Embedded control with movement detection: Utilizing MPU9250, MQTT, Node-RED, and InfluxDB and machine learning in esp32.

Nikolaos Mouzakitis

May 24, 2025

1 Introduction

This project integrates an ESP32 microcontroller with an MPU9250 inertial measurement unit (IMU) sensor to detect device movements and control 2 LEDs and PWM outputs accordingly. It implements a mechanism to persist and restore device state via MQTT messaging, Node-RED flow processing, and data storage in an InfluxDB time-series database. Also Grafana is used in the last part of the project for visualization.

The key functionalities are:

- Reading accelerometer and gyroscope Y-axis, and magmetometer X-axis data from the MPU9250.
- Applying threshold-based logic or machine learning to detect movement and determine LED and PWM states.
- Publishing device status data over MQTT.
- Restoring device LED and PWM states after reboot using InfluxDB queries and Node-RED.

An image of the Esp32 and the MPU sensor used in the project is presented below.

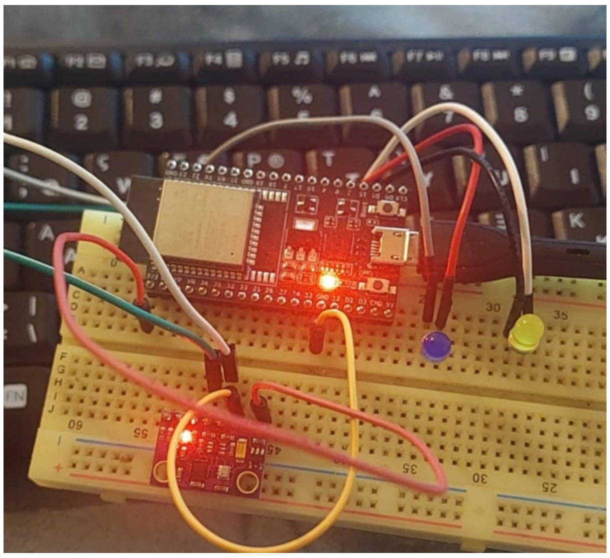


Figure 1: esp32 and mpu connected on breadboard.

2 MPU9250 Sensor Data and Movement Detection

2.1 Sensor Overview

The MPU9250 is a 9-axis IMU sensor combining a 3-axis accelerometer, 3-axis gyroscope, and 3-axis magnetometer. For this project, only the accelerometer Y-axis,gyroscope Y-axis and magnetometer X-axis values are utilized.

2.2 Data Acquisition

The ESP32 reads raw sensor values for accelerometer Y (`ay`) ,gyroscope Y (`gy`) which are used to control the activation of the two LEDs, and magnetometer X (`mx`) which controls the PWM, at periodic intervals.

2.3 Threshold-Based Movement Detection

Threshold values are defined empirically to decide when the ESP32 should change the states of its LEDs and PWM output.

2.4 Controlling LEDs and PWM

Based on movement detection, the device sets the LEDs and a common PWM variable which is applied on any LED that is on "active" state.

2.5 Status Publishing

The ESP32 publishes its current LED and PWM states on the MQTT topic `esp32/status` as a formatted string:

```
LED1:OFF LED2:ON PWM:15
```

This message serves to provide the real-time status of device outputs.

3 MQTT Communication Architecture

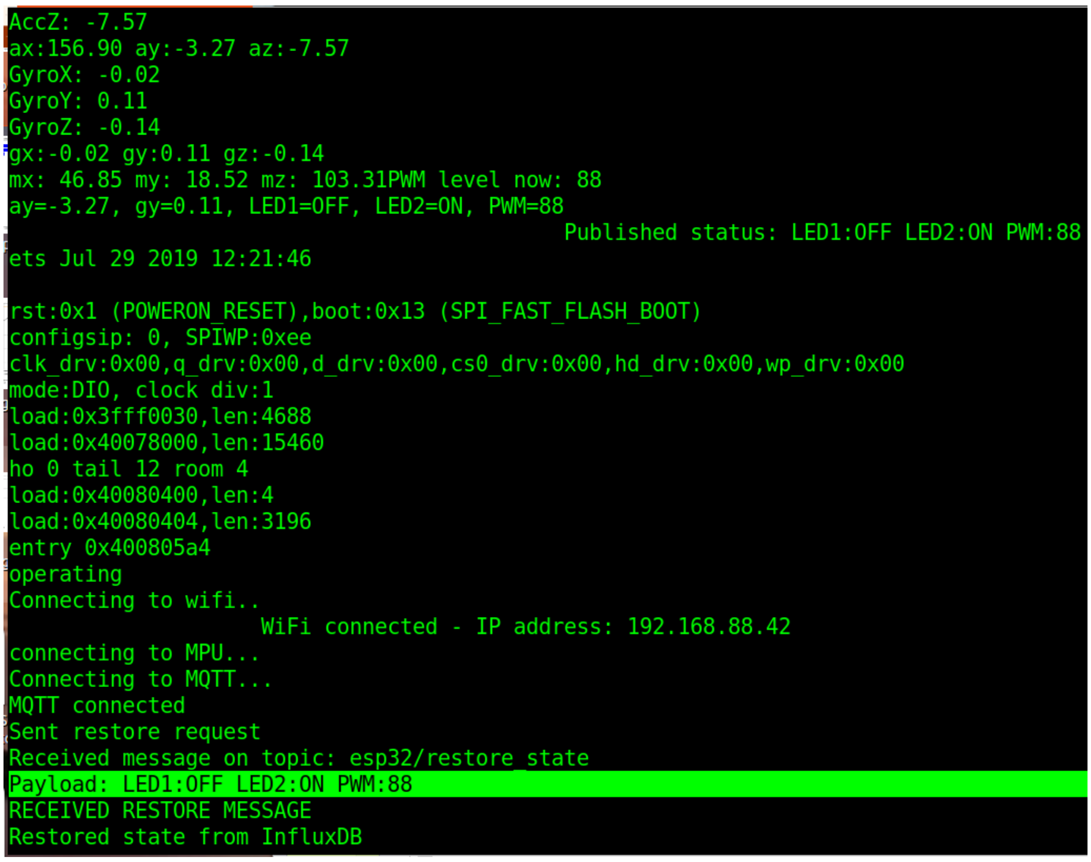
The ESP32 communicates with the Node-RED server via MQTT. Topics used include:

- `esp32/status` — Periodic status updates including LED and PWM states.
- `esp32/restore_request` — Sent by ESP32 at startup requesting last stored device state.
- `esp32/restore_state` — Published by Node-RED after querying InfluxDB with the restore state data.

