SAPIENZA
UNIVERSITÀ DI ROMA

FACOLTÀ D'INGEGNERIA DELL'INFORMAZIONE INFORMATICA E STATISTICA

Distributed Harmony: Cloud-Powered Playlist Federated Learning with Docker

CLOUD COMPUTING FINAL PROJECT

Professors:
Emiliano Casalicchio

Students:
Gheorghiu Claudiu
1845527
Mandara Angelo
2077139
Tamburini Tito
1837335

Academic Year 2022/2023

Contents

1 Problem Description

The central challenge addressed in this study is the **classification of songs** into distinct historical eras, a task essential for applications like music recommendation, historical analysis, and content curation.
However, traditional machine learning approaches encounter obstacles when dealing with this classification, especially concerning **data privacy** and **security issues**.
The sensitivity of music data and concerns about data sharing necessitate an innovative solution. **Federated learning**, a decentralized machine learning paradigm, provides an answer to these challenges.
It enables multiple parties or clients to collaboratively build a classification model while retaining data privacy and adhering to regulatory constraints. This approach is particularly beneficial when sensitive song data needs to be handled securely.
The study leverages a **master-slave architecture** to coordinate the federated learning process. The *master* node orchestrates the entire process, initializing the **logistic regression model's weights** and aggregating updates from *slave* nodes.
These slave nodes perform computations independently on their own subsets of data without sharing the raw data. This architecture ensures that data privacy and security are maintained throughout the model-building process.

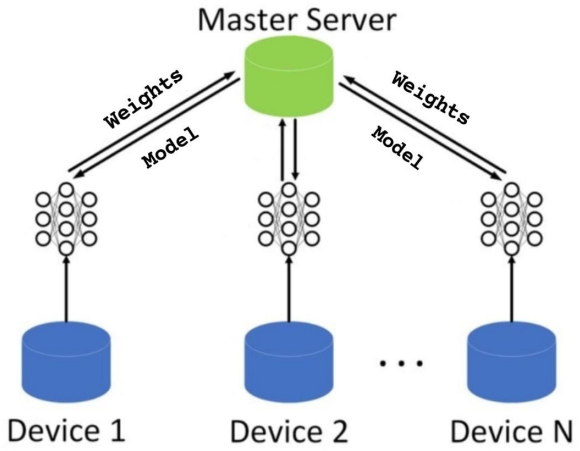

Figure 1: Federated Learning Process. The master initializes, refines, and redistributes model weights to slaves for predictive accuracy assessment, ensuring data privacy throughout.

The primary objective is to create a robust model capable of accurately assigning songs to their respective historical periods. By employing federated learning within a master-slave architecture, this study not only addresses the era-based song classification problem effectively but also **showcases the potential of federated learning in preserving data privacy while achieving critical classification tasks in sensitive data scenarios**.

In addressing the problem of era-based song classification, the utilization of **Docker** for weight distribution and the master-slave architecture is of paramount importance. Docker is an invaluable tool as it not only efficiently transfers model weights to client nodes but also **encapsulates the process**, **providing isolation and sandboxing that further enhances data privacy**. This isolation helps prevent unintended data leaks and unauthorized access.

Additionally, deploying the system in a cloud environment like **AWS** can provide **scalability and accessibility**, enabling efficient collaboration among client nodes while adhering to stringent data security requirements. This combination of methodologies and technologies effectively addresses the core problem of classifying songs into historical eras, safeguarding data privacy, and ensuring the efficient construction of a robust model.

2 Solution Design

To address the challenge of era-based song classification, we propose implementing a **federated learning system**.
A master-slave architecture will be used to coordinate the federated learning process.

