IoT Report

Andrea Policarpi - 0000950326
andrea.policarpi@studio.unibo.it

Nicola Amoriello - 0000952269
nicola.amoriello@studio.unibo.it

Contents

Chapter 1

Introduction

The Energy-efficient Weather Station project consists in the development of a proof-of-concept IoT weather station, with a focus on energy-aware data sampling techniques based on On-Line Reinforcement Learning (RL) algorithms.

The proposed weather station is capable of monitoring environmental parameters (temperature and humidity), collecting and handling IoT data, and forecasting their future values.

The system is composed of:

- End devices: DHT22 temp/hum sensors connected to an ESP32;
- MQTT Broker (Mosquitto), acting as a data bridge between ESP32 and database;
- Data management system, represented by Influx and Grafana tools;
- Data analytics, in charge of forecasting temperature and humidity by a LSTM, based on previous observations;
- Reinforcement learning application, which aims to self-adjust the sampling rate of the sensors on the base of the measurements quality.

The system is completed by a Web Dashboard, which allows the system administration.

Chapter 2

Project's Architecture

The IoT Weather Station system is composed by six macro-group of components:

- **IoT smart device,** composed by a DHT22 temperature connected to an ESP32;
- **Data Bridge,** represented by a Mosquitto MQTT broker acting as a bridge between the device and the data management system;
- **Data Management System,** represented by INFLUX and GRAFANA tools, in charge of data storage and visualization;
- **Cloud-level RL algorithm,** represented by a Python script that applies Q-learning to the self-tuning of the IoT device's sensing frequency. This block also comprehends a **performance evaluation** module, in charge of evaluating the effectiveness of the sampling interval adaptation scheme;
- **Data Analytics,** represented by a Python script in charge of temperature and humidity forecasting;
- **Web Dashboard,** which acts as an user interface for the system administration.

The overall system is represented in figure 2.1.

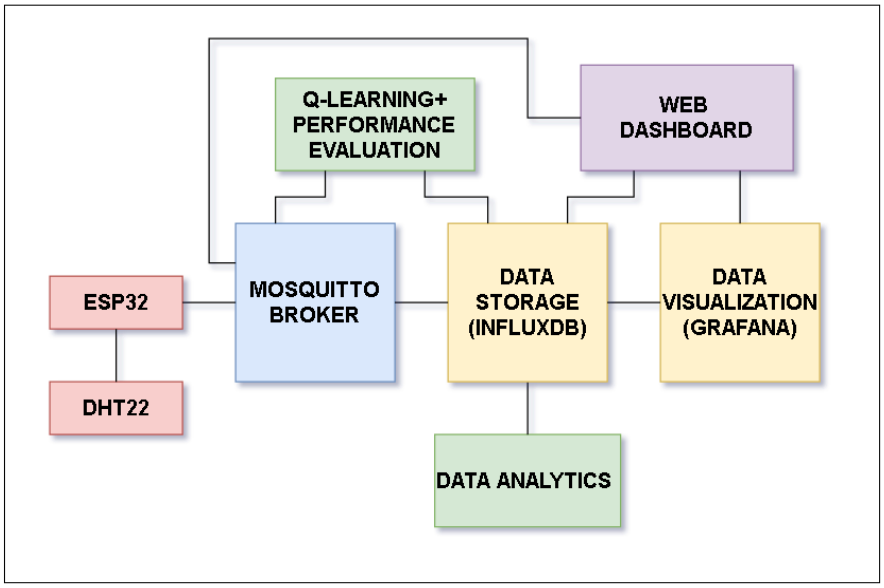

Figure 2.1: Overview of the IoT Weather Station system.

2.1 IoT smart device

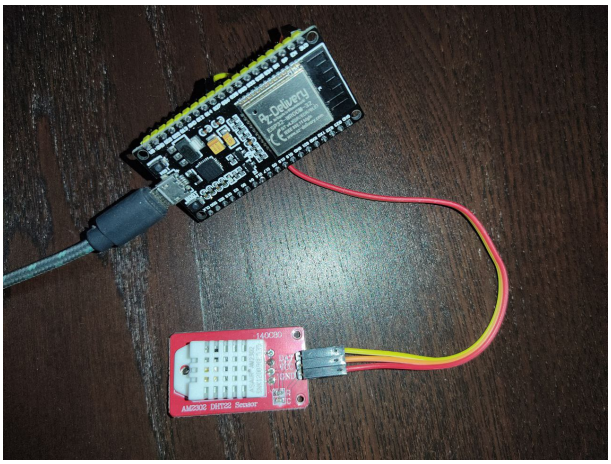

Figure 2.2: DHT22 and ESP32, composing the IoT device.

The device is composed of a DHT22 temperature/humidity sensor connected to an ESP32 microcontroller. An ID and GPS coordinates are associated to the device.

The DHT22's measurements are sent to the ESP32 with a variable sampling interval.

The ESP32:

1. Acquires the sensor values every SAMPLE_FREQ seconds, where the frequency is tuned by a Q-learning algorithm with respect to the signal quality. If the measurements fall outside certain safety ranges ([MIN TEMP: MAX TEMP] and [MIN HUM: MAX HUM]), the built-in LED of the ESP32 is also turned on as a visual alert. The safety boundaries are initialized to a fixed value, but can also be changed by the user at runtime by both the Web Dashboard or a direct message to the Mosquitto broker;

2. Transmits and receives data over a Wi-Fi connection, by using the MQTT protocol, to and from the Mosquitto broker. The device transmits the following data:

   - Temperature [°C] and Humidity [%RH] measurements;
   - WiFI signal strength [dBm] of the connection to the router;
   - ID and GPS coordinates of the device.

   And receives the following data:

   - Changes on temperature and humidity boundaries;
   - Changes on sampling frequency.

2.2 Data Bridge

The data bridge, which is represented by a Mosquitto MQTT broker, is in charge of:

- Transmitting the sensor measurements from the IoT device to the INFLUX database;
- Transmitting changes in the sampling frequency from the Q-learning Python script to the IoT device;
- Transmitting changes in the measurement boundaries from both the Web Dashboard or a direct message to the broker to the IoT device.

The choice of using MQTT over other messaging protocols is driven by its low use of resources, and the easiness of asynchronous communication between the device and the various components of the system (one-to-many communication) provided by the publish/subscribe messaging system between clients and broker.

2.3 Data Management System

The data management system is composed by an INFLUX database for the storage of all the produced time series, and a GRAFANA dashboard for their visualization. The following time series are stored and plotted:

- Temperature measurements;
- Humidity measurements;
- Wi-Fi RSSI;
- Temperature forecasting;
- Humidity forecasting.