3.1 Restore request and response flow

Upon powering on, on startup, the ESP32 immediately publishes a `get_latest` message on `esp32/restore_request`. This triggers Node-RED to query the latest device state stored in InfluxDB and respond by publishing a formatted restore state message on `esp32/restore_state`.

The ESP32 listens for this restore message and applies the LED and PWM settings accordingly before continuing normal operation. Restoration can be observed in the following Figure, where the Esp32 boots and assigns immidiately the latest saved state on the LEDs.



```
AccZ: -7.57
ax:156.90 ay:-3.27 az:-7.57
GyroX: -0.02
GyroY: 0.11
GyroZ: -0.14
gx:-0.02 gy:0.11 gz:-0.14
mx: 46.85 my: 18.52 mz: 103.31PWM level now: 88
ay=-3.27, gy=0.11, LED1=OFF, LED2=ON, PWM=88
                                            Published status: LED1:OFF LED2:ON PWM:88
ets Jul 29 2019 12:21:46

rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0030,len:4688
load:0x40078000,len:15460
ho 0 tail 12 room 4
load:0x40080400,len:4
load:0x40080404,len:3196
entry 0x400805a4
operating
Connecting to wifi..
                    WiFi connected - IP address: 192.168.88.42
connecting to MPU...
Connecting to MQTT...
MQTT connected
Sent restore request
Received message on topic: esp32/restore_state
Payload: LED1:OFF LED2:ON PWM:88
RECEIVED RESTORE MESSAGE
Restored state from InfluxDB
```

Figure 2: Latest state restoration as observed over UART

4 Node-RED and InfluxDB Integration

4.1 Node-RED Flow Design

Node-RED acts as the middleware for the bridging of MQTT and InfluxDB:

1. **MQTT In node** listens on `esp32/restore_request` and detects the `get_latest` message.
2. **InfluxDB Query node** performs a time-series query fetching the most recent LED and PWM state values.
3. **Function node** formats the InfluxDB query response into the MQTT payload string.
4. **MQTT Out node** publishes the formatted message on `esp32/restore_state`.

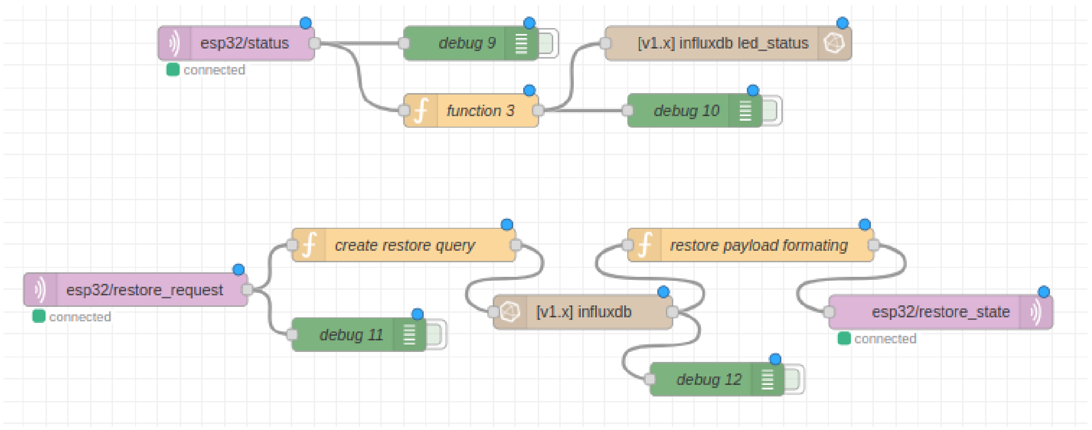


Figure 3: Node-Red data and processing flow

4.2 InfluxDB Query

As for the InfluxDB, version 1.6.4 has been utilized and installed as

```
echo "deb https://repos.influxdata.com/ubuntu bionic stable" |  sudo tee /etc/apt/sources.list.d/influxdb.list
sudo curl -sL https://repos.influxdata.com/influxdb.key | sudo apt-key add -
sudo apt get update
sudo apt-get install -y influxdb
```

After installation, a database is created to be used for this project.

```
influx
> create database esp32db
```

In `function3` of the Node-Red then, when run the following Javascript code in order to push data in the same database.

```
let parts = msg.payload.split(" ");
let led1 = parts[0].split(":")[1] === "ON" ? 1 : 0;
let led2 = parts[1].split(":")[1] === "ON" ? 1 : 0;
let pwm = parseInt(parts[2].split(":")[1]);

msg.payload = {
    LED1: led1,
    LED2: led2,
    PWM: pwm
};
msg.measurement = "led_status";
return msg;
```

On the InfluxDb out node in Node-Red is also specified and set the measurement's name as "led_status" and on the server configuration the respective host ip, port number and database name as "esp32db".

The InfluxDB query used in `create_restore_query` node is:

```
msg.query = 'SELECT LAST("LED1"), LAST("LED2"), LAST("PWM") FROM "led_status"';
return msg;
```

This query retrieves the most recent LED1, LED2, and PWM values stored.

