# ML-based Distributed Decision Making with Convergent Replicated Data Types

Project-I (CS57003) report submitted to

Indian Institute of Technology Kharagpur

in partial fulfilment for the award of the degree of

**Master of Technology**

**in**

**Computer Science and Engineering**

by

**Amit Kumar**

**(20CS30003)**

Under the supervision of

**Dr. Sandip Chakraborty**


**Department of Computer Science and Engineering**

**Indian Institute of Technology Kharagpur**

**Autumn Semester, 2024-25**

**November 7, 2024**

# DECLARATION

I certify that

(a) The work contained in this report has been done by me under the guidance of my supervisor.

(b) The work has not been submitted to any other Institute for any degree or diploma.

(c) I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.

(d) Whenever I have used materials (data, theoretical analysis, figures, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references. Further, I have taken permission from the copyright owners of the sources, whenever necessary.

Date: November 7, 2024 (Amit Kumar)

Place: Kharagpur (20CS30003)

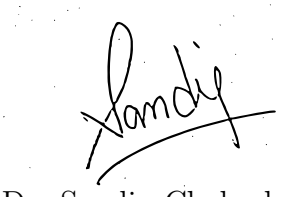
**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR**

**KHARAGPUR - 721302, INDIA**


## *CERTIFICATE*

This is to certify that the project report entitled "**ML-based Distributed Decision Making with Convergent Replicated Data Types**" submitted by **Amit Kumar** (Roll No. 20CS30003) to Indian Institute of Technology Kharagpur towards partial fulfilment of requirements for the degree of Master of Technology in Computer Science and Engineering is a record of bonafide work carried out by him under my supervision and guidance during Autumn Semester, 2024-25.



Dr. Sandip Chakraborty
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur

Date: November 7, 2024
Place: Kharagpur

# Acknowledgements

While bringing out this report to its final form, I came across a number of people whose contributions in various ways helped my field of research and they deserve special thanks. It is a pleasure to convey my gratitude to all of them. First and foremost, I would like to express my deep sense of gratitude and indebtedness to my supervisor Prof. Sandip Chakraborty for his invaluable encouragement, suggestions and support from an early stage of this research and providing me extraordinary experiences throughout the work. Above all, his priceless and meticulous supervision at each and every phase of work inspired me in innumerable ways. There involvement with originality has triggered and nourished my intellectual maturity that will help me for a long time to come.

I specially acknowledge my PhD mentor Prasenjit Karmakar for their advice, super- vision, and the vital contribution as and when required during this research. I am highly grateful that I had the opportunity to work such talented mentors and for their support and co-operation that is hard to express.

Date: November 7, 2024 (Amit Kumar)
Place: Kharagpur (20CS30003)

# Abstract

Name of the student: **Amit Kumar** Roll No: **20CS30003**
Degree for which submitted: **Master of Technology**
Department: **Department of Computer Science and Engineering**
Title: **ML-based Distributed Decision Making with Convergent Replicated Data Types**
Supervisor: **Dr. Sandip Chakraborty**
Month and year of thesis submission: **November 7, 2024**

Indoor air quality monitoring systems have traditionally operated in isolation, providing localized information about specific spaces. This limitation prevents the development of a comprehensive understanding of air quality dynamics across entire indoor environments. Our research presents a novel distributed system architecture leveraging multiple ESP32-based air quality sensors integrated through Set CVRDT (Conflict-free Replicated Data Type) for maintaining a global environmental context.

The system implements the Raft consensus algorithm for leader election and robust distributed coordination among sensors, enabling reliable information sharing across different locations within a household. By analyzing data from multiple sensors simultaneously, our system can track the propagation of air quality events (such as cooking fumes or potential fire hazards) throughout the building, facilitating timely interventions and automated responses.

We evaluated our approach using two comprehensive datasets: one focusing on academic settings across eight indoor activities, and another comprising cooking type and food measurements from a kitchen in a suburban food canteen in India. Our implementation includes the deployment of various machine learning models, converted to C code for ESP32 execution, demonstrating the feasibility of edge computing for real-time air quality analysis and event detection.

Our distributed approach enables more effective environmental monitoring compared to traditional isolated sensors, allowing for proactive measures based on a holistic understanding of indoor air quality dynamics. This system has significant implications for improving indoor air quality management and occupant safety through automated, context-aware environmental control.

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| **CO2** | **C**arbon Di**o**xide |
| **DALTON** | **D**istributed **A**ir Qua**L**i**T**y M**ON**itor |
| **ESP-32** | **E**spinif Microcontroller with integrated **S**eries **P**rocessor-32 bit |
| **PM10** | **P**articulate **M**atter (10 micrometers or less in diameter) |
| **PM2.5** | **P**articulate **M**atter (2.5 micrometers or less in diameter) |
| **ML** | **M**achine **L**earning |
| **VOC** | **V**olatile **O**rganic **C**ompounds |
| **WHO** | **W**orld **H**ealth **O**rganization |

# Chapter 1

# Introduction

## 1.1 Motivation

Indoor air quality has emerged as a critical concern in modern society, with poor air quality contributing to over 3.2 million deaths(2020, WHO) annually worldwide. This issue is particularly acute in developing nations like India, where inadequate regulation, limited awareness, and severe outdoor pollution significantly impact indoor air quality. The proliferation of low-cost air quality monitors in indoor spaces, while beneficial for individual awareness, has primarily focused on isolated measurements that fail to capture the complex dynamics of air pollution propagation across interconnected indoor spaces.

Recent studies[16] have demonstrated the strong correlation between human activities and indoor air quality fluctuations. For instance, research in academic settings has achieved 97.7% accuracy in classifying eight distinct indoor activities through air quality measurements alone. Furthermore, extensive data collection[17] across 30 indoor sites in India, spanning rural, suburban, and urban regions, has revealed

unique pollution patterns influenced by various daily activities. However, these insights remain confined to localized measurements, limiting their practical utility in comprehensive indoor environment management.

The potential applications of this distributed monitoring network are broad and impactful. Beyond real-time activity recognition and pollutant response, this system lays the foundation for improved air management strategies within households. For example, in the event of a fire, early pollutant propagation detection across sensors could trigger timely alerts and preventive actions. By providing occupants with actionable insights on ventilation and air quality, our system fosters healthier indoor environments and advances the capabilities of affordable air monitoring systems. The decentralized architecture ensures scalability and robustness, making it adaptable for various indoor settings, from residences to workplaces.

The challenge lies not merely in detecting air quality variations but in understanding how these changes propagate through connected indoor spaces and affect the overall indoor environment. For example, cooking activities in a kitchen can impact air quality throughout a household, yet current monitoring systems lack the capability to track such propagation patterns or provide timely, coordinated responses. This limitation significantly hampers the effectiveness of air quality management strategies and potentially compromises occupant safety during critical events such as fire hazards.

## 1.2 Solution

To address these challenges, we present a novel distributed system architecture that transforms standalone air quality monitors into a coordinated network of environmental sensors. Our solution leverages ESP32-based monitoring devices integrated through Set CVRDT (Conflict-free Replicated Data Type), enabling the maintenance of a consistent global environmental context across all monitoring points. This

approach is augmented by the Raft consensus algorithm for reliable leader election and distributed coordination, ensuring robust system operation even in challenging network conditions.

The practical implementation of our system demonstrates its capability to track pollution propagation patterns across different rooms, enabling automated responses based on a comprehensive understanding of the indoor environment. By converting sophisticated machine learning models to run efficiently on ESP32 devices, we achieve real-time activity detection and air quality analysis at the edge, while maintaining system-wide coordination through our distributed architecture.