To each time series are associated the ID and the GPS coordinates of the corresponding sensor, in order to keep track of the origin and location of the measurement/forecasting.

time	gps	humidity	id	strength	temperature
1634047558157808700	41.079, 14.542	47.2	MoianoSensor	-54	19.9
1634047559524112200	41.079, 14.542	47.2	MoianoSensor	-53	19.8
1634047560199813800	41.079, 14.542	47.2	MoianoSensor	-54	19.8
1634047561222819200	41.079, 14.542	47.3	MoianoSensor	-54	20
1634047562240589000	41.079, 14.542	47.3	MoianoSensor	-54	20
1634047563265179600	41.079, 14.542	47.2	MoianoSensor	-54	19.8
1634047564282586000	41.079, 14.542	47.2	MoianoSensor	-54	19.8
1634047565308096700	41.079, 14.542	47.3	MoianoSensor	-54	19.8
1634047566522838000	41.079, 14.542	47.3	MoianoSensor	-53	19.8
1634047567350656500	41.079, 14.542	47.4	MoianoSensor	-55	19.9

Figure 2.3: Example of data structure in INFLUX for measured data of a sensor.

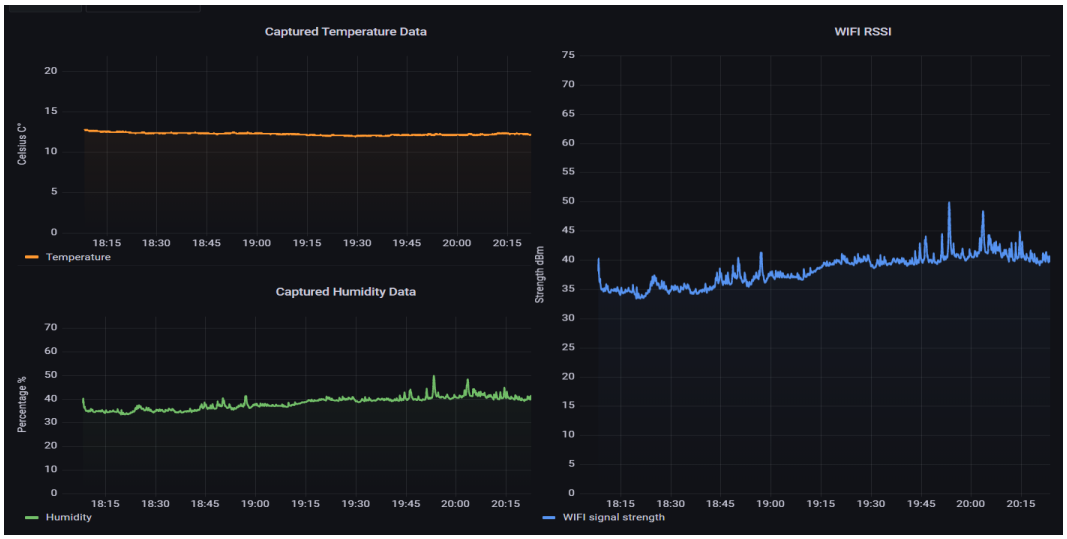

Figure 2.4: Example of data visualization in GRAFANA for measurements and WIFI signal strength.

2.4 Q-Learning algorithm

In order to avoid oversampling and save energy, while maximizing the signal quality, the sampling frequency is dynamically tuned by a Reinforcement Learning algorithm, running on a Python script at the Cloud level. The proposed algorithm is Q-Learning: it is a widely used RL algorithm, with the peculiarity of being model-free: at each time step, the best action is chosen by looking at the current state only.

The main components of a Q-learning algorithm are:

- A set S of possible **states** $s \in S$ of the system;
- A set A of possible **actions** $a \in A$ that the RL agent (in our case, the IoT device) can take;
- A **Q-table** Q, which cells $Q[s, a]$ represent a "quality value" for all possible state-action pairs;
- A **reward function** r_t, which quantifies the performance of the agent at each time step t.

The Q-learning algorithm is then divided in two steps: the *training phase*, in which the Q-table is built and refined, and the *application phase*, where for each state s_t the agent selects the best action a_t by looking at the fixed Q-table as a lookup table. The following sections describe these two steps; the problem modellization is instead described in Chapter 3.

2.4.1 Training phase

The Q-learning training phase is composed by a fixed number of iterations. At each iteration, corresponding to a time step t, the corresponding Q-table cell $Q[s_t, a_t]$ is updated, by means of the following formula:

$$Q_{t+1}[s_t, a_t] = \alpha(r_{t+1} + \gamma \max_a\{Q_t[s_{t+1}a]\} - Q_t[s_t, a_t]) + Q_t[s_t, a_t]$$

Where the two hyperparameters α and γ are the *learning rate* and the *discount factor*. The first determines the training convergence speed (for example, if $\alpha = 0$, we have $Q_{t+1}[s_t, a_t] = Q_t[s_t, a_t]$: a "zero speed training", in other words no training at all), while the second quantifies how much the agent should weight the *future rewards* with respect to an immediate reward (If $\gamma = 0$, the agent learns to maximize the immediate reward, even if this would cause a lower long term payoff). The steps of the training phase are summarized in the figure below.

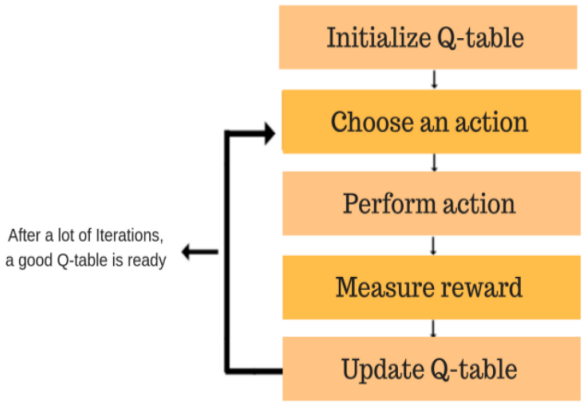

Figure 2.5: Training phase of the Q-learning algorithm.

2.4.2 Application phase

Once the Q-table is ready, choosing the best action for each state is simple: since $Q[s, a]$ represents the overall reward for computing action a in state s, it suffices to take a^* such that $a^* = argmax_x(Q[s, a])$.

2.5 Q-Learning Performance Evaluation

The effectiveness of the Q-learning sampling interval adaptation is evaluated in terms of two factors:

- Number of saved transmissions;
- Degradation on the quality of data collected when the service is active.

In fact, the introduction of a RL approach aims to resolve a trade-off between energy consumption and data quality. The modellization of these two factors is explained in Chapter 3.

2.6 Data Analytics

The Data Analytics component is represented by a Python script in charge of **forecasting** the temperature and humidity values in the next future time steps. The model chosen for the forecasting is a LSTM neural network, taking as input the latest t-k,...,t measurements and giving as output the future t+1,..,t+n predicted values.

LSTMs, an upgrade of Recurrent Neural Networks, are quite useful in time series prediction tasks due to their ability to maintain state and recognize patterns over the length of the time series.