In the `restore payload formating` node of Node-Red response if formated to be given as an input on the `restore_state` topic as:

```
let row = msg.payload[0]; // first row of result

let LED1 = row.last === 1 ? "ON" : "OFF";
let LED2 = row.last_1 === 1 ? "ON" : "OFF";
let PWM = row.last_2;

msg.payload = `LED1:${LED1} LED2:${LED2} PWM:${PWM}`;
msg.topic = "esp32/restore_state";
return msg;
```

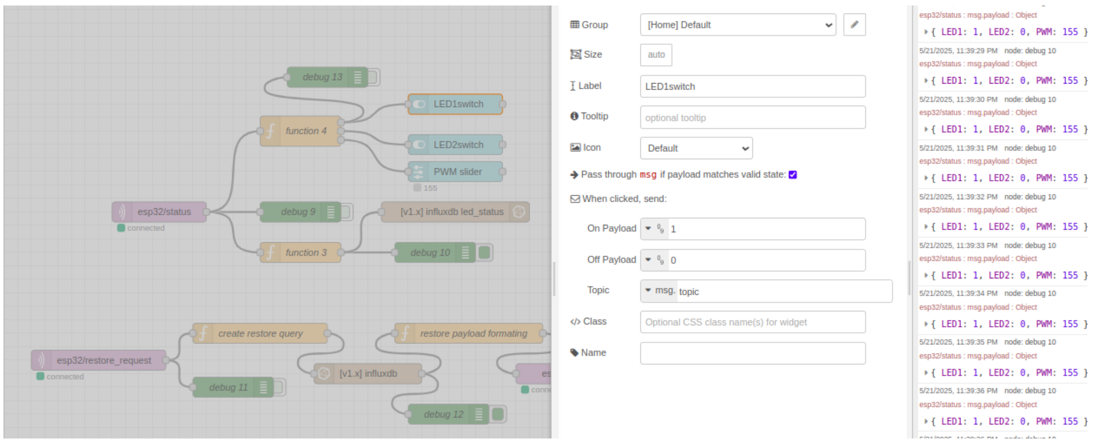
4.3 State Restoration

Upon receiving the restore payload, the ESP32 parses the string and applies the saved LED and PWM states, ensuring the desired recovery.

4.4 UI visualization

For the UI visualization 2 switches reflecting the current state of LED have been employed and a slider for representing the common PWM applied to them.

A new function named `function4` have been used to do the same operation as `function3` with the difference of returning3 outputs, one by one to be passed on the switches and the slider.



Group: [Home] Default
Size: auto
Label: LED1switch
Tooltip: optional tooltip
Icon: Default
Pass through msg if payload matches valid state: ☑
When clicked, send:
On Payload: 1
Off Payload: 0
Topic: msg. topic
Class: Optional CSS class name(s) for widget
Name:

```
esp32/status : msg.payload : Object
▸{ LED1: 1, LED2: 0, PWM: 155 }
5/21/2025, 11:39:29 PM   node: debug 10
esp32/status : msg.payload : Object
▸{ LED1: 1, LED2: 0, PWM: 155 }
5/21/2025, 11:39:30 PM   node: debug 10
esp32/status : msg.payload : Object
▸{ LED1: 1, LED2: 0, PWM: 155 }
5/21/2025, 11:39:31 PM   node: debug 10
esp32/status : msg.payload : Object
▸{ LED1: 1, LED2: 0, PWM: 155 }
5/21/2025, 11:39:32 PM   node: debug 10
esp32/status : msg.payload : Object
▸{ LED1: 1, LED2: 0, PWM: 155 }
5/21/2025, 11:39:33 PM   node: debug 10
esp32/status : msg.payload : Object
▸{ LED1: 1, LED2: 0, PWM: 155 }
5/21/2025, 11:39:34 PM   node: debug 10
esp32/status : msg.payload : Object
▸{ LED1: 1, LED2: 0, PWM: 155 }
5/21/2025, 11:39:35 PM   node: debug 10
esp32/status : msg.payload : Object
▸{ LED1: 1, LED2: 0, PWM: 155 }
5/21/2025, 11:39:36 PM   node: debug 10
esp32/status : msg.payload : Object
▸{ LED1: 1, LED2: 0, PWM: 155 }
```

Figure 4: configuration show for slider of led1.

A screenshot of the visualization is presented below, having the 2 switches and the slider, and for debug purposes the plots of the incoming data from accelerometer and gyrometer.

