Vietnamese – German University

DEPARTMENT OF COMPUTER SCIENCE

Project's Report:

"AIR QUALITY MONITORING ENVIRONMENTAL SENSING FARM"

Lecturer: Vo Bich Hien

Student – ID:

Nguyen Duc Trung – 10423116

Nguyen Ngoc Binh Son – 10423100

Tram Dinh Khoa – 10423059

Binh Duong – 05/2025

Vietnamese-German University

Contents

1. Introduction

Currently, monitoring environmental parameters such as temperature, humidity, and air quality plays an important role in managing and protecting living spaces. Thanks to the development of IoT (Internet of Things) technology, sensor systems can be deployed to collect environmental data in a flexible, accurate, and real-time manner.

The "Environmental Sensing Farm" model is designed to monitor key environmental factors such as temperature, humidity, and air pollution levels (PM2.5, CO_2, etc.). Sensors integrated into the IoT system allow real-time data collection and transmission to a server or cloud platform for processing, visualization, and timely alerts.

The application of this model serves not only research purposes but can also be implemented in smart agriculture, smart cities, and residential areas to improve quality of life and reduce the negative impacts of environmental factors on human health.

2. Methodology

a. Pipeline overview

The data processing and machine learning workflow follows a structured pipeline from sensor data acquisition to model deployment. The pipeline begins with real-time data collection using ESP32 microcontrollers equipped with environmental sensors. The collected data is then preprocessed and transmitted to a server or cloud platform, where it is used for training and testing machine learning models. The end goal is to deploy predictive models capable of detecting or predicting environmental conditions such as air quality.

b. Data collection

Environmental data such as temperature, humidity, PM2.5, and CO_2 levels are collected using an array of IoT sensors connected to the ESP32 development board. These sensors continuously monitor the ambient environment and send data to a centralized storage system via wireless transmission. This raw data is essential for both model training and real-time environmental monitoring.

c. Code Availability

All source code for sensor control, data transmission, and model integration is maintained in a version-controlled repository. The repository includes Arduino scripts for the ESP32, Python scripts for model training, and integration scripts for interfacing with cloud services. This ensures reproducibility and facilitates future modifications or extensions of the system.

d. Data Preprocessing

Prior to training the machine learning model, the raw data undergoes several preprocessing steps, including noise reduction, normalization, and segmentation. These steps are crucial to ensure data quality and improve model performance.

i. Implementation in Edge Impulse

Edge Impulse, a platform tailored for embedded machine learning, is utilized to facilitate the preprocessing and model training processes. Sensor data is uploaded to the Edge Impulse studio, where it is processed using built-in tools for filtering, spectral analysis, and feature extraction. This implementation allows seamless integration with the ESP32 hardware and supports deployment of the final model directly onto the edge device.

3. Model Design

a. Model Design

The core objective of the model design is to develop a lightweight, efficient machine learning model that can run on edge devices with limited computational power. The model is tailored to classify air quality levels or predict future environmental changes based on real-time sensor inputs. Given the resource constraints, a decision was made to use classification or regression models with low latency and memory footprint.

b. Model Development

The model development process includes selecting appropriate algorithms, training the models on historical sensor data, and validating their performance. A variety of models such as decision trees, logistic regression, and small-scale neural networks were explored. These models were trained and evaluated using the Edge Impulse platform, which provides built-in tools for testing accuracy, confusion matrices, and real-time performance evaluation. After tuning and validation, the optimal model was deployed back to the ESP32 for real-time inference.