## 1.3 Contribution

Our research makes several significant contributions to the field of indoor air quality monitoring. First, it introduces a novel approach to creating a unified environmental context from distributed sensors, moving beyond isolated measurements to comprehensive environmental understanding. Second, it demonstrates the feasibility of running complex analysis models on resource-constrained devices while maintaining distributed coordination. Finally, it provides a practical framework for implementing automated, context-aware responses to air quality events, potentially improving both environmental quality and occupant safety in indoor spaces.

The implications of this work extend beyond academic settings to various indoor environments, including residential buildings, educational institutions, and commercial spaces. By enabling proactive air quality management based on comprehensive environmental awareness, our system represents a significant step forward in addressing the critical challenges of indoor air pollution, particularly in developing regions where this issue has substantial public health implications.

# Chapter 2

# Related Work

This chapter presents a comprehensive review of existing research relevant to our work, organized into three main categories: distributed systems, IoT-based environmental monitoring, and machine learning approaches for activity identification.

## 2.1 Distributed Systems

In distributed systems, maintaining consistency and coordination among multiple nodes remains a fundamental challenge. The Raft consensus algorithm [14] provides an understandable approach to distributed consensus through a replicated state machine protocol. Unlike more complex alternatives, Raft decomposes consensus into three subproblems: leader election, log replication, and safety, making it more implementable while maintaining strong consistency guarantees.

For managing distributed data structures, Conflict-free Replicated Data Types (CRDTs) [12] have emerged as a powerful solution. Yu and Rostad [22] proposed a low-cost set CRDT based on causal lengths, which provides eventual consistency while minimizing storage and communication overhead. Their approach is particularly relevant

for IoT applications where resources are constrained and network connectivity may be intermittent.

Recent advances in 6G cellular networks have further enabled distributed sensing capabilities. Fang et al. [6] demonstrated how MIMO systems can be leveraged for joint communication and sensing, providing new possibilities for distributed environmental monitoring systems.

## 2.2 IoT-based Environmental Monitoring

The market for indoor air quality monitors has shown remarkable growth, estimated at US$ 5006 million in 2023 and projected to reach US$ 11672 million within the next decade [7]. Modern monitoring solutions [15, 1, 2] typically combine local sensing with cloud computing capabilities, enabling sophisticated data analysis and visualization.

Several studies have explored multi-device deployments for comprehensive environmental monitoring. Brazauskas et al. [3] developed real-time adaptive platforms for building information modeling, while Verma et al. [20] demonstrated the effectiveness of distributed sensor networks in detecting various indoor activities through air quality fluctuations. These systems typically monitor multiple parameters including CO2, VOCs, particulate matter, temperature, and humidity.

The integration of air quality monitors with household appliances has enabled new applications. Fang et al. [5] developed AirSense, an intelligent home-based sensing system that could detect activities like cooking, smoking, and spraying. Similar work by Ding et al. [4] explored the integration of environmental monitoring with split air conditioning systems for improved indoor climate control.

## 2.3 Machine Learning for Activity Identification

The application of machine learning to environmental sensor data has enabled sophisticated activity detection systems. Traditional approaches often relied on direct video [9] or audio [11] surveillance, raising significant privacy concerns. In contrast, environmental sensing provides a privacy-preserving alternative while still achieving high detection accuracy.

Recent studies have demonstrated remarkable success in activity recognition using various sensing modalities. Wearable devices have shown promise [13, 23], but their invasive nature and limited battery life make them unsuitable for continuous monitoring. RF-based sensing systems [18, 21] offer another approach, particularly effective in industrial and warehouse applications for tracking packages and monitoring workers.

Environmental sensor-based approaches have achieved significant results. Gambi et al. [8] demonstrated high accuracy in activity recognition using IoT air quality data. Zhong et al. [24] developed systems for indoor air quality forecasting and user interaction, while Zhu et al. [25] explored artificial intelligence techniques for non-uniform indoor environment control.

## 2.4 Privacy and Security Considerations

The correlation between indoor activities and air quality patterns raises important privacy considerations [19]. While environmental sensing is less intrusive than direct surveillance, the potential for activity inference necessitates careful consideration of data handling and user consent mechanisms [20].

## 2.5 Research Gap

Current air quality monitoring systems operate in isolation, preventing the development of a comprehensive environmental context across connected spaces. While existing research has demonstrated high accuracy in activity detection using air quality data [8, 20], these solutions lack mechanisms for coordinated sensing and decision-making across multiple devices. Additionally, while distributed systems research has developed robust solutions for consistency and consensus [14, 22], their application to environmental monitoring remains unexplored. Our work bridges this gap by implementing a distributed system that combines Set CVRDT for state management and Raft for consensus, enabling coordinated environmental monitoring and activity detection across multiple sensors while maintaining system-wide consistency.

# Chapter 3

# Data Collection

## 3.1 Lab Activity

For evaluating our distributed system's effectiveness in activity detection, we utilized a comprehensive dataset[16] from a three-month study conducted in an academic research laboratory setting. The dataset, collected by researchers using the DALTON air quality monitoring platform, comprises 7,456 data points with annotations for eight distinct indoor activities across three primary categories: engagement and occupancy (enter, exit), occupant behaviour (fan on/off, AC on/off), and prohibited practices (gathering, eating).

The data collection setup involved four air quality monitoring units positioned strategically in the corners of a research laboratory, enabling comprehensive coverage of the indoor space. Each monitoring unit captured multiple environmental parameters including:

1. Gaseous pollutants: CO2, VOCs (Volatile Organic Compounds)
2. Particulate matter: PM2.5, PM10
3. Environmental parameters: Temperature (T), Humidity (H)

The dataset's feature engineering encompasses various statistical measures calculated over sliding windows, providing rich contextual information about air quality variations:

1. Maximum and minimum values indicating pollution peaks and baseline levels
2. Standard deviation measuring pollutant level fluctuations
3. Rate of change calculations for both rising and falling pollution levels
4. Peak count and duration metrics for threshold exceedances
5. Long-stay measurements for persistent pollution levels

The dataset distribution reflects realistic usage patterns in an academic setting, with entry/exit events being the most frequent activities, followed by fan operations. Air conditioning events appear less frequently, typically corresponding to equipment cooling requirements. The prohibited activities (gathering and eating) represent the smallest portion of the dataset, aligning with expected laboratory protocols.

This comprehensive dataset provides an ideal testbed for our distributed system implementation, offering:

1. Multiple sensor locations for testing distributed coordination
2. Diverse activity patterns for evaluating detection capabilities
3. Rich feature sets for machine learning model evaluation
4. Real-world noise and variations for testing system robustness

The temporal resolution and feature richness of this dataset make it particularly suitable for evaluating our system's ability to maintain a coherent global context across distributed sensors while performing real-time activity detection.

The data collection setup involved deploying DALTON air quality monitoring modules at four corners of the research lab, capturing environmental changes influenced by lab activities [16]. The dataset includes a proportion of recorded activity classes, where the most frequent activities are lab members entering and exiting, while prohibited practices like eating and gathering are much less frequent (3.1) [16].