In this system, the *master* node orchestrates the entire process.
The *master* node initiates the federated learning process by initializing the **logistic** regression model and ensuring that every *slave* node begins the training phase with the same initial model.
So, the *master* node sends the initial weights of a logistic regression model to every *slave* node in the federated system, and the *master* also issues a **unique ID** to each *slave*.
Following the receipt of the initialized model, each *slave* node accesses a partition of the dataset specified by the ID provided by the *master* node and executes logistic regression training on the partition acquired.
When the **training phase** is complete, each *slave* node returns the **weights** of the trained logistic regression model to the *master* node.
The *master* node waits for each *slave* node in the federated architecture to deliver its weights before **aggregating** them to obtain the most recent version of the logistic regression model.
The *master* node sends the aggregated weights back to the *slave* nodes, and each *slave* node *tests* the new model on its division of the dataset.
At the completion of the testing period, each *slave* node computes a *metric* and delivers it back to the *master*. When all of the *slave* nodes have computed and submitted the

metric, the *master* node computes the average metric.

When all of these processes are completed, the federated learning program exits by printing the final measure of the model derived by aggregating the weights of each *slave* node on the screen.

Finally, with the obtained optimized model, a song can be classified by the master node which will assign it a decade of appartenance.

3 Solutions Implementation

The proposed federated learning system with a master-slave architecture, which uses **Docker** containers for the master and slave nodes, is a feasible solution to the problem of era-based song classification. This architecture makes use of distributed computing to ensure that each slave node starts with the same initial model and contributes to model improvement collectively.
The following is a breakdown of the solution implementation:

Docker Containers: Docker is used to create isolated environments for both the master and slave nodes, ensuring reproducibility and scalability.

Communication:
- To enable communication between master and slave nodes, a well-defined **RESTful API** is established using **Flask** for the master node.
- The slave nodes send **HTTP** requests to the master node's API to retrieve instructions and submit results.

Master Node Implementation:
- The master node is implemented as a **RESTful** web application in **Python**, using the **Flask** module.
- This web application serves as the central coordination point for the federated learning process.
The *master* node offers 4 services:
- **"/get_model"**, method = 'GET': The *slave* node uses this service to request either the initialized Logistic Regression Model or the updated Logistic Regression Model, the latter if it was built in the *master* node. The *master* node also sends a unique ID to the *slave* node when sends the initialized model.
- **"/get_model"**, method = 'POST': The *slave* node uses this service to send to the *master* node the coefficients it has obtained after training the Logistic Regression

model on its partition of the dataset. The *master* node, each time the coefficients are sent correctly, stores the coefficients and waits until all the *slave* nodes have completed the training phase, to aggregate the coefficients together.
- **"check_server"**, method = 'GET': The *slave* node makes a request to *master* node to check if it can request the new updated model. The *master* node will give a request status_code 200 only if all the *slave* nodes that participated to the federated learning process have completed the training phase and sent correctly to the *master* node the coefficients of their Logistic Regression model.
This service is pretty useful to synchronize all the *slave* nodes, because it may happen that some *slave* nodes finish the training phase earlier than others.
- **"test_model"**, method = 'POST': The *slave* node uses this service to send to the *master* node the metric obtained after testing the updated Logistic Regression model on its partition. The *master* node stores the metric received and waits until all the *slave* nodes have sent theirs, to average all the metrics obtained on each partition.

Some details:
- The *master* node implements a locking mechanism using the **Python** object **Mutex**, to make sure that even when there are multiple concurrent requests, the data is stored correctly without collision.
- In the *master* node the unique ID are all stored in queue which empties each time a *slave* node requests the initialized Logistic Regression model, basically each time a slave node enters the Federated Learning process.
- In the *master* node the weights are stored in Numpy arrays, to facilitate the aggregation.
- The *master* node counts how many slaves have sent their coefficients and metrics by using 2 counter global variables, it also store in a global variable how many slaves participate the Federated Learning Process.