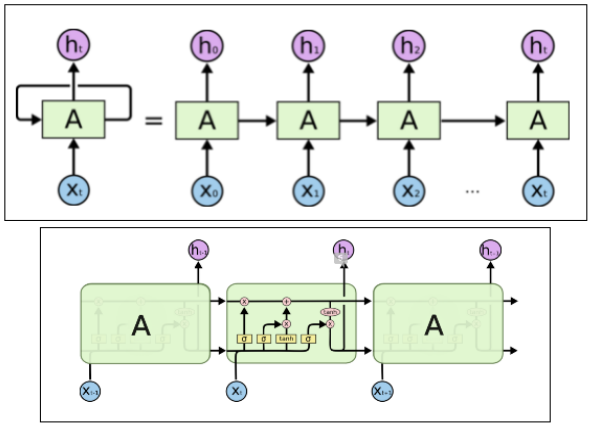

Figure 2.6: Architecture of a LSTM block.

At runtime, a pre-trained model is loaded for the forecasting; the LSTM training is previously done offline with a dataset of past measurements coming from the same IoT device in the considered location.

2.7 Web Dashboard

The Web Dashboard is a Django Python web application that allows the admin user to monitor and manage sensors.

Specifically, the functions are the following:

- View previously registered sensors with their relative GPS position;
- View and configure the parameters of the individual sensors;
- Add a new sensor to the system;
- Access the Grafana dashboards related to each sensor.

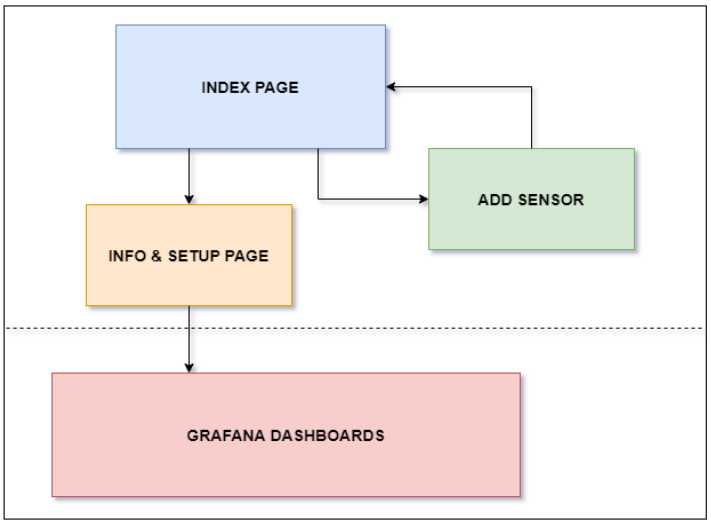

Figure 2.7: Overview of the Web Dashboard architecture.

The structure of the project is subdivided by following Django's typical MVT (Model-View-Template) logic.

- **Model:** Python module dedicated to Object–relational mapping (ORM), which is the mapping of database data in python objects;
- **View:** Python module that handles HTTP requests and responses;
- **Template:** A Client-side module that contains the static parts of the desired HTML output as well as some special syntax describing how dynamic content will be inserted.

In the Client-side there is also the **static** module, which contains Javascript scripts and CSS stylesheets that deal with the decoration and interactivity of the user interface.

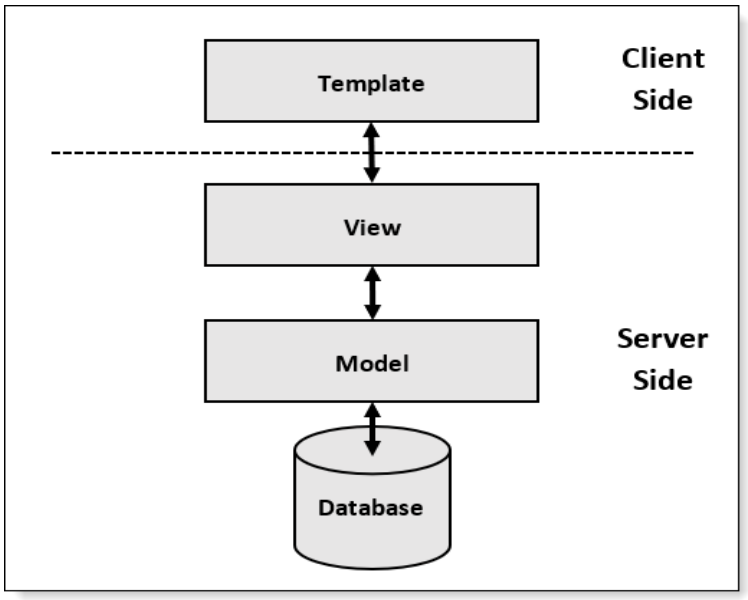

Figure 2.8: A slightly different view of Django's MTV stack.

Chapter 3

Project's Implementation

3.1 IoT smart device

The ESP32 has been programmed in Arduino; the loaded code can be summarized into three sections, namely *initialization*, *setup* and *loop*.

3.1.1 Initialization

The initialization part is reserved to the definition of all the constants and initial values of the program variables. We define:

1. The Wi-Fi ID and password;
2. The MQTT server IP, port, username and password;
3. The DHT22 type and pin number;
4. The ESP32 LED pin number;
5. The MQTT broker channel topics, in order to send (temperature, humidity, gps coordinates, device ID, WiFi RSSI) and receive (min/max temperature, min/max humidity, sampling frequency) data;
6. The device ID and GPS coordinates;
7. The starting values for min/max temperature, min/max humidity and sampling frequency.

The safety boundaries have been initialized to the following values:

- $MIN\_TEMP = 25.0$;
- $MAX\_TEMP = 35.0$;
- $MIN\_HUM = 40.0$;

- $MAX\_HUM = 60.0$.

And the $SAMPLE\_FREQ$ has been initialized to $1000ms$. In order to apply the Q-learning algorithm, we restricted the possible values for the sampling frequency to the set $\{250ms, 500ms, 1000ms\}$.

3.1.2 Setup

In the setup part we start collecting data from the DHT22, and connect to Wi-Fi and MQTT broker; we also subscribe to all the MQTT channel topics needed by the ESP32 to receive data.

3.1.3 Loop

In the loop part the IoT device is operative: at each cycle we:

1. Check, by means of a callback, if a command to change the sampling frequency or a safety boundary has arrived. If that is the case, the corresponding value is updated;
2. Read from the DHT22 the current temperature and humidity measurements, together with the current Wi-Fi RSSI;
3. Check if the measurements fall outside their respective safety boundaries. If so, a command to turn on the ESP32 LED is raised until the check of the next cycle;
4. Structure the output on a json document, in order to create a packet that will be published to the broker;
5. Apply a delay in order to enact the chosen sampling frequency.

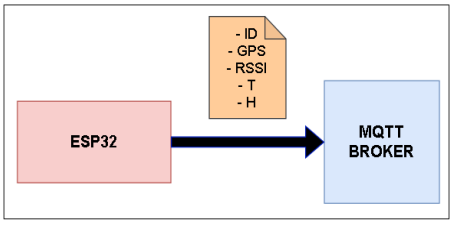

Figure 3.1: The json packet sent by the ESP32 to the broker.

