsa rilevata dal sensore.
Nel nostro caso abbiamo simulato:

- la frequenza di arrivo dei valori come un esponenziale con una frequenza di interarrivo pari a **lambda**, per simulare questa *attesa* si utilizza la **Thread.sleep(delay)**, con delay=valore successivo della sequenza esponenziale.

- il valore delle risorse, rappresentato da una coppia di valori e per generare valori separati è stato deciso di usare due Gaussiane con valor medio una di +**Config.MEAN**, l'altra −**Config.MEAN** ed entrambe con una deviazione standard pari a **Config.ST_DEV**.

```java
package it.unipi.cds.federatedLearning.node;

import java.util.Random;
import java.util.concurrent.TimeUnit;

import it.unipi.cds.federatedLearning.Config;

/**
 * This class is used to simulate the data generated by the sensors: to simulate the data generated
 * we have used two IID Gaussian distribution (seeded with two different seeds); to simulate the
 * interarrivals of the data we use an Exponential distribution IID for each sensor with a constant mean rate
 *
 */
public class DataGenerator implements Runnable{

  private RepositoryHandler repository = null;
  /*
   * Used to have an exponential distribution of the interarrivals of the data collected
   * exponentialSeed is used to generate a uniform distribution between 0 and 1
   * lambda is used as the mean rate of arrival of a message
   */
  private Random exponentialSeed;
  private double lambda = 0.9;

  /*
   * Used to have a fixed number of writes
   */
  private int numberOfWrites;
  /*
   * Used to have infiniteWrites
   */
  private Boolean infiniteWrites;

  /**
   * Constructor initializes the basic attributes and generate and create the exponentialSeed with a unique seed each time that is called
   * @param repository
   * @param numberOfWrites
```

```
38      * @param infiniteWrites
39      */
40     public DataGenerator(RepositoryHandler repository, int numberOfWrites,
        Boolean infiniteWrites) {
41       this.repository = repository;
42       this.numberOfWrites = numberOfWrites;
43       this.infiniteWrites = infiniteWrites;
44       this.exponentialSeed  = new Random();
45     }
46
47     /**
48      * Here we generate the Gaussian distributions and in a loop after the
        exponential delay it simulates the sending of the data
49      */
50     public void run(){
51       Random gaussianX = new Random();
52       Random gaussianY = new Random();
53
54       for (int i = 0; i < numberOfWrites || infiniteWrites; i++) {
55         double delay = getNextExponentialDelay() + 1.0;
56
57         try {
58           TimeUnit.SECONDS.sleep((long) delay);
59         } catch (InterruptedException ex) {
60           System.err.println(Thread.currentThread().getName() + " has been
       interrupted!");
61           Thread.currentThread().interrupt();
62         }
63         Double sensedDataX = gaussianX.nextGaussian();
64         Double sensedDataY = gaussianY.nextGaussian();
65         sensedDataX *= Config.ST_DEV;
66         sensedDataY *= Config.ST_DEV;
67         if(Thread.currentThread().getId()%2 == 0) {
68           sensedDataX += Config.MEAN;
69           sensedDataY += Config.MEAN;
70         }else {
71           sensedDataX -= Config.MEAN;
72           sensedDataY -= Config.MEAN;
73         }
74
75
76         try {
77           repository.write(sensedDataX, sensedDataY);
78         } catch (InterruptedException e) {
```

```
79            System.err.println(e.getMessage());
80          }
81        }
82      }
83
84      /**
85       * Used to generate a real exponential distribution starting from a
          Uniform distribution (exponentialSeed) with a mean value of lambda
86       * @return the simulated time necessary to the sensor to sense the data
87       */
88      public double getNextExponentialDelay() {
89          double result =  Math.log(1-exponentialSeed.nextDouble())/(-lambda);
90          if(result < 0)
91            result = -result;
92          return result;
93      }
94
95
96
97  }
```