Slave Node Implementation:
- The slave node is implemented as a **Python** client script.
- Each slave node communicates with the master node by making requests to the services provided by the master.
- These requests include actions like model initialization, data partition retrieval, model training, and metric computation.
Some details:
- The *slave* node uses **Pandas Python** module to load and read the partitions in format .csv.
- The *slave* node uses a Logistic Regression Model which we decided to build from scratch to avoid discrepancies in the format of the coefficients. This was done by adding in the *slave* docker a **Python** script with the definition of a custom **Python** class, with all the methods needed to run a successful Logistic Regression model.

4 Deployment of the Solution

We wanted to **deploy** our Federated Learning Application multiple times in different **AWS EC2 instances**, with different number of *slave* nodes and consequently different number of partitions of the dataset. To deploy our Federated Learning Application in **AWS**, we used the following bash script:

```
#!/bin/bash
export NUM_SLAVES=N  #where N is the scaling parameter
cd Master
docker build -t master_image .
python3 create_partitions.py
cd Slave
docker build -t slave_image .
cd ..
docker-compose up -d --scale slave=$NUM_SLAVES
```

The script sets the **NUM_SLAVES** environment variable to the number of slave that will participate to the Federated Learning process. Then it builds the *master* node Docker image and launches ***create_partitions.py*** which splits the dataset in many partitions as the number of *slave* nodes, next it builds the *slave* node Docker image. The partitions are directly loaded in the *slave* **Docker** image, each partition with a unique ID, so that each *slave* node can access only the partition the *master* node assigns to it. Finally it deploys the containers via **Docker Compose**:

```
version: '3'
services:
  master:
    image: master_image
    networks:
      - fed_network
    environment:
      - NUM_SLAVES=${NUM_SLAVES}
  slave:
    image: slave_image
    networks:
      - fed_network
    depends_on:
      - master
networks:
  fed_network:
    driver: bridge
```

We compressed all the file needed for the application execution in .zip file and loaded it in a **S3 bucket**.