Overall, the ESP32 command flow is synthesized in Fig.3.2.

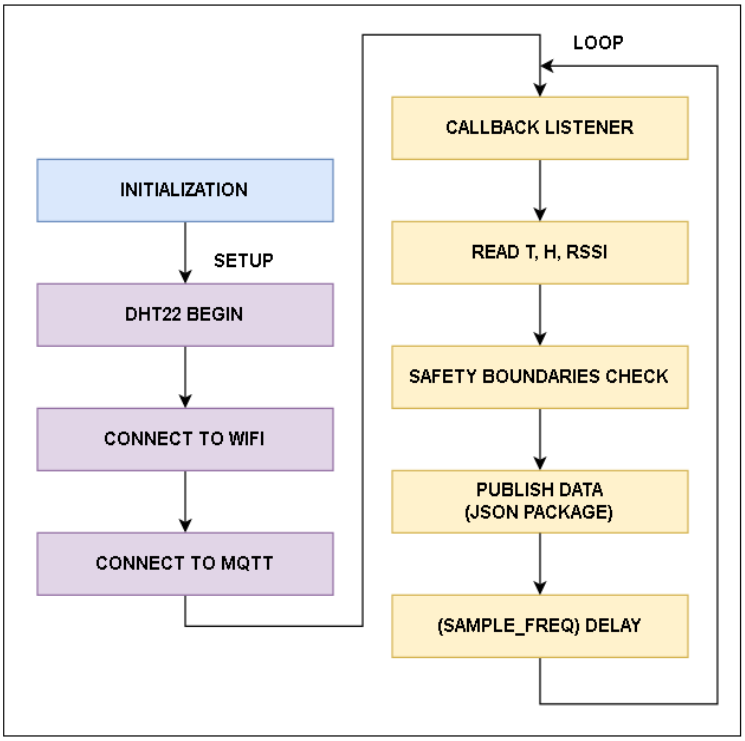

Figure 3.2: The ESP32 command flow.

3.2 Data Bridge

The Data Bridge has been programmed in Python; the loaded code can be summarized into three sections, namely *initialization*, *function definition* and *main*.

3.2.1 Initialization

The initialization part is reserved to the definition of all the constants and initial values of the program variables. We define:

1. The MQTT server IP, port, username, password and client name;
2. The INFLUXDB server IP, port, username, password and database;
3. The influxdb_client passing the previously defined parameters;
4. The topic name for subscribe at the relative MQTT topic.

3.2.2 Function definition

In the function definition part we define the utility functions for the data bridge. We define:

1. **_init_influxdb_database:** Initialize the influxDB client with the parameters previously defined;
2. **_send_sensor_data_to_influxdb:** Parse the data captured in a JSON structure and send it at the influxDB database.
   The JSON structure consists of the following entries:
   {*measurement*, *tags* { *gps, id* }, *fields* { *strength, temperature, humidity* };
3. **on_connect:** mqtt-paho callback for when the client receives a CONNACK response from the server;
4. **on_message:** mqtt-paho callback for when a PUBLISH message is received from the server.

3.2.3 Main

The main part is where the script starts up.
MQTT Client and INFLUXDB Client are initialized and an infinite loop is started where messages with topic = "quanto" are captured and saved on the timeseries database.

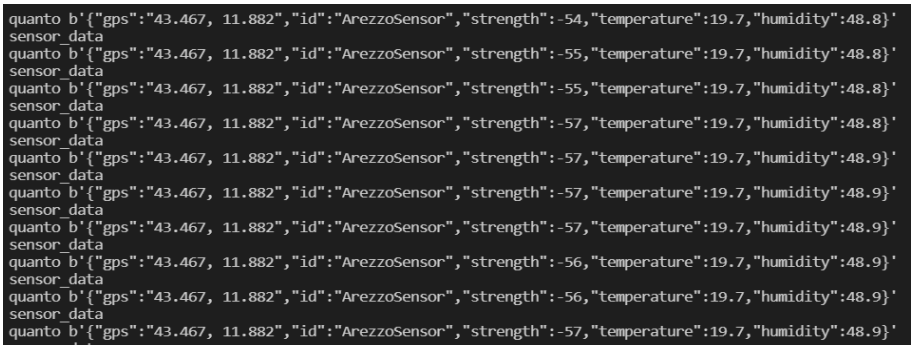

Figure 3.3: Example of data captured by the Data Bridge.

3.3 Data Management System

The Data Management System layer of our architecture is composed by an InfluxDB timeseries database and two Grafana dashboards for analytics and interactive visualizations of the stored data.

3.3.1 InfluxDB

We collect the data collected by sensors in *weather_stations* timeseries database. This is made up of 7 different measurements:

1. **quanto:** It represents the data saved by a sensor at a precise moment. The structure is defined as follows:
   (a) **time:** The timestamp indicating the date and time the row was saved (tag);
   (b) **id:** The unique identifier of the sensor that captured the data (tag);
   (c) **gps:** The GPS coordinates of the sensor that captured the data (tag);
   (d) **temperature:** The temperature in Celsius degrees captured by the sensor at that instant (field);
   (e) **humidity:** The humidity in percentage points captured by the sensor at that instant (field);
   (f) **strength:** received signal strength indicator (RSSI) of the message with which the data was received (field).
2. **predict_hum_10s:**
   (a) **time:** The timestamp indicating the date and time of the prediction (tag);
   (b) **id:** The unique identifier of the sensor for which the prediction was made (tag);
   (c) **temperature:** The predicted humidity in percentage points (field).
3. **predict_hum_20s:**
   (a) **time:** The timestamp indicating the date and time of the prediction (tag);
   (b) **id:** The unique identifier of the sensor for which the prediction was made (tag);
   (c) **temperature:** The predicted humidity in percentage points (field).
4. **predict_hum_30s:**
   (a) **time:** The timestamp indicating the date and time of the prediction (tag);
   (b) **id:** The unique identifier of the sensor for which the prediction was made (tag);
   (c) **temperature:** The predicted humidity in percentage points (field).
5. **predict_temp_10s:**
   (a) **time:** The timestamp indicating the date and time of the prediction (tag);

(b) **id:** The unique identifier of the sensor for which the prediction was made (tag);

(c) **temperature:** The predicted temperature in Celsius degrees (field).

6. **predict_temp_20s:**

   (a) **time:** The timestamp indicating the date and time of the prediction (tag);

   (b) **id:** The unique identifier of the sensor for which the prediction was made (tag);

   (c) **temperature:** The predicted temperature in Celsius degrees (field).

7. **predict_temp_30s:**

   (a) **time:** The timestamp indicating the date and time of the prediction (tag);

   (b) **id:** The unique identifier of the sensor for which the prediction was made (tag);

   (c) **temperature:** The predicted temperature in Celsius degrees (field).