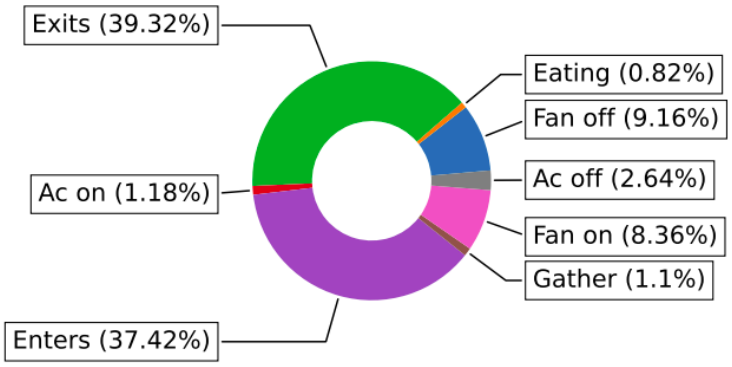



FIGURE 3.1: Class Distribution

## 3.2 Cooking Activity Dataset

Our system evaluation leverages a comprehensive dataset[17] from a six-month study across various indoor environments in India. This dataset contains 5,259 data points focusing on cooking activities and their impact on indoor air quality.

### 3.2.1 Dataset Characteristics

The dataset captures measurements from multiple co-located sensor devices, monitoring:

- **Environmental Parameters**:
  - Gaseous pollutants (CO2, VOCs)

  - Particulate matter (PM2.5, PM10)
  - Temperature and Humidity

- **Measurement Metrics**:
  - Statistical measures (mean, min, max, standard deviation)
  - Change rates and threshold-based metrics
  - Temporal patterns and duration measurements

### 3.2.2 Cooking Activity Classification

The dataset provides a hierarchical classification of cooking activities:

- **Cooking Categories (5)**:
  - Boiling, Deep-frying
  - Shallow-frying, Steaming
  - Combined methods (e.g., Shallow-frying with Boiling)
- **Food Items (11)**: Including diverse items such as:
  - tea, rice, fish, roti, egg, lentils, ladiesfinger, poppy seeds, pointed gourd, potato, leafy vegetables

### 3.2.3 System Validation

This dataset enables comprehensive validation of our system's capabilities:

- **Activity Detection**: Real-time classification of cooking methods and food types

- **Pollution Tracking**: Analysis of pollutant propagation patterns between spaces
- **Environmental Impact**: Assessment of different cooking activities' effects on air quality
- **System Adaptability**: Testing across various scales (residential kitchens to canteens)

The diverse nature of cooking activities and environments represented in this dataset provides a robust foundation for evaluating our distributed monitoring system's effectiveness in real-world scenarios.

# Chapter 4

# SetCvRDT Implementation and Testing

## 4.1 Introduction

State-based Conflict-free Replicated Data Types (CvRDTs) represent a fundamental advancement in distributed systems, particularly in scenarios requiring eventual consistency without central coordination. This implementation provides strong eventual consistency guarantees while maintaining simplicity in implementation.

The Set CvRDT implementation addresses several critical challenges in distributed systems:

- Network partitions and temporal disconnections between nodes
- Concurrent operations across distributed replicas
- State synchronization without centralized coordination
- Handling node failures and late-joining nodes

Our implementation focuses on maintaining three crucial properties:

- Strong eventual consistency: All replicas that have received the same updates eventually reach an equivalent state
- Commutativity and Associativity: The order of operations does not affect the final state
- Idempotency: Repeated application of the same operation has no additional effect

### 4.1.1 Leveraging Set CvRDT for Distributed Activity Recognition

In our distributed indoor air quality monitoring system, machine learning models running on ESP32 nodes independently analyze sensor data to detect activities that impact air quality. A critical challenge is maintaining a consistent view of these detected activities across all nodes, especially given the unreliable nature of indoor wireless networks and the need for immediate local responses to air quality changes. To address this, we implement a Set-based Conflict-free Replicated Data Type (CvRDT) to manage the distribution and synchronization of activity predictions across the network.

The Set CvRDT structure maintains three key components:

- The current set of valid predictions (`main_set`)
- Newly added predictions (`add_set`)
- Expired predictions (`rem_set`)

Each prediction entry contains the detected activity type, timestamp, source node identifier, and confidence level. This structure ensures that all nodes eventually

converge to a consistent view of detected activities across the monitored space, even when network connectivity is intermittent.

```
struct ActivityPrediction {
    uint16_t prediction_id;     // Unique identifier
    uint16_t node_id;          // Source node
    uint16_t timestamp;        // Detection time
    ActivityType activity;     // Detected activity
    float confidence;         // Prediction confidence
    RoomLocation location;    // Where activity was detected

};
```

Our implementation addresses several critical requirements:

1. **Real-time Response:** Each ESP32 node can independently detect and respond to activities while maintaining eventual consistency with other nodes.
2. **Resource Efficiency:** The system automatically manages prediction lifetime and removes expired predictions to maintain efficient memory usage on resource-constrained ESP32 devices.
3. **Network Resilience:** During network partitions, nodes continue to function independently and automatically reconcile their states when connectivity is restored.
4. **Scalability:** The system efficiently handles multiple nodes joining or leaving the network, making it suitable for deployment in various indoor environments.

This approach enables our system to maintain a globally consistent view of detected activities while allowing each node to operate autonomously when needed. The Set CvRDT structure forms the foundation for coordinated responses to air quality events, ensuring that all nodes work from a unified understanding of the indoor environment's state.

## 4.2 Properties and Implementation of Set CvRDT

### 4.2.1 Architecture Overview

The Set-based Conflict-free Replicated Data Type (CvRDT) [10] system implements a *set data structure that maintains consistency across multiple nodes, thus ensuring a distributed global data structure to store contextual information.* Each node exposes two primary endpoints: `/add` for inserting items and `/rem` for removing items from the locally stored `main_set`. The system ensures that operations performed at any node are eventually reflected across all nodes in the network.

### 4.2.2 Data Structure Components

Each node in the system maintains three ordered set data structures:

- `main_set`: Stores the current state locally and directly processes insertion and deletion requests. Insertions or deletions requests from users at the `/add` and `/rem` endpoints are applied directly to the `main_set`.
- `add_set`: Records all items inserted into the `main_set`. Whenever an item is inserted into the `main_set`, it's also inserted into the `add_set`.
- `rem_set`: Tracks all items removed from the `main_set`. Whenever an item is removed from the `main_set`, it's inserted into the `rem_set`.

### 4.2.3 Convergence Property

The convergence property ensures that eventually all replicas or nodes will end up in the same state. In this case, the `main_set` will be the same across all the replicas.

*Proof.* The implementation ensures that eventually all replicas or nodes will end up in the same state, with the `main_set` being identical across all replicas. This is achieved through the following mechanisms:

**State Synchronization** The `add_set` and `rem_set` are fully broadcasted over UDP using another thread every 1 second, as we're implementing a state-based CRDT.

**State Update Protocol** When a replica receives a UDP packet, it processes updates through the following steps:

1. It first iterates over the received `add_set` and inserts items into the `main_set` based on two conditions:
    - The item is not already present in its local `add_set` (ensuring that the item is not present in the `main_set`).
    - The item is not present in the received `rem_set` over UDP (skipping the extra operation of adding and removing the item later, which is useful for replicas with no initial state, such as newly added replicas).
2. If the item is found in the received `rem_set` and also in the local `main_set`, it gets removed from the `main_set`.
3. After iterating over the received `add_set`, it iterates over the received `rem_set` and removes items from the `main_set`, if found.
4. Whenever an item is inserted or removed from the `main_set`, the item gets inserted into the local `add_set` or `rem_set` accordingly.

**Fault Tolerance** In case of the failure of a node or if a node is newly added, they will eventually catch up with the same final state as other replicas by the above implementation.