2.2.3 Repository Handler

Modulo delegato a gestire l'accesso concorrente in scrittura alla repository, implementa il Fair Lock, e decide anche se chiamare il modulo **ModelCaller**, per richiedere la generazione di un nuovo modello. Per prendere questa decisione sono usate delle soglie **threshold** all'inizio e **newValues** successivamente.

```
1  package it.unipi.cds.federatedLearning.node;
2
3  import it.unipi.cds.federatedLearning.Config;
4
5  import java.io.*;
6  import java.nio.file.*;
7  import java.util.*;
8  import java.util.concurrent.atomic.*;
9
10 /**
11  * This class is used as a receiver of the data generated by the simulated
      sensors, it handles
12  * the concurrency of the writes with a Fair Lock and start the ModelCaller
      to perform the REST request when it is necessary
13  *
```

```
14  */
15 public class RepositoryHandler {
16
17   /*
18    * The first time the Machine Learning algorithm will be called only if
       there are a number of values higher than a certain threshold
19    */
20   private int threshold;
21   /*
22    * Used to know how many values were stored when the ML was called the
       last time
23    */
24   private int oldNumberOfSamples = 0;
25   /*
26    * When we have a number of data higher than the threshold, ML called
       when there are a certain number of new values
27    */
28   private AtomicInteger newValues = new AtomicInteger(0);
29   /*
30    * Used to store the actual number of values
31    */
32   private static AtomicInteger numberOfSamples = new AtomicInteger(0);
33
34   /*
35    * Constants used to store the files path
36    */
37   private final String samplePath = Config.PATH_NODE_COLLECTED_DATA+
       DataCollector.nodeCommunicationHandler.getNodeID()+ ".txt";
38   private final String readySamplesPath = Config.PATH_NODE_READY_DATA+
       DataCollector.nodeCommunicationHandler.getNodeID()+ ".txt";
39
40
41   /*
42    * Variables used to manage concurrency with a FairLock implementation
43    */
44   private boolean isLocked = false;
45   private Thread lockingThread = null;
46   private List<String> waitingThreads = new ArrayList<>();
47
48   /**
49    * Constructor of the class, it sets its attributes, and check if the
       directory used to store the data exist, if not it is created, with the
       necessary files
50    * @param threshold integer used as first threshold to decide when the
```

```
     machine learning must be started
51   * @param newValues integer used after threshold, it represents the
     number of new values, from the
52   * moment the machine learning has being called, necessary to start again
      the machine learning
53   */
54  public RepositoryHandler(int threshold) {
55    this.threshold = threshold;
56    this.oldNumberOfSamples = 0;
57    try {
58        File folder = new File(Config.PATH_NODE_BASEDIR);
59        if(!folder.exists())
60          folder.mkdir();
61        File cd = new File(samplePath);
62        File rd = new File(readySamplesPath);
63        if(!cd.exists())
64          cd.createNewFile();
65        if(!rd.exists())
66          rd.createNewFile();
67    }catch(IOException e) {
68      System.out.println(e.getMessage());
69      return;
70    }
71  }
72
73  /**
74   * Lock used for the Fair Lock, if the lock is free or the thread is at
     the beginning of the queue
75   * used for the implementation of the Fair Lock, then the current thread
     can get the lock, otherwise it
76   * must wait its turn
77   * @throws InterruptedException because it is used the function Thread.
     wait()
78   */
79  public void lock() throws InterruptedException{
80
81    String activeThread = Thread.currentThread().getName();
82
83    synchronized(this) {
84      waitingThreads.add(activeThread);
85      while(isLocked || waitingThreads.get(0) != activeThread) {
86        synchronized(activeThread) {
87          try {
88            activeThread.wait();
```

```
89            }catch(InterruptedException e){
90              waitingThreads.remove(activeThread);
91              throw e;
92            }
93          }
94        }
95        waitingThreads.remove(activeThread);
96        isLocked = true;
97        lockingThread = Thread.currentThread();
98      }
99    }
100
101   /**
102    * Unlock used for the Fair Lock, here the current thread check if it has
        the lock and if it
103    * has the lock, the current thread proceeds to release it, otherwise it
       throws an IllegalMonitorStateException
104    * @throws IllegalMonitorStateException when a lock that has not get the
       lock tries to release it
105    */
106   public void unlock() {
107     if(!isLocked || this.lockingThread != Thread.currentThread()) {