3.3.2 Grafana

The Dashboards created on Grafana are 2: *"Weather Data"* and *"Weather Data with predictions"*.
"Weather Data" contains 3 different timeseries line charts printed on 3 different plots: one represents the sampled temperature, another the sampled humidity, the other the RSSI.
"Weather Data with predictions" contains 6 different plots. The firsts 3 of these represent the sampled temperature coupled, respectively for each of the graphs, with the prediction in the next 10, 20 and 30 seconds.
The other 3 of these represent the sampled humidity coupled, respectively for each of the graphs, with the prediction in the next 10, 20 and 30 seconds.

3.4 Q-Learning algorithm

Our Q-learning implementation modelizes the problem as follows:

- A **state** is a pair $S = (q, SR)$, where q is the *data quality*, defined by starting from the sum of absolute differences between two consecutive temperature and humidity measurements:

  $$e_t = |temp_t - temp_{t-1}| + |hum_t - hum_{t-1}|$$

  and the SR is the device's current *sampling rate*. In order to discretize the states, we assume that the sampling rate can only assume one of these three values: $sr \in \{250ms, 500ms, 1000ms\}$.

As for q discretization, given a (hyperparameter) threshold τ, the data quality can either be:

  - Too Low: $q = L$ *if* $\tau \leq e_t$
  - Perfect: $q = P$ *if* $\frac{1}{2}\tau < e_t < \tau$
  - Too High: $q = H$ *if* $e_t \leq \frac{1}{2}\tau$

  The rationale behind this is that while a data quality below τ is never acceptable, a data quality too high suggests that we could reduce the sampling frequency, and thus saving energy, while respecting the given threshold.

- For each state, we have three possible **actions**:

  - Halve the sampling rate: -SR
  - Maintain the sampling rate: =SR
  - Double the sampling rate: +SR

  In other words: $a \in \{+SR, = SR, -SR\}$

- At each step t, the **reward** r_{t+1} is computed as:

$$r_{t+1} = K\frac{s_{t+1}}{s_0}, \textit{ where } K = \begin{cases} +1.5 & \textit{if } q = P \\ +1 & \textit{if } q = H \\ -1 & \textit{if } q = L \end{cases}$$

  maximizing the tradeoff between energy savings and data quality. S_0 is $250ms$, the highest frequency rate. The ratio $\frac{s_{t+1}}{s_0}$ represents the rate of transmissions avoided. For example, if the sampling interval is 1000ms, the device is transmitting four times less than if it was 250ms.

- We set the *learning rate* $\alpha = 0.9$ and the *discount factor* $\gamma = 0.2$. These values come from the fact that we found that the device works better with high values of α and low values of γ: the device seems to perform better when its decisions are mostly based on the current status of the environment, while low importance is given to future estimated rewards.

  So, as for the problem modellization, we have the following Q-table:

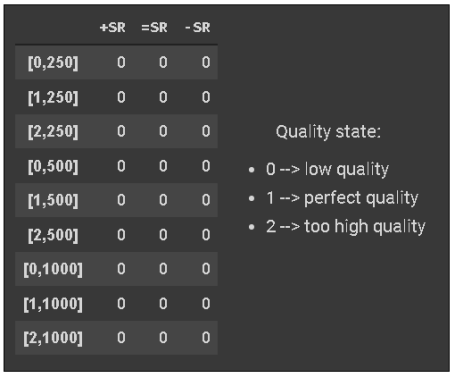

Figure 3.4: The model's Q-table, with all cells initialized to zero.

In the training loop the Q-table cell values will be computed and optimized. In this phase, since we don't know yet which action is best in each state, at each iteration we choose the action by means of an ϵ-*greedy strategy*: we usually take the currently best action (the one with the highest Q-table value for that state), but there is an ϵ% chance that the action is instead chosen randomly. ϵ is an hyperparameter; in our implementation we chose $\epsilon = 20\%$.

The rest of the training loop is illustrated in Fig.3.4, along with the subsequent application phase.

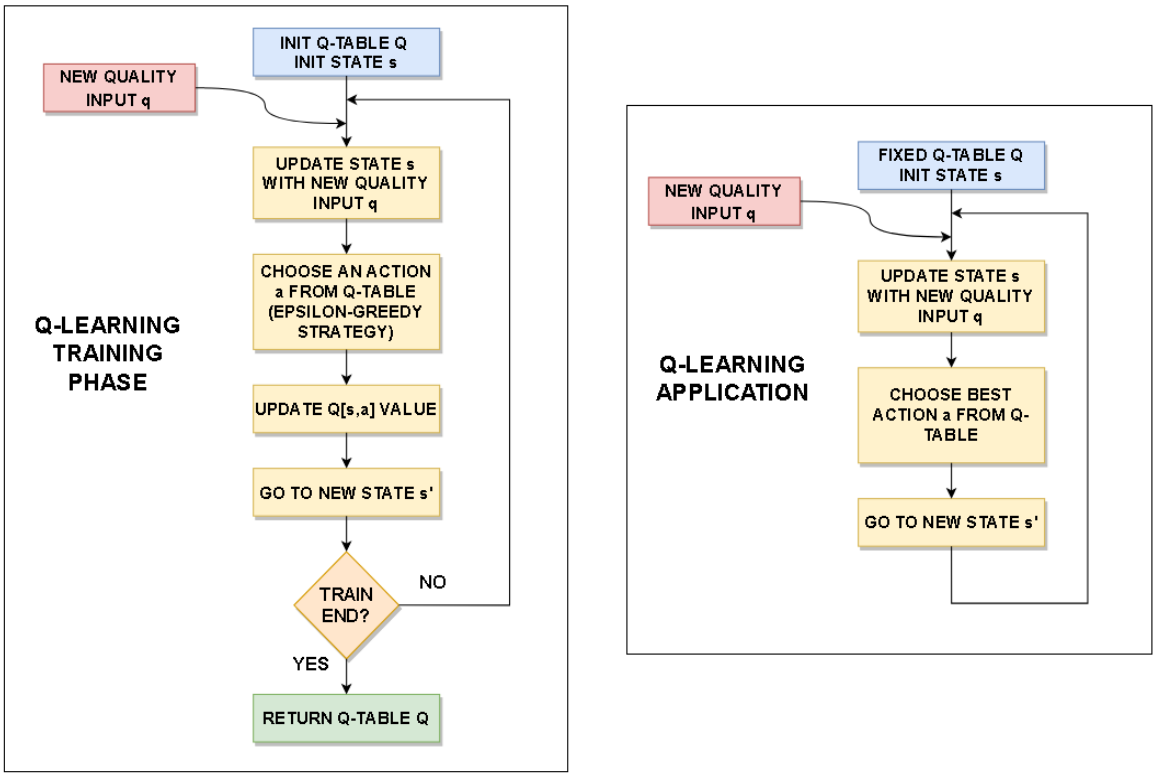

Figure 3.5: Q-learning algorithm, in both training and application phases.