**Conclusion** The above points demonstrate the convergence property, ensuring that the `main_set` eventually ends up in the same state irrespective of node failure or newly added nodes. □

### 4.2.4 Commutativity Property

The commutativity property ensures that the order in which set operations (insertion and deletion of integer values) are performed by the replicas does not affect the final state. For instance, the order of inserting multiple items into the set has no impact on the final state of the set.

*Proof.* Consider a system with three nodes: A, B, and C. The following sequence of operations demonstrates commutativity:

1. **Initial Operation at Node A:** Node A receives request to insert element 5. Element 5 is inserted into Node A's `main_set`. Element 5 is simultaneously added to Node A's `add_set`
2. **Concurrent Operation at Node B:** Node B receives request to insert element 8 Element 8 is inserted into Node B's `main_set` Element 8 is simultaneously added to Node B's `add_set`
3. **State Broadcasting:** Both Node A and Node B broadcast their respective states
4. **Final States:** Node A's `main_set` $= \{5, 8\}$ Node B's `main_set` $= \{5, 8\}$ Node C's `main_set` $= \{5, 8\}$

**Key Observation** Node C achieves the same final state $\{5, 8\}$ regardless of the order in which it receives the UDP packets from nodes A and B. The insertion sequence (5 then 8, or 8 then 5) has no impact on the final state of its `main_set`.

**Conclusion** This example demonstrates that our implementation ensures commutativity, as the final state of the system is independent of the order of operations. □

### 4.2.5 Associativity Property

The associativity property ensures that rearranging or grouping the operations won't affect the final state of the system. Our implementation maintains this property inherently through its set-based structure.

### 4.2.6 Idempotency Property

This property ensures that performing the same operation multiple times won't affect the final state.

*Proof.*

- If an item is already present in the `main_set`, another insertion request for the same item won't affect the state of the set. Upon receiving an insertion request via UDP by the `add_set`, it won't affect the final state as the implementation first checks if the element to be inserted via the UDP packet is already present in the local `add_set` or not. Whenever an item gets inserted into the `main_set`, it also gets inserted into the local `add_set`.

- Similarly, for the `rem_set` in the received UDP packet, it gets compared with the local `rem_set`, and only items not present in the local `rem_set` are stepped up for removal from the `main_set`.

- Applying the same operation on the set is inherently idempotent. In our case, we ensure idempotency on the basis of CRDT, i.e., upon receiving multiple UDP packets with no changes at all in the `add_set` or `rem_set`, it isn't going to affect the final state of our sets.

□

### 4.2.7 Handling Reinsertions

- It is to be noted that, as we're using two different sets to maintain the insertion and deletion of items from the `main_set`, it won't allow us to insert the same item twice. For example, if an item is added first and then removed from the `main_set`, we won't be able to insert the same item again. To solve this issue, we're attaching a unique ID with each item in the set.
- Whenever an item is inserted using the `/add` endpoint, it gets linked with a unique ID and the pair gets inserted into the `main_set` and the `add_set`. When removing the item, the `rem_set` also stores the unique ID along with the item.
- This removes the conflict, ensuring that an item, irrespective of removal, can be inserted again whenever required or requested.

This removes the conflict, ensuring that an item, irrespective of removal, can be inserted again whenever required or requested.

## 4.3 Testing and Verification of Set CvRDT

### 4.3.1 Implementation Challenges in Set CvRDT

#### 4.3.1.1 Multiple Instance Deletion Problem

During testing, we encountered an issue with removal operations in our Set CvRDT implementation:

- **Problem Description:**
  - When multiple instances of the same value (e.g., "33") existed with different unique IDs

  - The removal operation only deleted the first matching instance
  - Example scenario:
    * Value 33 existed with IDs: ["2117599665",33], ["6069681432",33], ["3390855067",33]
    * Removal operation only removed one instance
    * Other instances remained in the set, causing inconsistency
  - Root Cause: Comparator only matched the first occurrence of a value

- **Solution Implementation:**
  - Modified the comparator to properly handle value-based comparisons
  - Implemented iterative removal using while loop for multiple instances
  - Added value-based search functionality to find all matching instances
  - Solution ensured complete removal of all instances of a value

The challenge highlighted the importance of careful consideration in handling duplicate values with unique identifiers in distributed systems.

### 4.3.2 Test Case: Multiple Node Insert-Remove Operations

This test case demonstrates the system's ability to handle multiple insertions and removals across different nodes while maintaining consistency, particularly focusing on the reinsertion mechanism with unique IDs.

#### 4.3.2.1 Initial Insertion Phase

First, node 192.168.0.103 adds element 33:

```
# Broadcasting from 192.168.0.103
192.168.0.103 -> {"add_set":[["2117599665",33]],"rem_set":[]}

# Received by other nodes
Received msg: {"add_set":[["2117599665",33]],"rem_set":[]}
```

Subsequently, multiple nodes perform insertions:

- Node 192.168.0.104 adds 44
- Node 192.168.0.105 adds 55
- Node 192.168.0.106 adds 66

After all broadcasts and updates, the system converges to:

```
# Broadcasting and Received by all nodes
{"add_set":[["1796048333",66],["2117599665",33],
            ["5138773642",55],["996384795",44]],"rem_set":[]}

# Main set state for all nodes
main_set: [["1796048333",66],["2117599665",33],
           ["5138773642",55],["996384795",44]]
```

#### 4.3.2.2 Removal and Reinsertion Operations

Node 192.168.0.106 removes element 33:

```
# Broadcasting from 192.168.0.106 and Received by all nodes
{"add_set":[["1796048333",66],["2117599665",33],
            ["5138773642",55],["996384795",44]],
 "rem_set":[["2117599665",33]]}

# Main set state for all nodes
main_set: [["1796048333",66],["5138773642",55],
```

```
          ["996384795",44]]
```

#### 4.3.2.3 Concurrent Reinsertion Scenario

Node 192.168.0.106 attempts to add 33:

```
# Broadcasting from 192.168.0.106 and Received by nodes(except
    192.168.0.103)
{"add_set":[["1796048333",66],["2117599665",33],
           ["5138773642",55],["6069681432",33],
           ["996384795",44]],
 "rem_set":[["2117599665",33]]}
```

Node 192.168.0.103 (which missed previous update) adds 33:

```
# Broadcasting from 192.168.0.103
{"add_set":[["1796048333",66],["2117599665",33],
           ["3390855067",33],["5138773642",55],
           ["996384795",44]],
 "rem_set":[["2117599665",33]]}

# Received msg at 192.168.0.103
{"add_set":[["1796048333",66],["2117599665",33],
           ["5138773642",55],["6069681432",33],
           ["996384795",44]],
 "rem_set":[["2117599665",33]]}
```

After some time,

```
# Main set, add set and rem set state for all nodes
main_set: [["1796048333",66],["3390855067",33],
          ["5138773642",55],["6069681432",33],
          ["996384795",44]]
{"add_set":[["1796048333",66],["2117599665",33],
           ["3390855067",33],["5138773642",55]
           ,["6069681432",33],["996384795",44]],
```

```
 "rem_set":[["2117599665",33]]}
```

#### 4.3.2.4 Final Removal Operation

Node 192.168.0.105 removes all instances of 33:

```
# Broadcasting from 192.168.0.105 and Received by all nodes
{"add_set":[["1796048333",66],["2117599665",33],
            ["3390855067",33],["5138773642",55]
            ,["6069681432",33],["996384795",44]],
 "rem_set":[["2117599665",33],["6069681432",33],
             ["3390855067",33]]}

# Final main set state for all nodes
main_set: [["1796048333",66],["5138773642",55],
          ["996384795",44]]
```

This test case demonstrates several key properties of our Set CvRDT implementation:

- Successful handling of concurrent operations
- Proper functioning of the unique ID mechanism for reinsertions
- Consistent state convergence across all nodes
- Correct handling of multiple removes and adds of the same value

### 4.3.3 Test Case: Concurrent Operations and Mass State Changes

This test case demonstrates the system's behavior with numerous reinsertions of the same value and mass removals, focusing on extensive concurrent operations.