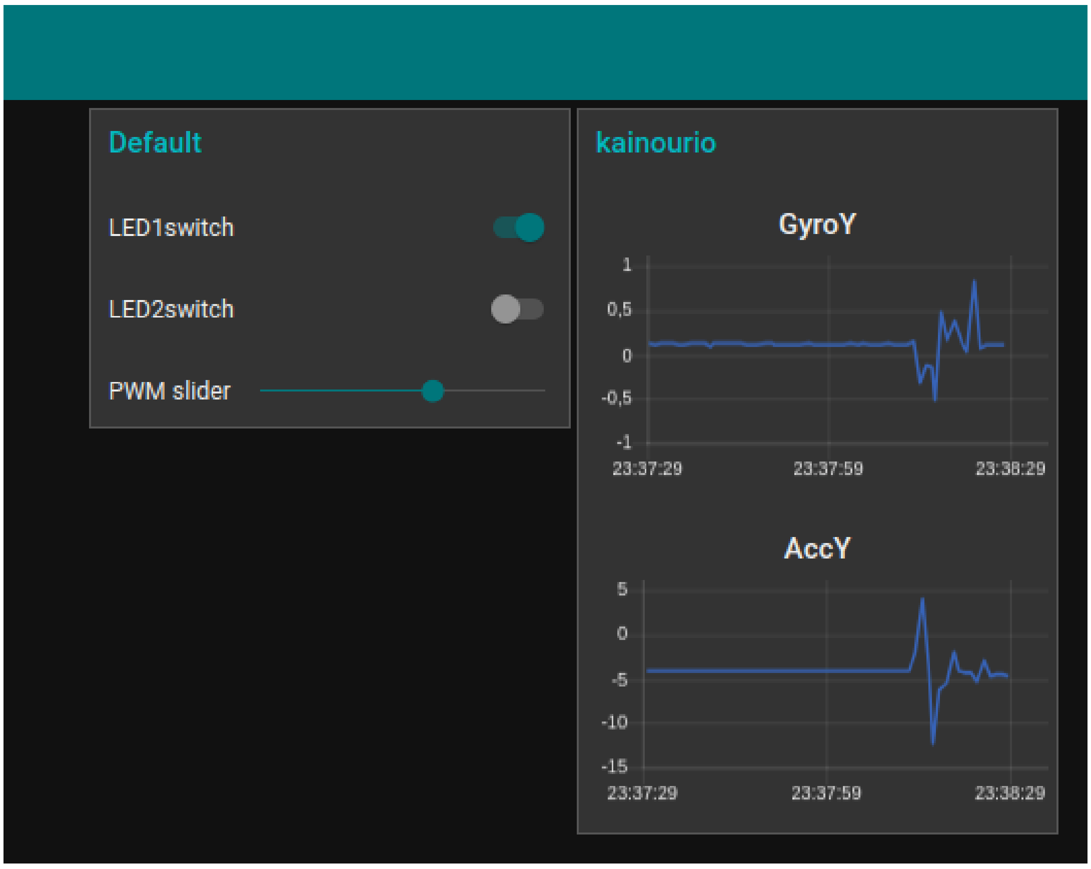


Figure 5: Node-Red UI.

5 Summary of Data Flow

Component	Topic / Action	Description
ESP32	Publish `esp32/restore_request`	Sends `get_latest` on startup
Node-RED	MQTT In on `esp32/restore_request`	Detects restore request
Node-RED	InfluxDB query	Fetch last saved LED/PWM states
Node-RED	Function node	Format restore state payload
Node-RED	MQTT Out on `esp32/restore_state`	Publishes restore state
ESP32	MQTT Subscribe `esp32/restore_state`	Receives and applies restore state
ESP32	Publish `esp32/status`	Periodic status updates

6 Motion detection via machine learning

In the second part, the goal is to move from the threshold supported movement detection and achieve the operational functionalities by employing machine learning. Communication with Node-Red remains the same, as the machine learning classifier is executing only in ESP32 having as a result the usage of the same topics and the same integration across ESP32-NodeRed.

6.1 Data acquisition

For data acquisition for each gesture, `mpu_record.py` have been utilized in order to send over UART a message "record" and then create samples for each of the two classifiers respectively. After manually we create the appropriate files to pass in the Python scripts to generate the C functions to be used in ESP32 and use the trained classifiers.

6.2 LEDs activation

The system in this configuration uses the readings from the MPU9250 IMU and machine learning to control the LEDs. For the data collection and feature processing the IMU continuously reads acceleration (ay) and gyroscope (gy) data along the Y-axis. These values are stored in an implemented circular buffer named featureBuffer (source: `CircularBuffer.h`) that maintains the 20 most recent samples(10 measurements of each Y axis of accelerometer and gyroscope)

When the buffer is full (20 samples), the features are processed for classification. The raw features are first scaled using the StandardScaler parameters which were generated on the `micro2.py` Python training script (scaler_mean and scaler_scale). The system then calculates Euclidean distances between the current scaled features and three reference scaled patterns (LED1, LED2, and Neutral) as a similarity measure. If the distance to the Neutral pattern is below 2.0, our implemented system considers it a neutral position and takes no action.

Otherwise if any of the euclidean distances between the current stored sample and the three classes(LED1 TOGGLE,LED2 TOGGLE and NEUTRAL), the features are passed to the SVM classifier (trained in `micro2.py`) which predicts whether the motion corresponds to LED1 toggle, LED2 toggle, or Neutral classes.

When a "LED1_TOGGLE" prediction is made, and at least 900ms have passed since the last toggle (debouncing method). The led1State boolean is flipped (true to false or false to true). As for the PWM duty cycle, it is set to pwmLevel variable if ON, or to 0 if OFF. Same logic as LED1, is followed for controlling "LED2_TOGGLE" prediction.

If confidence is below 75%, the system falls back into using just the distance metrics for classification. As an addition the 900ms delay between toggles is used in order to prevent rapid accidental toggling that can occur inside the same running sliding window as traversing the circular buffer of features. Also the confidence threshold helps for filtering out uncertain predictions.

The SVM classifier used for the LEDs activation and determining actions as LED1_TOGGLE, NEUTRAL and LED2_TOGGLE is getting as an input 20 features which consist of 10 samples of continuous values of (ay,gy) :
(*ay1,gy1,ay2,gy2,...ay10,gy10*).

6.3 Control of PWM

For controlling the common PWM applied in each LED in case it is "ON", we create a second classifier which works with data acquired from magnetometer X axis readings.

The SVM classifier used for determining the increase/decrease of the global used PWM value is getting as an input 9 features which consist of 9 samples of continuous values of (mx) :
(*mx1,mx2,...mx9*).

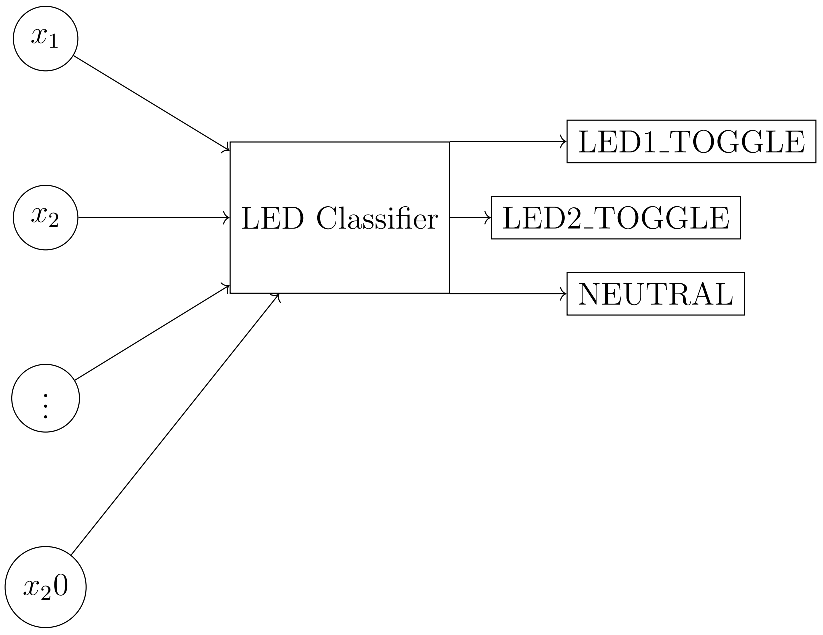


Figure 6: The SVM classifier used for the LED activation actions.