3.5 Q-Learning Performance Evaluation

As a first step, we define an *observation window* T, during which we count the number of samples and monitor the data quality. In our implementation, we chose T = 300s (5min). Then, the Q-learning performances are evaluated in terms of *number of saved transmissions* and *data quality degradation*. These concepts are associated with two metrics:

- % of saved transmission with respect to baseline case (where the baseline case is a fixed sample rate of 250ms): $\frac{N_{tot\_samples}}{N_{samples\_if\_250ms}} \in [0, 1]$;
- % of data with quality under threshold: $\frac{N°of\ q = L}{N°of\ q} \in [0, 1]$.

We then computed these metrics with various combinations of hyperparameters, and collected the results of each experiment in a csv file. This allows to find the best hyperparameters combination for the Q-learning model.

3.6 Data Analytics

The data analytics section is represented by a Python script in charge of forecasting the future values of temperature and humidity measurements, up to some future target times; in our implementation, the forcasting time windows are 10, 20 and 30 seconds in the future (where to a wider window corresponds a lower forecasting accuracy).

In order to do so, we make use of two LSTM neural networks, one for each time series, whose architecture is depicted in Fig.3.6:

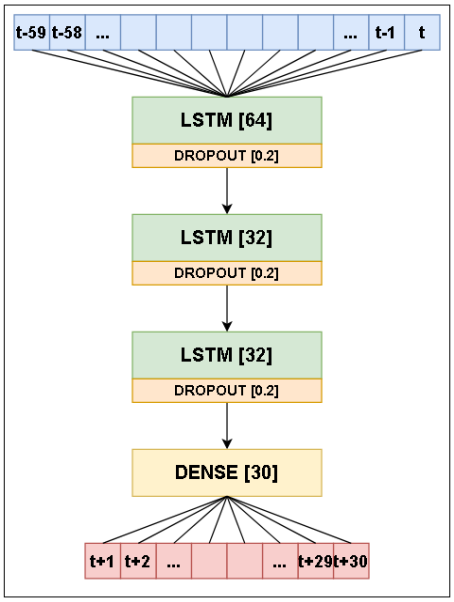

Figure 3.6: The LSTM network architecture.

The network takes as input the values of the latest 60 seconds of the time series (with a timestep of 1s), and then exploits a funnel structure of three consecutive LSTM layers to process the data. After each layer, a dropout layer with rate 0.2 is applied during training in order to prevent model overfitting. A final Dense layer, with a number of neurons equal to the maximum forecasting window (in our implementation, 30), gives as output the forecast values.

It should be noted that the model outputs the target forecasting sequences in one forward operation, rather than in a step-by-step procedure typical of iterative models. This design choice comes from two main considerations:

- **The available time for forecasting is limited.** Since the forecasting is executed at each new measurement, given the possible sampling frequences the model could have less than 250ms to predict up to 30s in the future.

While an iterative prediction is time-expensive, the one-shot prediction of the full 30s window drastically improves the inference speed;

- **Some SOTA architectures for Time Series Forecasting are non-iterative.** The one-shot prediction strategy is used by a number of architectures (for example, the Informer) able to reach State-of-the-art results in the time series forecasting problem.

Once predicted the $t+1, ..., t+30$ forecasting window, the corresponding 10s, 20s and 30s values are sent to INFLUX in order to build the temperature and humidity forecasting time series, which will then be shown on GRAFANA.

It must be highlighted that the model is trained offline by a second Python script: prior to the implementation of the forecasting system, we fit the model with a dataset of many past measurements and evaluate the training by means of a test set (in out implementation, we used 20000 timesteps for the training dataset, and around 4000 timesteps for the test set). Once the training is done, in the online phase (when the forecasting is active) we simply load the model weights and proceed.

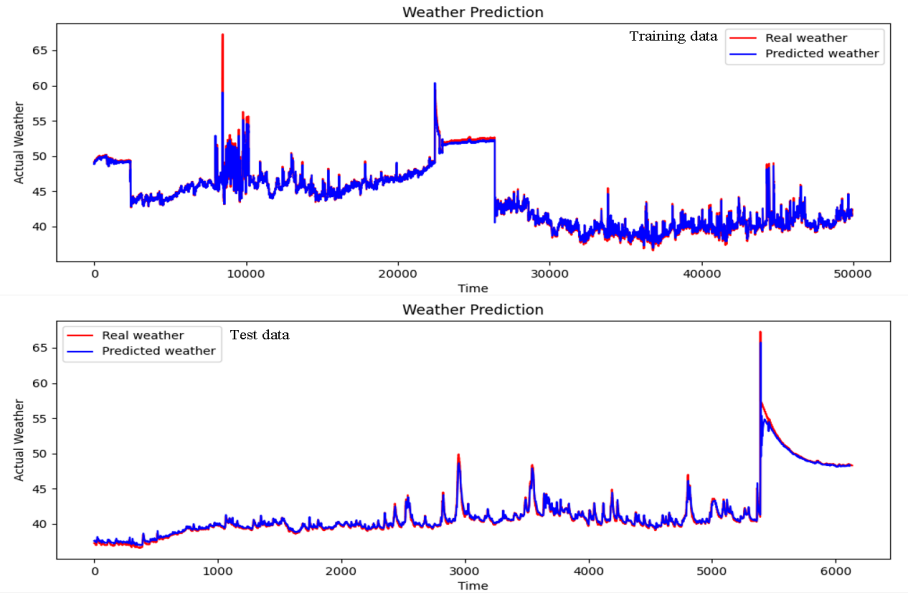

Figure 3.7: Example of humidity forecasting results for train and test datasets.

A note about the 60-elements time-series used as input for the forecasting: since the stored measurements are not evenly spaced in time (due to the variable sampling frequency) we obtain the target input from an interpolation of the lat-

est 60 seconds's data in INFLUX, by calling a query that groups measurements with a granularity of 1s.

3.7 Web Dashboard

3.7.1 Server Side

The Server-side is relegated to data processing and sending/receiving input/output with clients.

The *utils.py* module is used to instantiate the influxDB client and the MQTT Client and manage the latter's callbacks.

The *model.py* module have only a class called *Sensor*. This allows us to define the unique and non-null sensor object, which are used to keep track of the sensors registered on the platform.

The *views.py* module presents all the logic of receiving / sending the http request / response. It is composed by the following components:

- **index:** the view function that generates the main page of the application, returning the *index.html* template and the previously recorded sensor data;
- **add_sensor:** the view function to register new sensors. After registration you are redirected to the main page, with an error message in case of failure;
- **sensor_info:** the view function that generates the page with the information of the selected sensor with its settings and last sampled data. In case of absence of the selected sensor, you will be redirected to the missing_sensor.html page;
- **missing_sensor:** the view function that generates an error page in case of missing address;
- **change_sensor_parameter:** the view function to edit the sensor settings. After registration you are redirected to the info page of the edited sensor.