#### 4.3.3.1 Sequential Operations from Node 192.168.0.103

```
# Node 192.168.0.103 adds 31
192.168.0.103 -> {"add_set":[["4343304636",31]],"rem_set":[]}

# Node 192.168.0.103 adds 32
192.168.0.103 -> {"add_set":[["4343304636",31],
                             ["4419977690",32]],"rem_set":[]}

# Node 192.168.0.103 removes 32
192.168.0.103 -> {"add_set":[["4343304636",31],
                             ["4419977690",32]],
                  "rem_set":[["4419977690",32]]}

# Main set state after these operations
main_set: [["4343304636",31]]
```

#### 4.3.3.2 Concurrent Operations of Value 10 Across Multiple Nodes

Multiple nodes repeatedly add element 10, demonstrating the unique ID mechanism:

```
# Broadcasting and Received messages after multiple add operations
   of 10
{"add_set":[
    ["4343304636",31],
    ["4419977690",32],
    ["5167911320",10],  # First add of 10
    ["5399923041",10],  # Second add of 10
    ["5518743674",10],  # Third add of 10
    ["5599595286",10],  # Fourth add of 10
    ["5672231968",10],  # Fifth add of 10
    ["5756328104",10],  # Sixth add of 10
    ["5824517186",10],  # Seventh add of 10
    ["5928374561",10],  # Eighth add of 10
    ["6023456789",10],  # Ninth add of 10
```

```
    ["6123456789",10],  # Tenth add of 10
    ["6223456789",10],  # Eleventh add of 10
    ["6323456789",10]   # Twelfth add of 10
],
"rem_set":[
    ["4419977690",32],  # Removed 32
    ["5167911320",10],  # First removal of 10
    ["5399923041",10],  # Second removal of 10
    ["5518743674",10],  # Third removal of 10
    ["5599595286",10],  # Fourth removal of 10
    ["5672231968",10]   # Fifth removal of 10
]}

# Main set state after concurrent operations
main_set: [["4343304636",31],
           ["5756328104",10],   # Sixth add of 10 remains
           ["5824517186",10],   # Seventh add of 10 remains
           ["5928374561",10],   # Eighth add of 10 remains
           ["6023456789",10],   # Ninth add of 10 remains
           ["6123456789",10],   # Tenth add of 10 remains
           ["6223456789",10],   # Eleventh add of 10 remains
           ["6323456789",10]]   # Twelfth add of 10 remains
```

#### 4.3.3.3 Final Mass Removal Operation

Node 192.168.0.103 initiates a mass removal:

```
# Final broadcast and received message showing mass removal
{"add_set":[
    ["4343304636",31],["4419977690",32],["5167911320",10],
    ["5399923041",10],["5518743674",10],["5599595286",10],
    ["5672231968",10],["5756328104",10],["5824517186",10],
    ["5928374561",10],["6023456789",10],["6123456789",10],
    ["6223456789",10],["6323456789",10]
],
```

```
"rem_set":[
    ["4343304636",31],     # Removing 31
    ["4419977690",32],     # Already removed 32
    ["5167911320",10],["5399923041",10],["5518743674",10],
    ["5599595286",10],["5672231968",10],["5756328104",10],
    ["5824517186",10],["5928374561",10],["6023456789",10],
    ["6123456789",10],["6223456789",10],
    ["6323456789",10]      # Removing all instances of 10
]}

# Final main set state after mass removal
main_set: []
```

This test case demonstrates:

- Handling of multiple insertions of the same value with unique IDs
- Proper management of concurrent add operations
- Successful tracking of multiple instances of the same value
- Effective mass removal of all elements
- Maintenance of consistency during complex concurrent operations

### 4.3.4 Key Findings

The testing revealed three critical aspects of our implementation:

1. **Performance Under Load**:
    - Successfully handled multiple concurrent operations
    - Maintained consistency during rapid sequential operations
    - Efficient management of state synchronization

2. **Reliability Features**:

    - Automatic recovery from network partitions
    - Consistent state convergence across scenarios
    - Effective conflict resolution in concurrent operations

These results validate our implementation's suitability for distributed air quality monitoring systems, demonstrating both theoretical correctness and practical reliability.

# Chapter 5

# RAFT Implementation and Testing

## 5.1 Introduction

In distributed indoor air quality monitoring systems, maintaining a coordinated response to detected activities requires not just consistent state sharing (achieved through Set CvRDT) but also reliable decision-making leadership. The Raft consensus algorithm plays a crucial role in our system by providing a robust leader election mechanism that complements our Set CvRDT implementation.

### 5.1.1 Motivation and Integration

The integration of Raft with Set CvRDT in our system serves several critical purposes:

1. **Coordinated Activity Detection:**

- While Set CvRDT ensures all nodes maintain consistent state of detected activities, Raft's leader election mechanism ensures centralized decision-making.
- The leader node becomes responsible for aggregating activity predictions from all nodes and making final determinations about ongoing activities.

2. **Resource Optimization:**
   - Instead of all nodes processing and analyzing the global state, the elected leader takes primary responsibility for complex decision-making.
   - This is particularly important for resource-constrained ESP32 devices, as it prevents redundant processing across nodes.
3. **Reliable State Management:**
   - Set CvRDT ensures all nodes eventually have consistent activity prediction data.
   - The Raft leader uses this consistent state to make authoritative decisions about detected activities.
   - This combination prevents conflicting interpretations of the same activity data across different nodes.

### 5.1.2 System Architecture Overview

Our implementation combines the strengths of both algorithms:

- **Data Flow:**
  - Individual nodes use local ML models to generate activity predictions
  - Predictions are shared across nodes using Set CvRDT
  - The Raft-elected leader processes the aggregated predictions

- **Decision Making:**
  - The leader node analyzes the global state maintained by Set CvRDT
  - Makes authoritative decisions about detected activities
  - Broadcasts decisions to all follower nodes

This approach creates a robust and efficient system where Set CvRDT handles state replication and consistency, while Raft provides the necessary leadership structure for coordinated decision-making in our distributed air quality monitoring system.

## 5.2 RAFT Implementation and Testing

### 5.2.1 System Design

My implementation [10] focuses on enabling collaborative decision-making in a network of low-power WiFi-enabled ESP32 devices. The system implements a distributed leader election mechanism that ensures coordinated operation while maintaining efficient resource utilization. Each node in the network participates in a democratic process of leader election, with clear roles and responsibilities assigned based on their current state.

#### 5.2.1.1 Node States and Roles

The system operates through three distinct states that define the behavior and responsibilities of each node:

1. **Follower State**
   - All nodes initialize in the Follower state, representing the default operational mode where they listen for and respond to leadership directives.

- Followers maintain a passive role in the network, focusing on their primary task of collecting and processing local sensor data while awaiting instructions from the leader.
- Each follower maintains an election timeout, randomly set between 3000-5000ms, to ensure distributed election triggering.
- In this state, nodes directly transmit their collected environmental data and activity predictions to the current leader, awaiting confirmation before executing any significant operations.
- The random election timeout helps prevent simultaneous election initiations across multiple nodes, reducing network contention.