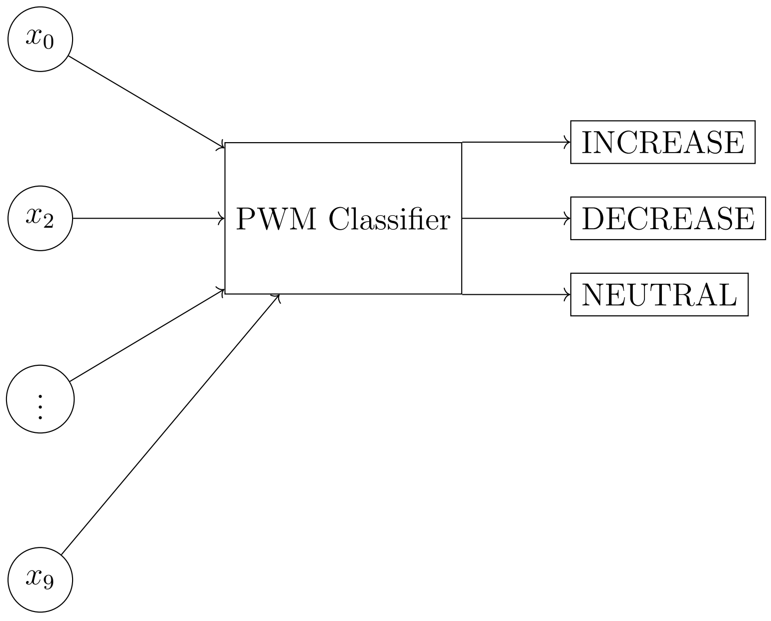


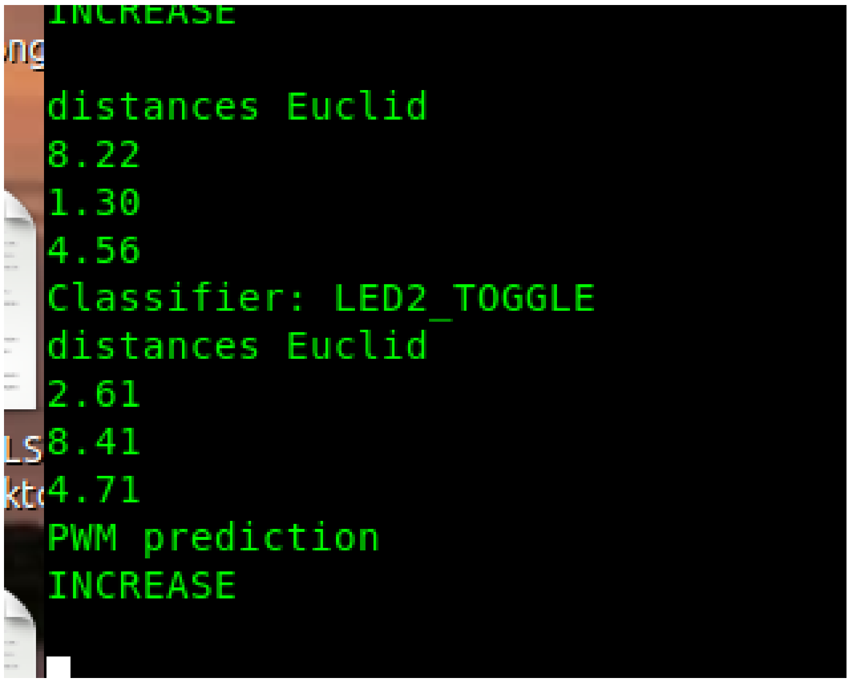
Figure 7: The SVM classifier used for the PWM actions.

6.3.1 Classifier's training and integration with the ESP32 platform.

Both of the SVM classifiers are trained in a local host with higher computational capabilities than ESP32, in order to achieve classification accuracy and then transfer in the embedded system only the libraries which include the weights of the trained classifier. After generating both of the classifiers their respective library files PWMClassifier.h and LEDClassifier.h are included in the source file which will be compiled and uploaded to the ESP32. In the source files the training for both is implemented in `micro2.py,micro3.py` respectively. Also in these files after training we do create one reference average sample per class which later is used for similarity comparison in the ESP32.

Why euclidean distance was chosen over cosine similarity: When detecting the class closest to a given point in feature space, Euclidean distance can be more appropriate than cosine similarity in situations where the absolute proximity of the vectors is more important than their directional alignment. While cosine similarity measures how aligned two vectors are (ignoring magnitude), Euclidean distance accounts for both the direction and the magnitude. This becomes crucial in our situation as well as shown experimentaly from the data, where cosine similarity was shown better similary to wrong gesture because the sample of the wrong class was an amplification of the correct's one. Its like in a graphical space when one class is slightly rotated or "hidden behind" another in angle they may have different orientations but still be physically close in space. In such cases, cosine similarity might consider them very different due to the angle, even though they are spatially close. Euclidean distance, in the other hand would correctly reflect their true closeness, making it better suited for identifying the nearest class based on actual position rather than orientation. For this reason in this project Euclidean distance has been utilized over the cosine similarity as a similarity measure.