3.7.2 Client Side

The Client-side is relegated to the **template** and **static** modules which respectively deal with the structure of the web page and its appearance/behavior.
The **template** module contains the following elements:

- **index.html:** The structure of the main web page of the web dashboard, where it is possible to view previously registered sensors and to register new ones;
- **sensor_info.html:** The HTML page dedicated to a single sensor and to the setting of its parameters;

- **missing_info:** The error page that appears if the selected sensor is currently unavailable.

The **static** module contains the following elements:

- **index.js:** The Javascript file used to modify the Document Object Model (DOM) of application pages;
- **maps.js:** Contains the script that allows you to pass data to the Google Maps API to represent the coordinates of the sensors on the map;
- **chart.js:** Contains the script that allows you to pass the data to the Plotly API to represent the graphs with the sensor data.

Furthermore, external frameworks and libraries were used:

- **Bootstrap:** A free and open source CSS framework used to arrange and decorate the user interface;
- **Google Maps API:** The Google API to be able to manage the map and the coordinates of the registered sensors;
- **Plotly Graphing Library:** A JavaScript library used to plot a preview of sensor sampled data.

Chapter 4

Results

4.1 Q-learning sampling interval adaptation

The performances, in terms of saved transmissions and quality degradation, of the Q-learning algorithm as a function of the hyperparameters are shown in Fig.4.1:

% of saved transmissions		Discount Factor γ		
		0.3	0.6	0.9
Learning rate α	0.3	28.08	27.63	29.32
	0.6	35.52	33.24	31.04
	0.9	39.25	35.96	34.31

% of measurements over τ		Discount Factor γ		
		0.3	0.6	0.9
Learning rate α	0.3	18.24	21.25	17.21
	0.6	15.14	13.53	12.54
	0.9	13.41	14.52	13.24
Baseline : No RL (α = 0)		9.14		

Figure 4.1: Q-learning performances results.

As we can see, the device works better with high values of α and low values of γ: the device seems to perform better when its decisions are mostly based on the current status of the environment, while low importance is given to future estimated rewards.

Overall, the sampling interval adaptation scheme performs very well: By looking at the results of the best α-γ combination, we can see that in the best combination we achieve:

- A reduction of 39% in the N° of transmissions, with respect to a fixed 250ms sampling;
- A quality of signal comparable to the fixed 250ms sampling.

4.2 Data Forecasting

The model seems to perform well also on the forecasting part. As we can see in Fig.4.2, the model is responsive to environmental changes: we can see that the increase in humidity at 20:35 from 40 %RH to 48 %RH is correctly predicted.

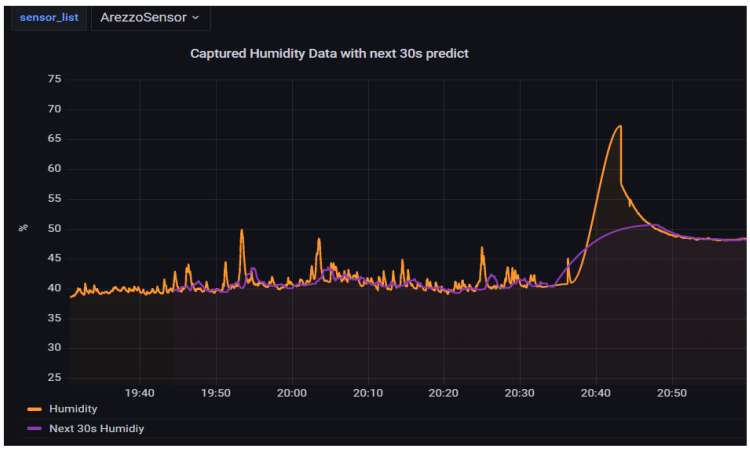

Figure 4.2: Example of forecasting in which the model predicts a major humidity shift.

An example of forecast time series for both temperature and humidity values is depicted in Fig.4.3.

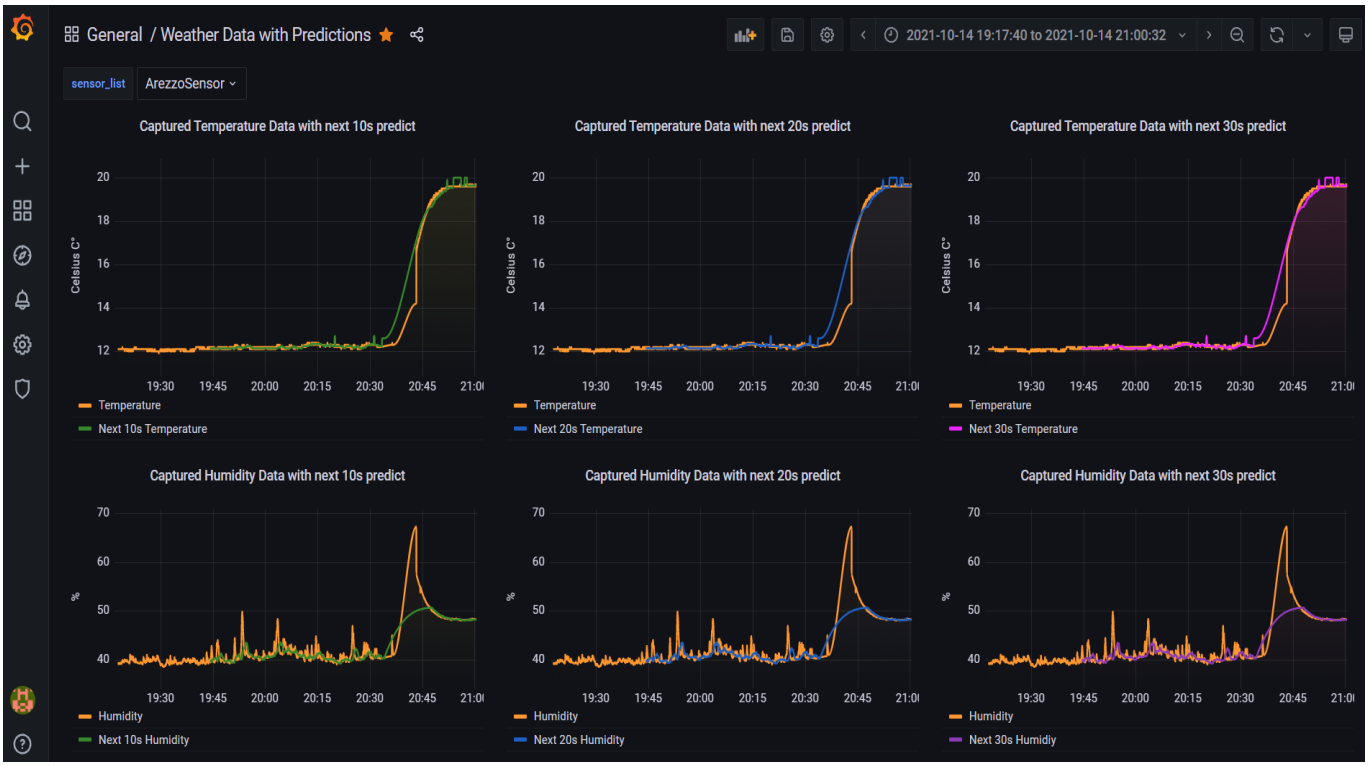

Figure 4.3: Forecasted values related to measured values, as shown on GRAFANA.