2. **Candidate State**

- When a follower's election timeout expires without receiving a heartbeat, it transitions to the Candidate state, initiating the leadership election process.
- The Candidate state represents an active participation phase where the node seeks leadership authority for the current term.
- Upon entering this state, the node immediately increments its term counter and initiates a voting process by broadcasting vote requests to all other nodes.
- Candidates persist in this state until either they win the election, another leader emerges, or a timeout occurs requiring a new election cycle.
- During this state, the node maintains a vote count and continues to process incoming vote responses until a majority is achieved or the election times out.

3. **Leader State**

- The Leader state represents the authoritative node for the current term, responsible for coordinating network activities and maintaining system stability.
- Only one node can be in the Leader state at any given time within a specific term, ensuring consistent decision-making across the network.
- The leader maintains its authority by broadcasting regular heartbeat messages every 250ms to all followers, preventing unnecessary election triggers.
- Leaders are responsible for collecting and aggregating environmental data and activity predictions from all follower nodes.
- They process this aggregated data to make final determinations about detected activities and coordinate system-wide responses.

### 5.2.2 Election Process

#### 5.2.2.1 Candidate Transition Protocol

The transition from Follower to Candidate follows a carefully orchestrated process:

1. **Term Management**
   - The node increments its local term counter, establishing a new election period to maintain system-wide time ordering.
   - The increased term number ensures that older election attempts are automatically invalidated, preventing conflicting leadership claims.
   - Each node maintains this term counter locally, using it to validate incoming vote requests and heartbeat messages.
   - Term numbers provide a logical clock for the system, helping maintain consistency in distributed decision-making.

2. **Vote Request Broadcasting**

   - The candidate constructs a vote request message containing its unique identifier (IP address) and the current term number.
   - This message is broadcast using UDP to reach all nodes in the network simultaneously.
   - If no majority is achieved within the election timeout period, the candidate rebroadcasts its vote request with the same term number.

#### 5.2.2.2 Voting Mechanism

The voting process comprises several interconnected components that ensure fair and reliable leader election:

(a) **Self-Voting Protocol**

   - When a node enters the Candidate state, it immediately casts a vote for itself, initializing the vote count to 1.
   - This self-vote is recorded in the node's local state by setting the `votedFor` variable to its own nodeId.

(b) **Vote Request Processing**

   - Upon receiving a vote request, nodes perform a thorough validation process comparing the received term with their local term.
   - If the received term is higher than the local term, nodes update their own term, reset their state to Follower, and grant their vote.
   - This term comparison ensures that older election attempts are automatically superseded by newer ones.
   - Votes are granted only once per term, preventing multiple votes from the same node within a single term.

(c) **Vote Response Handling**

- Vote responses are sent directly to the requesting candidate using unicast UDP messages.
- Each response contains the voter's nodeId and a boolean vote value (1 for approval).
- The candidate maintains a running tally of received votes, checking against the known total number of nodes.
- When the vote count exceeds half the total number of nodes, the candidate transitions to Leader state.

### 5.2.3 Leader Operations

Once a node achieves leadership, it implements several critical operational procedures:

(a) **Heartbeat Mechanism**

- The leader broadcasts heartbeat messages every 250ms to maintain its authority.
- Each heartbeat contains the leader's nodeId and current term, allowing followers to validate leadership.
- The regular heartbeat interval prevents unnecessary election timeouts in follower nodes.
- Heartbeats serve as both a leadership assertion and a network health indicator.

(b) **State Coordination**

- The leader becomes the central point for collecting environmental data and activity predictions from all nodes.
- It processes incoming data and maintains a global view of the system's state.

### 5.2.4 Network Communication Implementation

Our system implements a robust network communication framework:

(a) **Message Types and Formats**

- **Vote Request Messages:**
  - Format: "VoteRequest [nodeId] [term]"
  - Broadcast to all nodes during election
  - Contains sender's identity and current term
- **Vote Response Messages:**
  - Format: "VoteResponse [nodeId] [voteValue]"
  - Unicast directly to candidate
  - Includes voter's decision (1 for approval)
- **Heartbeat Messages:**
  - Format: "Heartbeat [leaderId] [term]"
  - Broadcast every 250ms
  - Maintains leader's authority

(b) **UDP Communication Strategy**

- Message handlers process incoming packets asynchronously.
- Network partitions are handled gracefully through term numbering.

(c) **Node Discovery Process**

- New nodes broadcast "Joined" messages containing their IP address.
- Existing nodes respond with "ACK" messages to acknowledge new members.
- The node registry is maintained dynamically as nodes join and leave.
- This enables automatic scaling of the voting system as the network changes.

## 5.3 Testing and Challenges

Our testing phase involved multiple ESP32 devices configured in various scenarios to validate the Raft implementation's reliability and identify potential challenges. We conducted extensive testing with different node configurations and failure scenarios.

### 5.3.1 Challenges in Implementation

#### 5.3.1.1 Infinite Election Problem

During initial implementation and testing, we encountered a critical issue that revealed a fundamental flaw in our election mechanism. Consider the following scenario with three ESP32 nodes:

- Node A (10.5.20.101): Initial leader
- Node B (10.5.20.102): Follower
- Node C (10.5.20.103): Follower

When Node A (leader) failed, the following sequence occurred:

(a) Both remaining nodes detected leader failure through missed heartbeats

(b) Both nodes transitioned to candidate state with term=12

(c) Election Process:

- Node B: Became candidate, voted for itself (1 vote)
- Node C: Became candidate, voted for itself (1 vote)
- Neither node could achieve majority (required 2 votes)
- Both nodes timed out and restarted election
- Term remained at 12 during re-elections

(d) Result: System entered an infinite loop of failed elections

**Root Cause and Solution** The infinite election problem stemmed from two critical design flaws: static term numbers during election retries and persistent self-voting with equal terms. We resolved this by implementing term progression in the election protocol, where:

- Each election attempt increments the candidate's term number
- Election timeouts trigger additional term increments
- Term differences enable nodes to break voting deadlocks
- Higher terms take precedence in vote collection

This solution successfully prevents election deadlocks by ensuring term progression during failed elections, allowing the system to consistently achieve leadership consensus.

### 5.3.2 Three-Node Configuration Testing

We initially tested with three ESP32 nodes using fixed IP addresses (10.5.20.101, 10.5.20.102, 10.5.20.103).