A screenshot is presented also which shows both classifiers classifying the gesture as to increase the PWM which is applied on the LEDs and with the LED2 toggled. We can observe the classification outputs and also the Euclidean distances for each sample against the reference scaled average vectors who describe each class.

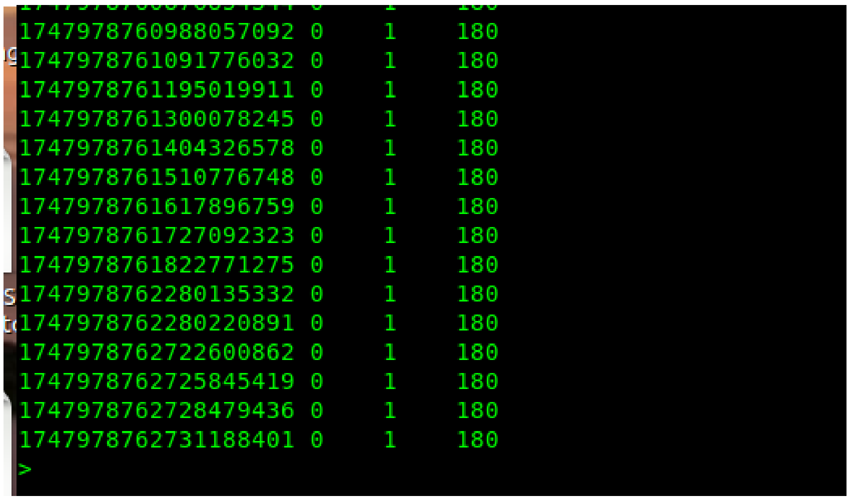


```
INCREASE

distances Euclid
8.22
1.30
4.56
Classifier: LED2_TOGGLE
distances Euclid
2.61
8.41
4.71
PWM prediction
INCREASE
```

Figure 8: classification screenshot.

From out terminal also we can observe the values inside our esp32db database in `led_status` measurements.c There is timestamp, led1 status(0-off 1-on), led2 status(same as led1) and the common pwm value.



```
1747978760988057092 0    1    180
1747978761091776032 0    1    180
1747978761195019911 0    1    180
1747978761300078245 0    1    180
1747978761404326578 0    1    180
1747978761510776748 0    1    180
1747978761617896759 0    1    180
1747978761727092323 0    1    180
1747978761822771275 0    1    180
1747978762280135332 0    1    180
1747978762280220891 0    1    180
1747978762722600862 0    1    180
1747978762725845419 0    1    180
1747978762728479436 0    1    180
1747978762731188401 0    1    180
>
```

Figure 9: Querying our esp32db: select * from led_status

6.4 Classifier selection

Although the support vector machine classifier has been selected and implemented for both of the classification tasks performed in the project, in the script `micro_explore.py` we have tested a range of classifiers and hyperparameters in order to obtain the broader picture of what they offer and how they behave on our data. Of course instead of the SVM we could have used any of these since the API calls from the generated header files follow the same style.

As we can observe utilizing our available data, SVM (using different kernel from the one we implemented and the Gaussian Naive Bayes classifiers) had 8% better F1-score than the SVM with the linear kernel that we have used.

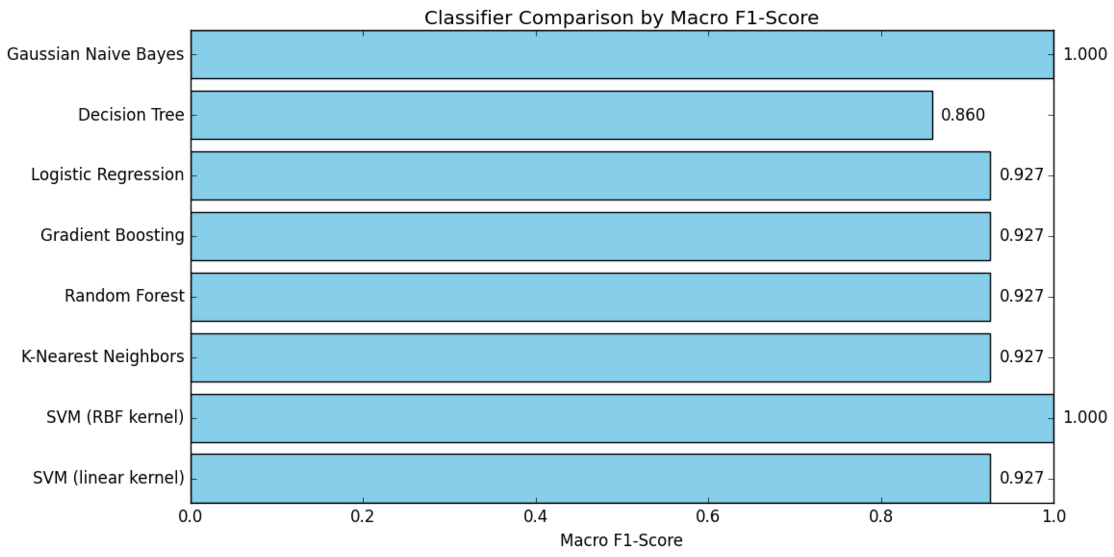


Figure 10: Classifier's space exploration.

7 Containerizing server side

Node-Red, InfluxDB and Grafana will be run as docker containers in the last part of this project.