4.3 Final view on web Dashboard

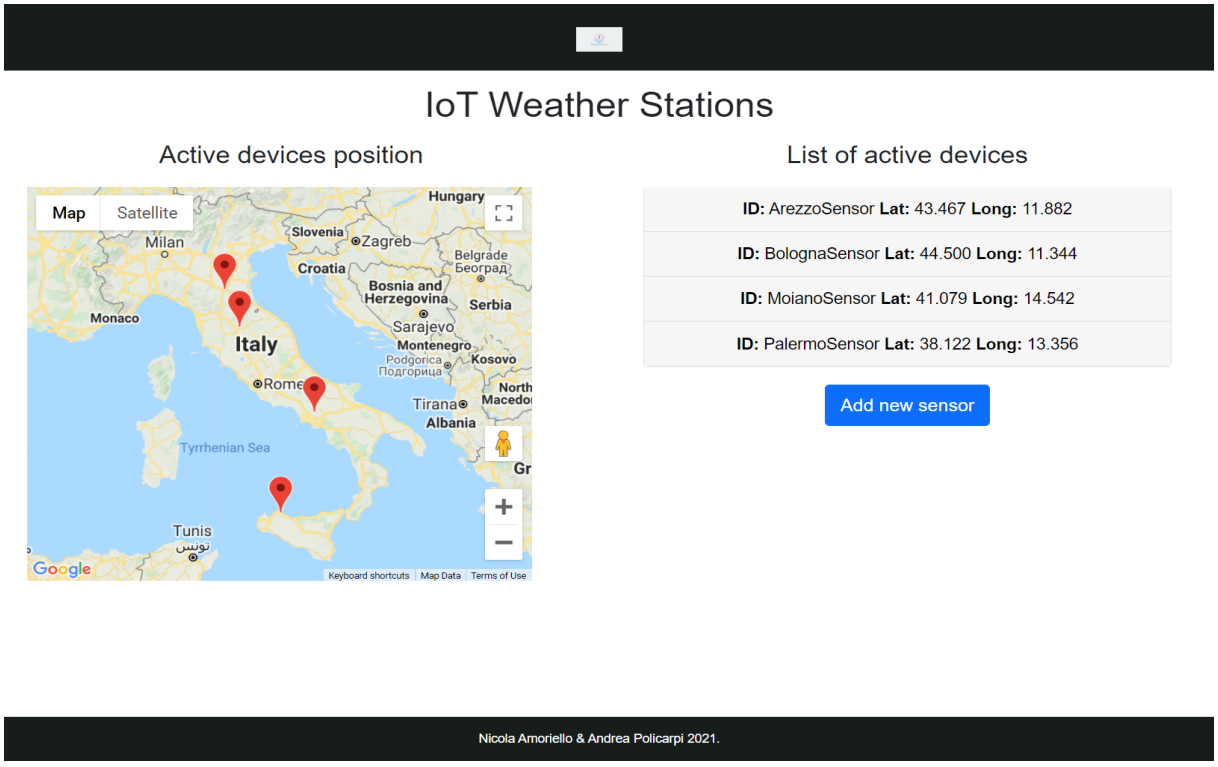

Figure 4.4: Main page of the web dashboard, with the list of active devices and their position on the map.

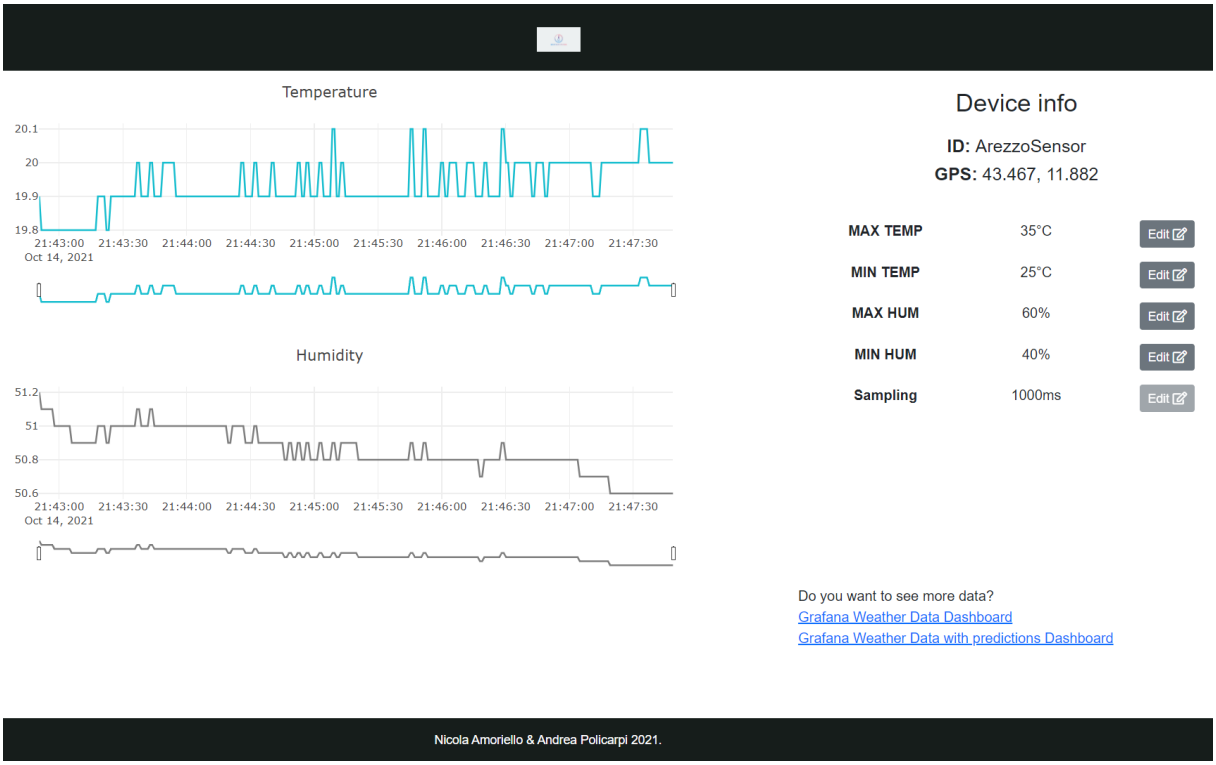

Figure 4.5: Example of page associated to each device. On the right we have the time series graphs, while on the left are listed the sensor parameters, with the possibility to change them at runtime. On the bottom-left, there is the link to GRAFANA.