#### 5.3.2.1 Test Scenarios and Analysis

(a) **Initial Leader Election**

- Observation: Node 10.5.20.102 became leader in term 4
- Analysis: During simultaneous startup, random election timeouts led to Node 10.5.20.102 being the first to win election
- This demonstrates successful initial leader election with random timeouts preventing election conflicts

(b) **Natural Leadership Transition**

- Observation: Leadership transferred to 10.5.20.101 in term 6

- Possible Cause: Node 10.5.20.101 might have missed heartbeats from Node 10.5.20.102, triggering new election
- Higher term number (6) ensures other nodes accept the new leadership

(c) **Leader Recovery Scenarios**

- After leader shutdown and restart:
  - Term 7: Node 10.5.20.102 regained leadership
  - Term 9: Node 10.5.20.103 became leader
  - Term 11: Node 10.5.20.101 became leader again
- Each transition demonstrates successful handling of node failures and rejoins
- Term increments ensure proper leadership succession

### 5.3.3 Four-Node Configuration Testing

Extended testing with four nodes (adding 10.5.20.104) revealed additional dynamics:

(a) **Stable Operation**

- Initial state: Node 103 as leader with term=14
- Demonstrated stable leadership with larger node count

(b) **Leader Failure Scenarios**

- Leader shutdown triggered successful re-election
- Leader recovery behavior dependent on timing:
  - If heartbeat received first: Remains follower
  - If timeout occurs first: Participates in new election

(c) **Follower Dynamics**

- Follower failures: No impact on leadership

- Follower recovery: Potential leadership changes based on term numbers
- Missed heartbeats occasionally trigger leadership changes

(d) **Critical Scenario: Split Vote**

- With two nodes offline (total nodes = 4, active nodes = 2):
  - Maximum possible votes = 2
  - Required majority = $\lfloor \frac{4}{2} \rfloor + 1 = 3$
  - Results in inability to elect leader
- Demonstrates importance of maintaining quorum for leader election

### 5.3.4 Key Testing Insights

Our testing revealed several important aspects of the implementation:

- Term number progression is crucial for breaking election deadlocks
- System naturally handles individual node failures and recoveries
- Quorum requirements prevent split-brain scenarios
- Timing of heartbeats versus timeouts influences system behavior

# Chapter 6

# Machine Learning Models on ESP32 using EmLearn

## 6.1 Introduction

In our distributed indoor air quality monitoring system, the ability to detect and classify activities based on sensor data directly on ESP32 devices is crucial for real-time response and system efficiency. However, implementing machine learning models on resource-constrained devices like ESP32 presents unique challenges. This chapter explores our implementation of various machine learning models using the EmLearn library, which enables the conversion of Python-trained models to efficient C code suitable for embedded systems.

### 6.1.1 Motivation and Challenges

Running machine learning models on ESP32 devices presents several key challenges:

- Limited computational resources and memory
- Lack of native floating-point operation support

- Need for real-time prediction capabilities
- Resource-efficient model implementation requirements

### 6.1.2 Implementation Approach

Our implementation focuses on five different machine learning models:

(a) Decision Trees

(b) Random Forest

(c) Extra Trees

(d) Gaussian Naive Bayes

(e) Neural Network (sklearn_mlp)

These models were selected based on their potential for efficient implementation on resource-constrained devices and their ability to handle our activity classification tasks.

### 6.1.3 Performance Considerations

The ESP32's hardware limitations significantly influence model selection and implementation:

- 16-bit integer operations preferred over floating-point calculations
- Limited decimal precision (maximum one decimal place)
- Memory constraints affecting model complexity
- Need for balance between accuracy and prediction speed

Initial performance metrics reveal significant variations across models:

- **Decision Trees**: Fastest prediction times (6-9 microseconds average)

- **Random Forest**: Excellent accuracy-speed trade-off (16-34 microseconds average)
- **Extra Trees**: High accuracy with moderate speed (43-64 microseconds average)
- **Gaussian Naive Bayes**: Limited by floating-point operations (157-356 microseconds average)
- **Neural Network**: Constrained by hardware limitations (61-89 microseconds average)

This chapter details our implementation process, the challenges encountered, and the solutions developed to achieve efficient activity classification on ESP32 devices using the EmLearn library. We present comprehensive performance analyses across different datasets and discuss the trade-offs between model complexity, accuracy, and computational efficiency.

## 6.2 Lab Activity Dataset Evaluation

Our evaluation encompasses multiple machine learning models for lab activity detection using air quality sensor data. We carefully designed the implementation process to ensure efficient operation on resource-constrained ESP32 devices while maintaining high accuracy.

### 6.2.1 Implementation Considerations

#### 6.2.1.1 Dataset Split and Training

In our implementation, we utilized a 70-30 train-test split with random state 3 for reproducibility:

```
train, test = train_test_split(dataset,
                               test_size=0.3,
```

```
3                                   random_state=3)
```

This split provided sufficient training data while maintaining a robust test set for validation.

#### 6.2.1.2 Neural Network Architecture Considerations

While previous research [16] has demonstrated high accuracy (97.7% F1-score) using neural networks with three 64-neuron hidden layers for similar activity detection tasks, our implementation faced different constraints. Our primary objective was not to replicate these results but to achieve efficient execution on ESP32 devices. Initial attempts to implement larger networks (64,64 neurons) failed due to:

- Memory constraints when converting to C code using EmLearn
- Excessive computational requirements for ESP32
- Generated C code complexity exceeding ESP32's capabilities

Consequently, we opted for a smaller architecture with two 10-neuron hidden layers:

```
sklearn.neural_network.MLPClassifier(
    hidden_layer_sizes=(10,10),
    activation='relu',
    max_iter=1000)
```

While this reduced architecture was successfully deployed on ESP32, it resulted in significantly lower accuracy (49.45%) compared to the original research. This trade-off between model complexity and hardware constraints highlights the challenges in implementing sophisticated neural networks on resource-constrained devices.

### 6.2.2 Data Preparation and Training Process

The training process involved several critical preprocessing steps to ensure compatibility with ESP32's limitations:

```
def load_dataset():
    data = pandas.read_csv('lab_activity.csv')
    df = data.copy()

    # exclude timestamp - ts
    df = df.drop(columns=['ts'])

    # exclude any feature starting with PMS10
    df = df.loc[:, ~df.columns.str.startswith('PMS10')]

    # exclude any feature starting with VoC
    df = df.loc[:, ~df.columns.str.startswith('VoC')]

    # Optimize for ESP32's 16-bit integer operations
    for column in df.columns:
        if column != 'target':
            max_val = df[column].max()
            if max_val * 100 > 2**16:
                if max_val * 10 < 2**16:
                    df[column] = (df[column] * 10).astype(int)
                else:
                    df[column] = (df[column] * 100).astype(int)
    return df
```

Key preprocessing steps included:

- Data scaling to fit within ESP32's 16-bit integer range
- Removal of high-cardinality features to reduce model complexity
- Conversion of floating-point values to fixed-point representation

- Feature selection focusing on most discriminative air quality parameters (exclude timestamp and any feature starting with PMS10 or VoC)

### 6.2.3 Model Implementation and Performance

TABLE 6.1: Comprehensive Model Performance Analysis for Lab Activity Classification

| Model | Accuracy | Avg. Time (µs) | Max Time (µs) | ESP32 Errors |
|---|---|---|---|---|
| Decision Tree | 93.56% | 9 | 42 | 0 |
| Random Forest | 97.41% | 34 | 330 | 0 |
| Extra Trees | 97.36% | 62 | 463 | 0 |
| Gaussian NB | 48.28% | 356 | 382 | 0 |
| Neural Network | 49.45% | 89 | 438 | 0 |

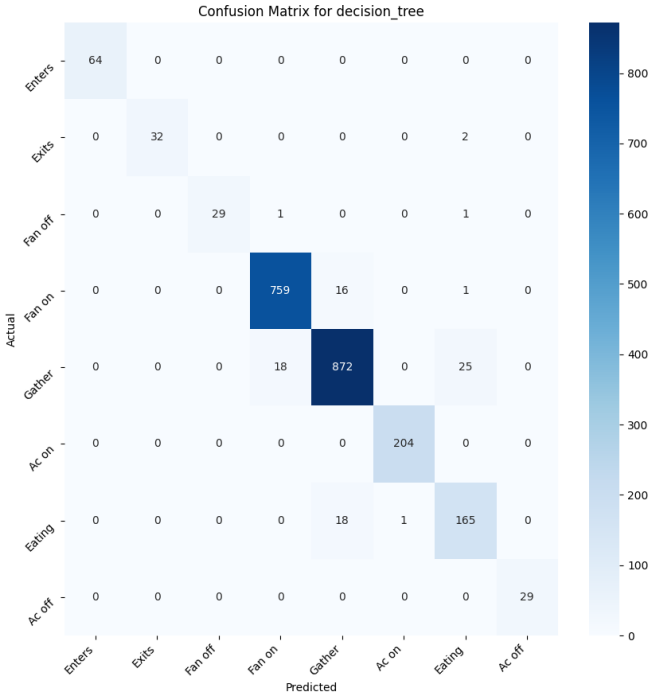



FIGURE 6.1: Lab Activity: Confusion Matrix for Decision Tree

Our evaluation reveals a clear performance divide between model types. Tree-based models demonstrated superior performance, with Random Forest achieving the best accuracy-speed trade-off (97.41% accuracy, 34 µs). Extra Trees