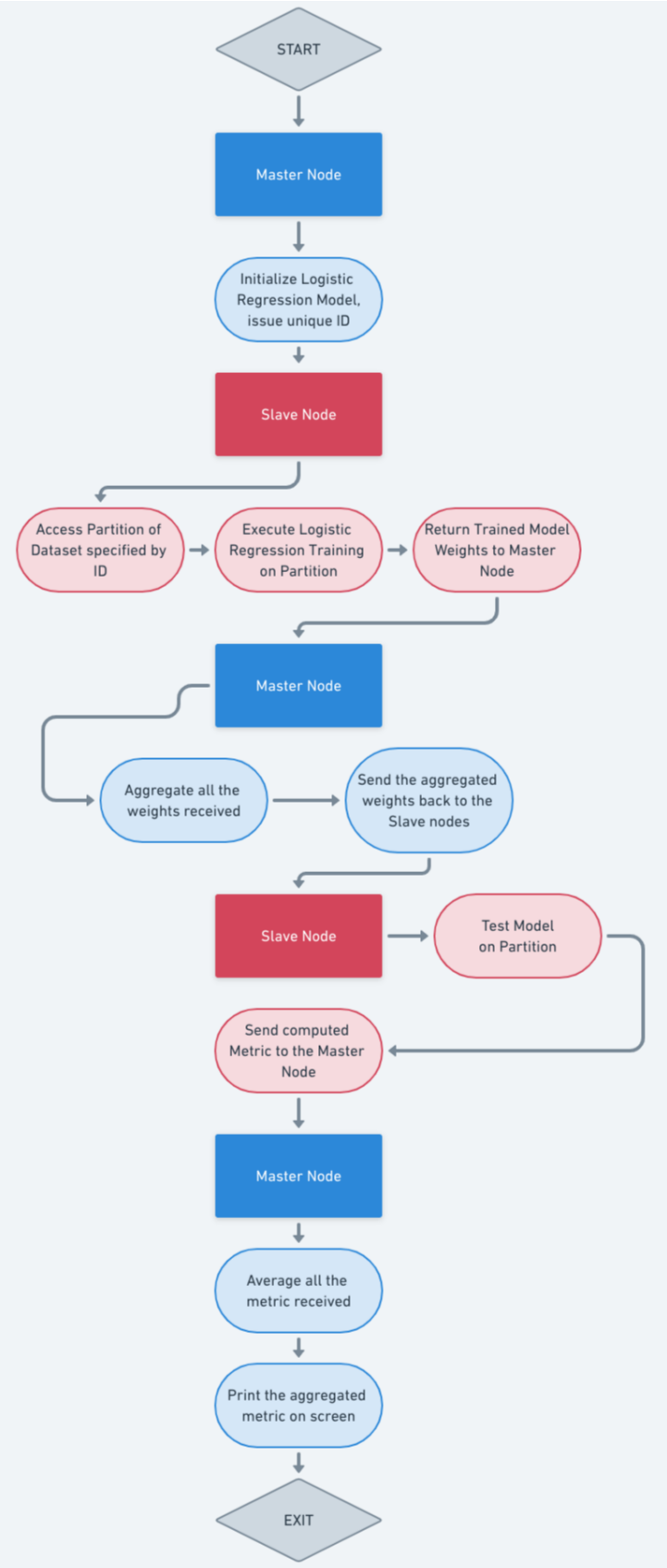

Figure 2: Federated Learning Process: workflow and interactions between the *master* node and a *slave* node participating to the Federated Learning Process.

We launched multiple the same **EC2 istance**, with these characteristics:
-Amazon Machine Image: *Amazon Linux*
-Instance Type: *t2.large*
-Security Group:
-Inbound Rule
-TCP ,Port: 2377 ,Origin: 0.0.0.0/0
-SSH ,Port: 22 ,Origin: 0.0.0.0/0
-HTTP ,Port: 80 ,Origin: 0.0.0.0/0
User Data:

```
!/bin/bash
sudo yum -y update
sudo yum -y install docker python pip
service docker start
usermod -a -G docker ec2-user
chkconfig docker on
```

Once the istances are running we execute the following *CLI's*:

```
pip install docker-compose pandas
wget 'https://clcfinalproject.s3.amazonaws.com/CLC_Final_Project.zip'
unzip CLC_Final_Project.zip
cd CLC_Final_Project
nano docker_bash.sh        #now we set the number of slaves
bash docker_bash.sh
```

After all these steps our application is running on an EC2 instance.

5 Experimental Design and Results

When all our istances are running, we used **AWS CloudWatch** to monitor the **CPU Credit Usage** and the **CPU Usage**.
Here we report the plots **AWS CloudWatch** provided to us, these measurement are taken during the entire execution of our Federated Learning Application, from the initialization of the **EC2 istances** to the termination of the Application.
Based on the **AWS CloudWatch CPU usage**, the **EC2 instance** with the most nodes (70, orange line) has only attained a maximum CPU consumption of 30%. This shows that your application is making good use of the computational resources available on the instance, and there is still plenty of room to scale it up if necessary.
Given that **CPU usage** increases proportionally with the number of nodes (10% for 10 nodes, green line, and 30% for 70 nodes), the program is expected to scale well as the number of nodes increases. This is an indication of good scalability.