4. What we have done/ evidence

a. Esp – 32

```
// ==== Server ====
WiFiServer server(80);

// ==== Connect to Wi-Fi ====
bool connectToWiFi(int timeoutPerNetwork = 7000) {
  preferences.begin("wifi", true);
  lastSSID = preferences.getString("ssid", "");
  lastPass = preferences.getString("pass", "");
  preferences.end();

  Serial.print("Trying saved WiFi: ");
  Serial.println(lastSSID);

  if (lastSSID != "") {
    WiFi.begin(lastSSID.c_str(), lastPass.c_str());
    unsigned long start = millis();
    while (WiFi.status() != WL_CONNECTED && millis() - start <
timeoutPerNetwork) delay(250);
    if (WiFi.status() == WL_CONNECTED) return true;
```

```
  }

  for (int i = 0; i < wifiCount; i++) {
    Serial.print("Trying: ");
    Serial.println(wifiList[i][0]);

    WiFi.begin(wifiList[i][0], wifiList[i][1]);
    unsigned long start = millis();
    while (WiFi.status() != WL_CONNECTED && millis() - start <
timeoutPerNetwork) delay(250);
    if (WiFi.status() == WL_CONNECTED) {
      preferences.begin("wifi", false);
      preferences.putString("ssid", wifiList[i][0]);
      preferences.putString("pass", wifiList[i][1]);
      preferences.end();
      return true;
    }
  }

  return false;
}
// ==== Setup ====
void setup() {
  Serial.begin(115200);

  wifiConnected = connectToWiFi();
  if (wifiConnected) {
    Serial.print("Connected to WiFi. IP: ");
    Serial.println(WiFi.localIP());
    server.begin();
  } else {
    Serial.println("WiFi connection failed.");
  }

  dht.begin();
  pinMode(LED, OUTPUT);
  pinMode(BUZZER, OUTPUT);

  pmsSerial.begin(9600, SERIAL_8N1, PMS_RX, PMS_TX);
  pms.passiveMode();
  pms.wakeUp();


}
```

```
// ==== Main Loop ====
void loop() {
  float temperature = dht.readTemperature();
  float humidity = dht.readHumidity();

  WiFiClient client = server.available();
  if (client) {
    Serial.println("Client connected");
    String request = client.readStringUntil('\r');
    client.flush();
    Serial.println(request);

    if (request.indexOf("GET /data") >= 0) {
      String json = String("{") +
        "\"temperature\":" + String(temperature, 2) + "," +
        "\"humidity\":" + String(humidity, 2) + "," +
        "\"pm1_0\":" + String(pmsData.PM_AE_UG_1_0) + "," +
        "\"pm2_5\":" + String(pmsData.PM_AE_UG_2_5) + "," +
        "\"pm10\":" + String(pmsData.PM_AE_UG_10_0) +
      "}";

// send json to the app
      client.println("HTTP/1.1 200 OK");
      client.println("Content-Type: application/json");
      client.println("Access-Control-Allow-Origin: *");
      client.println("Connection: close");
      client.println();
      client.println(json);

      Serial.println("Sent JSON:");
      Serial.println(json);
    }

    delay(5);
    client.stop();
    Serial.println("Client disconnected");
  }
```