108       throw new IllegalMonitorStateException("Calling thread has not locked
       this lock");
109     }
110     this.isLocked = false;
111     lockingThread = null;
112     if(waitingThreads.size() > 0) {
113       String sleepingThread = waitingThreads.get(0);
114       synchronized (sleepingThread) {
115         sleepingThread.notify();
116       }
117     }
118   }
119
120   /**
121    * Function used to simulate the reception of the data from a sensor node
       . Here we use
122    * the Fair Lock to handle the concurrency of the write operations in a
       single file.
123    * If the number of values written is above the threshold then we call
       the read.
124    * @param sensedDataX first value sensed of the pair of value sensed
125    * @param sensedDataY second value sensed of the pair of value sensed
```

```
126     * @throws InterruptedException because of the lock function
127     */
128    public void write(Double sensedDataX, Double sensedDataY) throws
        InterruptedException{
129
130      this.lock();
131
132      int instantNumberOfSamples = numberOfSamples.incrementAndGet();
133
134      try(
135        FileWriter fw = new FileWriter(samplePath, true);
136          )
137      {
138        fw.append(sensedDataX.toString() + "," + sensedDataY.toString() + "\n
       ");
139      }catch(IOException e) {
140        System.out.println(e.getMessage());
141        return;
142      }
143
144      if(!DataCollector.aModelIsBeingGeneratedNow) {
145        int value = newValues.get();
146        if((oldNumberOfSamples == 0 && instantNumberOfSamples >= threshold)
147            || (oldNumberOfSamples > 0 && instantNumberOfSamples -
       oldNumberOfSamples >= value)
148        ){
149          DataCollector.aModelIsBeingGeneratedNow = true;
150          if(oldNumberOfSamples == 0) {
151            this.read(threshold);
152          }
153          else {
154            this.read(value);
155          }
156          oldNumberOfSamples = instantNumberOfSamples;
157        }
158      }
159      this.unlock();
160    }
161
162    /**
163     * Function used to read and move the value stored from the sensors to
        another file that will be used
164     * by the machine learning algorithm, this is made to avoid the fact that
        new values can arrive while the
```

```
165     * machine learning algorithm is working, and for all the new value we
        must check before starting the algorithm
166     * if it is an outlier or not. After moving the data it starts
        ModelCaller
167     * @param valuesToRead used to specify the actual number of values inside
         the file used to store the data from the sensors
168     *
169     */
170    private void read(int valuesToRead){
171      try(FileWriter fw = new FileWriter(readySamplesPath, true);
172          ){
173
174        String readyData = new String (Files.readAllBytes(Paths.get(
       samplePath)));
175
176        if(oldNumberOfSamples == 0)
177          fw.write(readyData);
178        else
179          fw.append(readyData);
180        /*
181         * Call the function to send the synchronous REST request
182         */
183        Runnable caller = new ModelCaller(valuesToRead);
184        new Thread(caller).start();
185        newValues.set((int) (numberOfSamples.get()*Config.
       PERCENTAGE_OLD_VALUES));
186        //We flush the file with the sample not ready to reduce redundancy of
        the data
187        try(FileWriter fw2 = new FileWriter(samplePath);){
188          fw2.write("");
189        }catch(IOException ex) {
190          System.out.println(ex.getMessage());
191        }
192
193      }catch(IOException e) {
194        System.out.println(e.getMessage());
195      }
196
197    }
198 }
```

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                #Creating a dataframe from that tuples
13                df = pd.concat([df,df2.loc[correct]])
14                #Writing on file the new dataframe
15                df.to_csv(BASE_DATA_PATH+id+".txt",index = None,header =
      None)
16                #checking if the number of outliers is above the threshold
17                if df2.loc[outliers].shape[0] <= int(df2.shape[0] * 0.5):
18                    return df,True
19                else:
20                    return df,False
21        else:
22            return df,True
23    def update(self,id):
24        with open(BASE_MODEL_PATH+str(id)+".json","r") as oldModelFile:
```

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               oldModel = np.array(json.load(oldModelFile)['centers'])
2           with open(MERGED_MODE_NODE_PATH,"r") as mergedModelFile:
```

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

Capitolo 6

Testing