In the first part, the creation of the `.yml` file

```
nicko@nicko:~/implementations/esp32/mpu_assignment/docker$ cat docker-compose.yml
version: '3'

services:
  node-red:
    image: nodered/node-red
    ports:
      - "1880:1880"
      - "18883:1883"   #different from mosquitto on host.
    volumes:
      - node-red-data:/data
    restart: unless-stopped
    depends_on:
      - influxdb

  influxdb:
    image: influxdb:1.6.4
    ports:
      - "8086:8086"
    volumes:
      - influxdb-data:/var/lib/influxdb
    environment:
      - INFLUXDB_DB=esp32db
      - INFLUXDB_ADMIN_ENABLED=true
    restart: unless-stopped

  grafana:
    image: grafana/grafana
    ports:
      - "3008:3000"      #different because of conflict on host.
    volumes:
      - grafana-data:/var/lib/grafana
    restart: unless-stopped
    depends_on:
      - influxdb

volumes:
  node-red-data:
  influxdb-data:
  grafana-data:
```

is used in order to pass the information of what container we aim to generate and

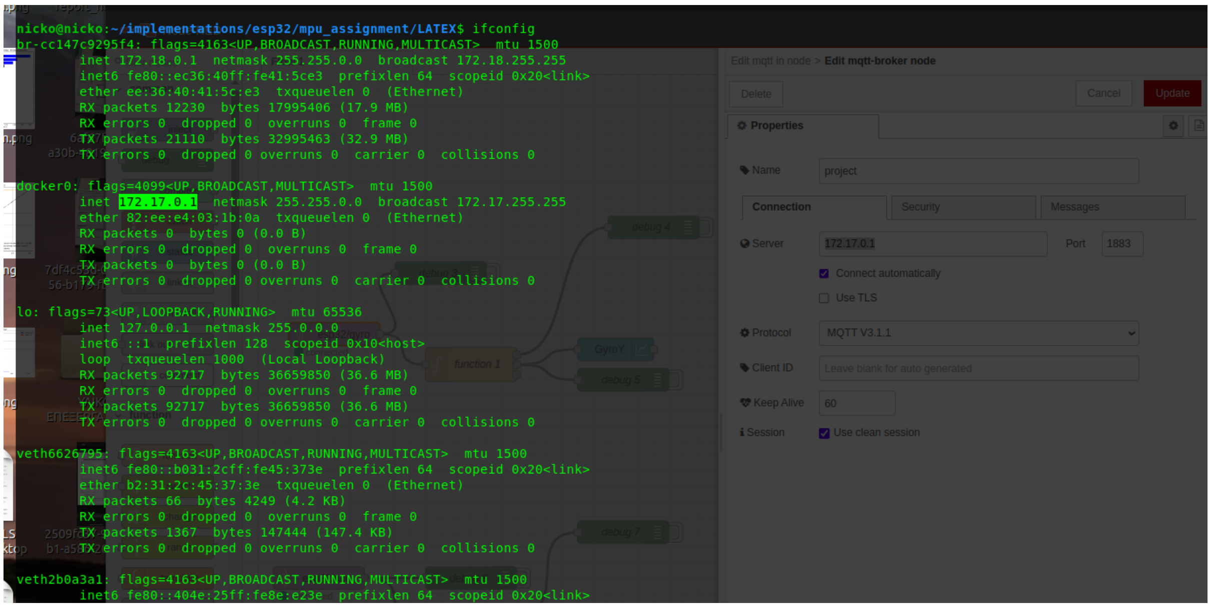
```
docker-compose up -d
```

completes this action.

To run influxdb queries and inspection of the dataset:

```
docker exec -it docker-influxdb-1 influx
```

In Node-Red when configuring the MQTT in/outs, we need to pass the "Server" IP as Linux "see"s it on an `ifconfig` command as can be see on the following figure.



```
nicko@nicko:~/implementations/esp32/mpu_assignment/LATEX$ ifconfig
br-cc147c9295f4: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.18.0.1  netmask 255.255.0.0  broadcast 172.18.255.255
        inet6 fe80::ec36:40ff:fe41:5ce3  prefixlen 64  scopeid 0x20<link>
        ether ee:36:40:41:5c:e3  txqueuelen 0  (Ethernet)
        RX packets 12230  bytes 17995406 (17.9 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 21110  bytes 32995463 (32.9 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 82:ee:e4:03:1b:0a  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 92717  bytes 36659850 (36.6 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 92717  bytes 36659850 (36.6 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

veth6626795: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::b031:2cff:fe45:373e  prefixlen 64  scopeid 0x20<link>
        ether b2:31:2c:45:37:3e  txqueuelen 0  (Ethernet)
        RX packets 66  bytes 4249 (4.2 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1367  bytes 147444 (147.4 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

veth2b0a3a1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::404e:25ff:fe8e:e23e  prefixlen 64  scopeid 0x20<link>
```