b. Training small machine predict

```
# === 3. Build Model ===
def build_model(input_dim):
    model = Sequential([
        Dense(128, activation='relu', input_shape=(input_dim,)),
        Dropout(0.2),
        Dense(64, activation='relu'),
        Dropout(0.2),
```

```
        Dense(2)    # Output: next temp & humidity
    ])
    model.compile(optimizer='adam', loss='mse', metrics=['mae'])
    return model

# === 4. Train Model ===
def train_model(model, X_train, y_train):
    early_stop = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
    checkpoint = ModelCheckpoint("best_model.keras", save_best_only=True)

    history = model.fit(
        X_train, y_train,
        validation_split=0.2,
        epochs=200,
        batch_size=32,
        callbacks=[early_stop, checkpoint],
        verbose=1
    )
    return history

# === 5. Evaluate & Visualize ===
def evaluate_model(model, X_test, y_test, scaler_y):
    y_pred_scaled = model.predict(X_test)
    y_pred = scaler_y.inverse_transform(y_pred_scaled)
    y_true = scaler_y.inverse_transform(y_test)

    # Metrics
    rmse_temp = mean_squared_error(y_true[:, 0], y_pred[:, 0], squared=False)
    mae_temp = mean_absolute_error(y_true[:, 0], y_pred[:, 0])
    rmse_humid = mean_squared_error(y_true[:, 1], y_pred[:, 1], squared=False)
    mae_humid = mean_absolute_error(y_true[:, 1], y_pred[:, 1])

    print(f"\nTemperature → RMSE: {rmse_temp:.2f}, MAE: {mae_temp:.2f}")
    print(f"Humidity    → RMSE: {rmse_humid:.2f}, MAE: {mae_humid:.2f}")

    return y_true, y_pred

def plot_history(history):
    plt.figure(figsize=(12, 5))

    plt.subplot(1, 2, 1)
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Val Loss')
    plt.title("Loss (MSE)")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend()

    plt.subplot(1, 2, 2)
```

```
        plt.plot(history.history['mae'], label='Train MAE')
        plt.plot(history.history['val_mae'], label='Val MAE')
        plt.title("Mean Absolute Error")
        plt.xlabel("Epoch")
        plt.ylabel("MAE")
        plt.legend()

        plt.tight_layout()
        plt.show()

def plot_predictions(y_true, y_pred):
        plt.figure(figsize=(12, 5))

        plt.subplot(1, 2, 1)
        plt.plot(y_true[:100, 0], label='True Temp')
        plt.plot(y_pred[:100, 0], label='Pred Temp')
        plt.title("Temperature Prediction")
        plt.xlabel("Sample")
        plt.ylabel("Temp (°C)")
        plt.legend()

        plt.subplot(1, 2, 2)
        plt.plot(y_true[:100, 1], label='True Humid')
        plt.plot(y_pred[:100, 1], label='Pred Humid')
        plt.title("Humidity Prediction")
        plt.xlabel("Sample")
        plt.ylabel("Humidity (%)")
        plt.legend()

        plt.tight_layout()
        plt.show()

def plot_confusion_matrix(y_true, y_pred):
        true_bins = pd.cut(y_true[:, 0], bins=10)
        pred_bins = pd.cut(y_pred[:, 0], bins=true_bins.categories)

        cm = pd.crosstab(true_bins, pred_bins, rownames=["Actual Temp"], colnames=["Predicte
d Temp"])
        plt.figure(figsize=(10, 8))
        sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
        plt.title("Binned Confusion Matrix: Temperature")
        plt.show()
```

c. Code Air monitor to use the data sent by ESP – 32

UI components and a database helper are declared. Define environmental ranges. Default fallback ESP32 data URL.

```
class MainActivity : AppCompatActivity() {
    private lateinit var temperatureText: TextView
    private lateinit var humidityText: TextView
    private lateinit var faceView: FaceView
    private lateinit var dbHelper: SensorDatabaseHelper
    private lateinit var stateText: TextView
    private var lastNotifiedAirState: String? = null

    private val minTemp = 20.0
    private val maxTemp = 34.0
    private val minHumid = 30.0
    private val maxHumid = 80.0

    @Volatile
    private var esp32Url = "http://192.168.2.5/data" // fallback IP

    private val channelId = "air_monitor_alerts"
    private val notificationId = 1
```

Application of those code:

CreateNotificationChannel() – for Android notifications
StartBackgroundService() – to collect sensor data
StartListeningForESP32() – UDP listener for ESP32 broadcasts
FetchSensorData() – likely loads current data

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    temperatureText = findViewById(R.id.temperatureText)
    humidityText = findViewById(R.id.humidityText)
    faceView = findViewById(R.id.FaceView)
    stateText = findViewById(R.id.stateText)

    dbHelper = SensorDatabaseHelper(this)

    createNotificationChannel()
    startBackgroundService()
    startListeningForESP32()
    fetchSensorData()

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU &&
        ActivityCompat.checkSelfPermission(this,
```

```
Manifest.permission.POST_NOTIFICATIONS) != PackageManager.PERMISSION_GRANTED
    ) {
        ActivityCompat.requestPermissions(this,
arrayOf(Manifest.permission.POST_NOTIFICATIONS), 100)
    }
```

Creates a UDP socket to listen on port 4210 and update the weather data val name = "Air Quality Alerts" - launches SensorService. - sets up a notification channel:

```
private fun startListeningForESP32() {
    CoroutineScope(Dispatchers.IO).launch {
        try {
            val socket = DatagramSocket(4210,
InetAddress.getByName("0.0.0.0"))
            socket.broadcast = true
            val buffer = ByteArray(1024)

            while (true) {
                val packet = DatagramPacket(buffer, buffer.size)
                socket.receive(packet)
                val message = String(packet.data, 0, packet.length)
                if (message.startsWith("ESP32_WEATHER|")) {
                    val ip = message.split("|")[1]
                    esp32Url = "http://$ip/data"
                    Log.d("UDP", "ESP32 discovered at: $esp32Url")
                }
            }
        } catch (e: Exception) {
            e.printStackTrace()
        }
    }
}
    private fun evaluateEnvironment(temp: Double, humidity: Double, pm25:
Double): Pair<FaceView.Emotion, String> {
        // Determine air state from PM2.5
        val airState = when {
            pm25 < 0 -> "Invalid" // Optional: handle negative or impossible
values
            pm25 <= 12.0 -> "Good"
            pm25 > 12.0 && pm25 <= 35.4 -> "Moderate"
            pm25 > 35.4 && pm25 <= 55.4 -> "Unhealthy for Sensitive Groups"
            pm25 > 55.4 && pm25 <= 150.4 -> "Unhealthy"
            pm25 > 150.4 && pm25 <= 250.4 -> "Very Unhealthy"
```

```
            pm25 > 250.4 -> "Hazardous"
            else -> "Unknown"
        }

        // Evaluate emotion based on temperature and humidity
        val tempOutOfRange = temp < minTemp || temp > maxTemp
        val humidOutOfRange = humidity < minHumid || humidity > maxHumid

        val tempNearEdge = temp in (minTemp..(minTemp + 1)) || temp in
((maxTemp - 1)..maxTemp)
        val humidNearEdge = humidity in (minHumid..(minHumid + 5)) ||
humidity in ((maxHumid - 5)..maxHumid)

        val emotion = when {
            // Air quality overrides emotions for unhealthy levels:
            airState == "Very Unhealthy" || airState == "Hazardous" ->
FaceView.Emotion.ANGRY
            airState == "Unhealthy" || airState == "Unhealthy for Sensitive
Groups" -> FaceView.Emotion.SAD

            // Temperature & humidity affect emotion only if air quality is
good/moderate:
            tempOutOfRange -> FaceView.Emotion.ANGRY
            humidOutOfRange -> FaceView.Emotion.SAD
            tempNearEdge || humidNearEdge -> FaceView.Emotion.NEUTRAL
            else -> FaceView.Emotion.HAPPY
        }

        return Pair(emotion, airState)
    }


    private fun getAirQualityMessage(airState: String): String {
        return when (airState) {
            "Unhealthy for Sensitive Groups" -> "You might suffer if you have
breathing issues."
            "Unhealthy" -> "I fear nothing, but this air, it scares me"
            "Very Unhealthy" -> "Your lungs are probably crying, wear a
mask!"
            "Hazardous" -> "Dangerous air quality, keep breathing this air
will shorten your lifespan"
            else -> "Air quality is okay."  // Should not be shown in noti
        }
```

```
    }


    private fun sendNotification(airState: String) {
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.TIRAMISU &&
            ActivityCompat.checkSelfPermission(this,
Manifest.permission.POST_NOTIFICATIONS) != PackageManager.PERMISSION_GRANTED
        ) {
            Log.w("Notification", "POST_NOTIFICATIONS permission not
granted.")
            return
        }

        val title = "Air Quality Alert: $airState"

        val message = getAirQualityMessage(airState)

        val builder = NotificationCompat.Builder(this, channelId)
            .setSmallIcon(R.mipmap.ic_launcher)
            .setContentTitle(title)
            .setContentText(message)
            .setPriority(NotificationCompat.PRIORITY_HIGH)