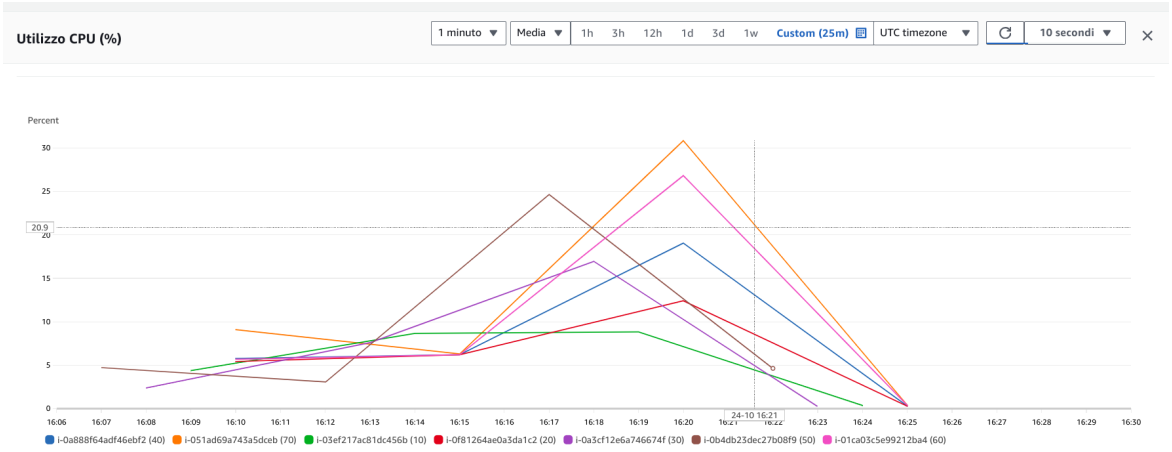

Figure 3: AWS CloudWatch CPU usage plot

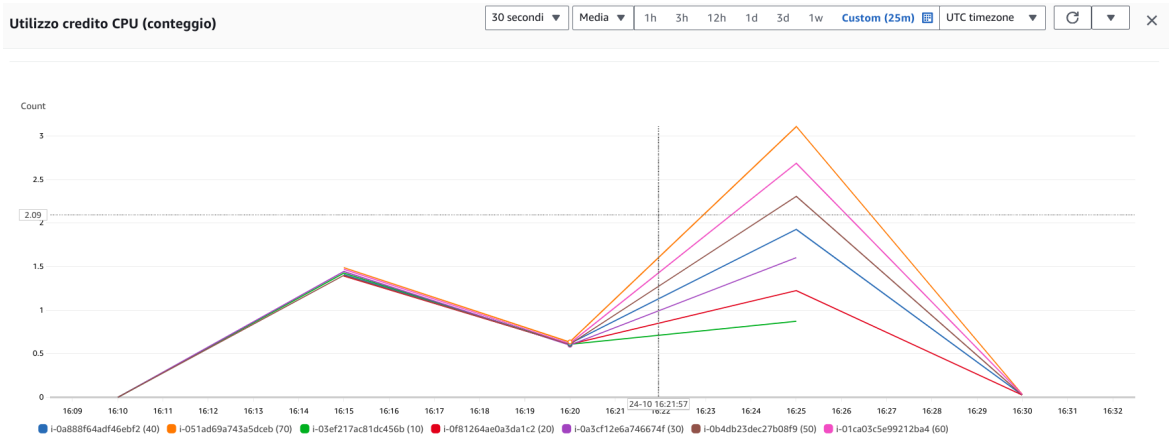

Figure 4: AWS CloudWatch Credit CPU usage plot

As expected, the **AWS CloudWatch Credit CPU Usage Count** is coherent with the **CPU usage** of each instance; more credit is used as more slave nodes work in the Federated Learning process.

Future developments

In order to expand the design of our application in the future, we planned to develop a secondary web service in the *master* node that allows any user to utilise our Federated Learning model.

When a user submits the name of a **Spotify** song, our model predicts the song's decade of belonging. After the Federated Learning process, our model should provide a forecast with an accuracy of **66%**, based on our training.

References

[1] Buyya Rajkumar/Vecchiola, Christian/Selvi. *Mastering Cloud Computing: Foundations and Applications Programming.* 2013.

[2] Alex Rodriguez. *RESTful Web services: The basics (2008/2015) IBM developerWorks series.*

[3] Morgan Kaufmann Dan Marinescu. *Cloud Computing: Theory and Practice.* 2013.