Edit mqtt in node > **Edit mqtt-broker node**

Properties

Name: project

Connection | Security | Messages

Server: 172.17.0.1 Port: 1883

[✓] Connect automatically

[ ] Use TLS

Protocol: MQTT V3.1.1

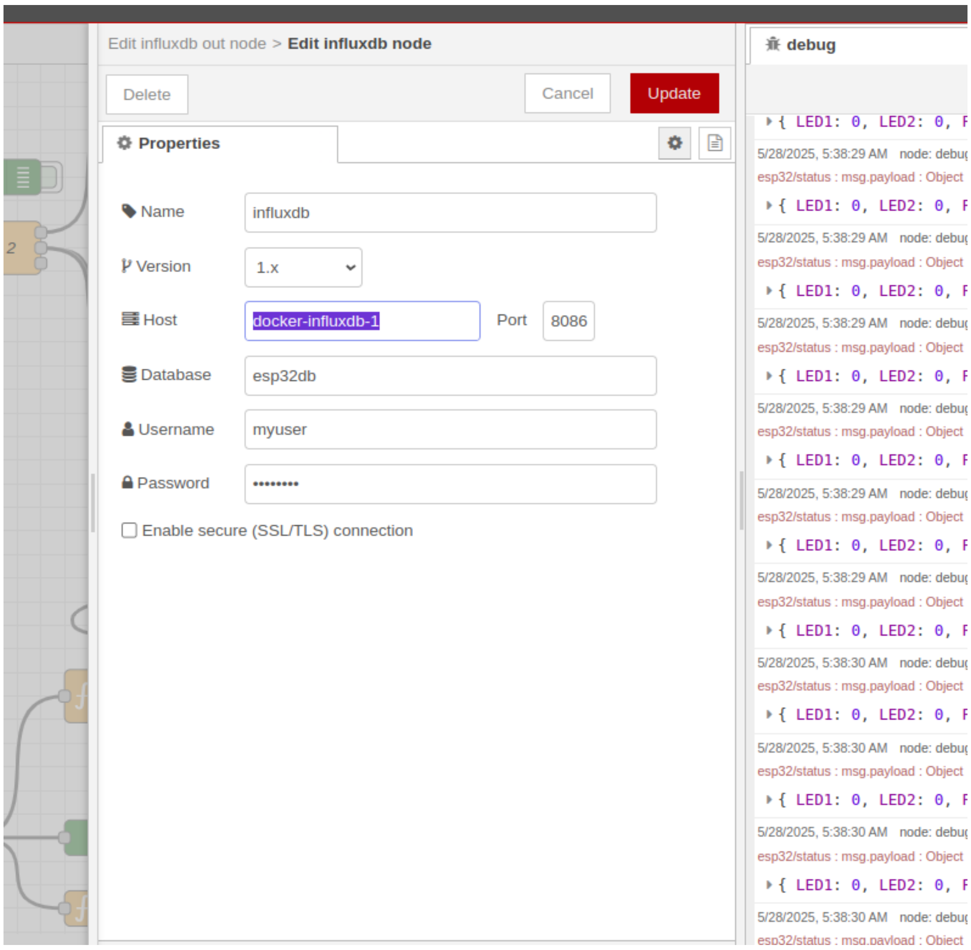
Client ID: Leave blank for auto generated

Keep Alive: 60

Session: [✓] Use clean session

Figure 11: Setting server ip for MQTT nodes

For the configuration of InfluxDB nodes inside the Node-Red, on the "Server" IP, we need to pass the name of the docker container which in our case is `docker-influxdb-1`.



Edit influxdb out node > **Edit influxdb node**

Properties

Name: influxdb

Version: 1.x

Host: docker-influxdb-1 Port: 8086

Database: esp32db

Username: myuser

Password: ••••••••

[ ] Enable secure (SSL/TLS) connection

```
▸{ LED1: 0, LED2: 0, F
5/28/2025, 5:38:29 AM  node: debu
esp32/status : msg.payload : Object
▸{ LED1: 0, LED2: 0, F
5/28/2025, 5:38:29 AM  node: debu
esp32/status : msg.payload : Object
▸{ LED1: 0, LED2: 0, F
5/28/2025, 5:38:29 AM  node: debu
esp32/status : msg.payload : Object
▸{ LED1: 0, LED2: 0, F
5/28/2025, 5:38:29 AM  node: debu
esp32/status : msg.payload : Object
▸{ LED1: 0, LED2: 0, F
5/28/2025, 5:38:29 AM  node: debu
esp32/status : msg.payload : Object
▸{ LED1: 0, LED2: 0, F
5/28/2025, 5:38:29 AM  node: debu
esp32/status : msg.payload : Object
▸{ LED1: 0, LED2: 0, F
5/28/2025, 5:38:30 AM  node: debu
esp32/status : msg.payload : Object
▸{ LED1: 0, LED2: 0, F
5/28/2025, 5:38:30 AM  node: debu
esp32/status : msg.payload : Object
▸{ LED1: 0, LED2: 0, F
5/28/2025, 5:38:30 AM  node: debu
esp32/status : msg.payload : Object
▸{ LED1: 0, LED2: 0, F
5/28/2025, 5:38:30 AM  node: debu
esp32/status : msg.payload : Object
```

Figure 12: Setting server ip for InfluxDB nodes.

For the password previously we assigned via command line on `influx-cli`:

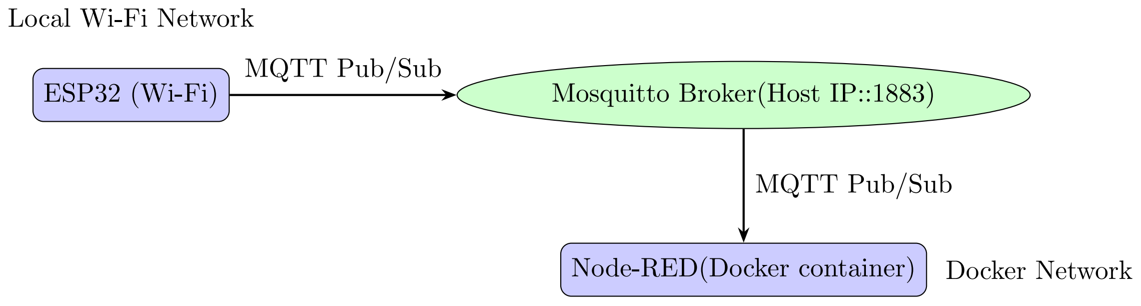
```
CREATE USER "myuser" WITH PASSWORD 'mypassword'
GRANT ALL ON "esp32db" TO "myuser"
```

By completing those configurations we have the same functionality as before by running the two server services in separate Docker containers, without the need to modify anything on the source code of the ESP32.

7.1 Current MQTT communication architecture

The ESP32 device connects to the local Wi-Fi network using the credentials `ssid` and `password`. It establishes an MQTT client connection to the Mosquitto broker running on the host machine at its IP address and port `1883`. Meanwhile, the Node-RED service is containerized within the Docker and also connects to the same Mosquitto broker on the host, using the Docker host IP address to communicate with the broker as shown in the upcoming figure.

All MQTT messages published by the ESP32 are received by the Mosquitto broker and then routed to Node-RED flows subscribed to the respective topics. This setup enables the communication between the hardware and the Node-RED environment through a shared MQTT broker.


Local Wi-Fi Network
ESP32 (Wi-Fi)
MQTT Pub/Sub
Mosquitto Broker(Host IP::1883)
MQTT Pub/Sub
Node-RED(Docker container)
Docker Network