        NotificationManagerCompat.from(this).notify(notificationId,
builder.build())
    }
}
```

5. Implementation

a. Tools and Frameworks used

To ensure a robust and scalable system, a variety of tools and frameworks were employed throughout the project. Edge Impulse was the primary machine learning platform used for model development, data preprocessing, and deployment. The Arduino IDE served as the main development environment for programming the ESP32 microcontrollers. Additional libraries and frameworks included:

Arduino Libraries: For sensor communication and data handling (e.g., DHT, PMS, PM sensors).

Edge Impulse CLI: For data upload and deployment.

Serial Monitor and PlatformIO: For debugging and testing embedded code.

b. Hardware Decision

The hardware was chosen to balance cost, power efficiency, and functionality. The ESP32 microcontroller was selected due to its integrated Wi-Fi, Bluetooth, and sufficient processing capabilities for edge ML tasks. The sensor suite included:

- **DHT22** for temperature and humidity,
- **PMS5003** for measures the density of dirt using laser sensor,
- **PM2.5 Sensor** for particulate matter,
- **LED, buzzer, and LCD** modules for real-time feedback and alerts.

This combination provides a comprehensive environmental monitoring system capable of real-time sensing and interaction.

c. Arduino Library and Hardware Testing with files

All components were integrated using well-established Arduino libraries. Testing procedures included unit tests for individual sensors and integration tests to verify multi-sensor operation. Code files were structured and commented for clarity, with test logs recorded during hardware validation. Data from sensors were cross-checked with environmental baselines to ensure accuracy.

6. Results

a. Model training

```
# Load and process data
    df = load_and_preprocess("/kaggle/input/temp-and-hum-dataset/sensordata.csv")
    X_train, X_test, y_train, y_test, scaler_x, scaler_y = prepare_data(df)

    # Build and train model
    model = build_model(X_train.shape[1])
    history = train_model(model, X_train, y_train)

    # Evaluate
    loss, mae = model.evaluate(X_test, y_test)
    print(f"\nTest MAE (scaled): {mae:.4f}")

    y_true, y_pred = evaluate_model(model, X_test, y_test, scaler_y)

    # Visualizations
    plot_history(history)
    plot_predictions(y_true, y_pred)
    plot_confusion_matrix(y_true, y_pred)

    # Export TFLite model
    export_tflite_model(model)
```

b. Model testing

The trained model was validated using a separate testing dataset. Performance was evaluated using standard metrics such as accuracy, precision, and recall. Edge Impulse's confusion matrix tool was utilized to visualize classification results and assess misclassifications.

c. Evaluation

The final model demonstrated reliable performance with high accuracy in classifying air quality levels. Evaluation on-device confirmed that the model could perform real-time inference without latency issues. The system proved robust under varied environmental conditions during field testing.

7. Discussion

a. Model Performance and Efficieny

The deployed model performed with over 90% accuracy in controlled conditions and retained solid performance in field environments. Its lightweight structure ensured efficient operation on the ESP32, with low memory and power consumption, aligning with edge computing best practices.

b. Challenge

Key challenges included: Noise in sensor data requiring extensive filtering. Memory constraints on the ESP32, limiting model complexity. Environmental variability affecting sensor reliability, especially in outdoor conditions.

c. Environmental and Cost Benefits

The system promotes low-cost environmental monitoring, supporting sustainability by encouraging early detection of pollution. Its modular and reusable design makes it ideal for scalable deployment in agriculture, smart cities, and residential applications.

d. Comparing MQ135 and PMS5003

In summary, the MQ135 and PMS5003 are both air quality sensors but differ significantly in operation, efficiency, and purpose. The MQ135 operates based on a chemical sensing mechanism that detects a range of gases like NH_3, NO_x, alcohol, benzene, and CO_2, making it suitable for general indoor air quality monitoring. However, it is less accurate and requires calibration, which affects its efficiency. On the other hand, the PMS5003 uses laser-based light scattering to provide precise and real-time measurements of particulate matter (PM1.0, PM2.5, and PM10), offering higher reliability and sensitivity, especially for detecting fine dust. Therefore, while MQ135 is more versatile in detecting gas pollutants, the PMS5003 is more effective and accurate for particulate monitoring in environments that require detailed air quality data.

8. Conclusion

a. Summary of findings

The project successfully designed, implemented, and evaluated an IoT-based air quality monitoring system using embedded machine learning. It demonstrated the feasibility of deploying smart sensing systems in real-time applications, supported by efficient hardware and software integration.

b. Potential Impact

This system has the potential to significantly contribute to environmental monitoring efforts, especially in underserved or rural areas. Its adaptability and cost-efficiency make it a strong candidate for broader implementation in smart farming, public health monitoring, and environmental protection initiatives.