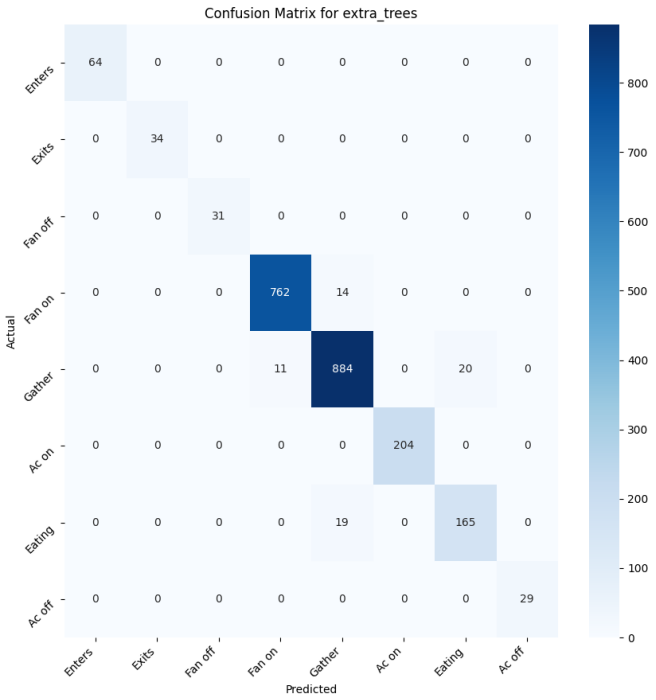



FIGURE 6.2: Lab Activity: Confusion Matrix for Extra Trees

showed similar accuracy but required longer computation time, while Decision Trees offered the fastest predictions with good accuracy.

Models requiring floating-point operations performed poorly on ESP32. Both Neural Network (using two 10-neuron hidden layers) and Gaussian Naive Bayes achieved less than 50% accuracy, with significantly longer prediction times. These results highlight the importance of integer-based operations for efficient ESP32 deployment.

### 6.2.4 Implementation Verification and Error Analysis

We developed a comprehensive testing framework to verify the EmLearn-generated C code:

```
1 int test() {
2     const int n_features = testset_features;
```

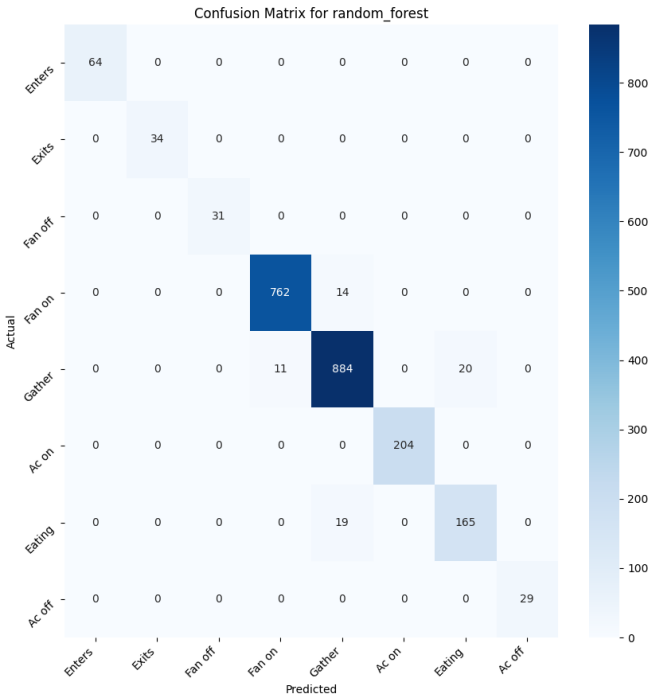



FIGURE 6.3: Lab Activity: Confusion Matrix for Random Forest

```
3     const int n_testcases = testset_samples;
4     int errors = 0;
5
6     for (int i=0; i<n_testcases; i++) {
7         const int16_t *features = testset_data + (i*n_features)
      ;
8         const int expect = testset_results[i];
9         const int32_t out = model_predict(features, n_features)
      ;
10
11        if (out != expect) {
12            errors += 1;
13        }
14    }
15    return errors;
16 }
```

Remarkably, all models showed zero errors when comparing Python model predictions with their C implementations, indicating successful conversion and deployment despite the resource constraints.

### 6.2.5 Resource Utilization Analysis

Our implementation revealed several critical insights about resource utilization:

- **Memory Efficiency**: Tree-based models demonstrated superior memory efficiency due to their integer-based operations and simple decision structures.
- **Computation Time**: Maximum prediction times stayed under 500 microseconds, with tree-based models significantly outperforming others:
  - Decision Tree: 42 µs max
  - Random Forest: 330 µs max
  - Extra Trees: 463 µs max
- **Float vs Integer Operations**: Models requiring floating-point calculations showed significantly higher computation times and lower accuracy, highlighting the importance of integer-based operations for ESP32 deployment.

This comprehensive evaluation establishes Random Forest as the optimal choice for our application, balancing high accuracy with reasonable computational demands while maintaining reliable performance on ESP32 devices.

## 6.3 Cooking Activity Dataset Evaluation

Our cooking activity dataset provides a unique opportunity to evaluate model performance on two distinct prediction tasks using the same sensor data: identifying the type of cooking method being used (cooking_type) and determining what food is being cooked (cooking_food).

### 6.3.1 Dataset Preprocessing & Modeling

The sensor data preprocessing follows similar steps for both prediction tasks:

```
def load_dataset():
    data = pandas.read_csv('cook_data.csv')
    df = data.copy()

    # For cooking_type prediction
    df = df.drop(columns=['food'])
    df = df.rename(columns={'ctype': 'target'})

    # OR for food prediction
    # df = df.drop(columns=['ctype'])
    # df = df.rename(columns={'food': 'target'})
```

Key preprocessing steps include:

- Removal of timestamp and PMS10 features
- Integer conversion for ESP32 compatibility (scaling values to fit 16-bit range)
- Categorical encoding of target variables

### 6.3.2 Challenges with Complex Models

Similar to our lab activity implementation, complex models faced significant challenges:

TABLE 6.2: Model Performance for Cooking Type Classification

| Model | Accuracy | Avg. Time | Max Time | ESP32 Errors |
|---|---|---|---|---|
| Decision Tree | 97.28% | 6 µs | 39 µs | 0 |
| Random Forest | 99.68% | 16 µs | 271 µs | 0 |
| Extra Trees | 99.81% | 43 µs | 339 µs | 0 |
| Gaussian NB | 47.72% | 157 µs | 181 µs | 0 |
| Neural Network | 44.36% | 61 µs | 352 µs | 0 |

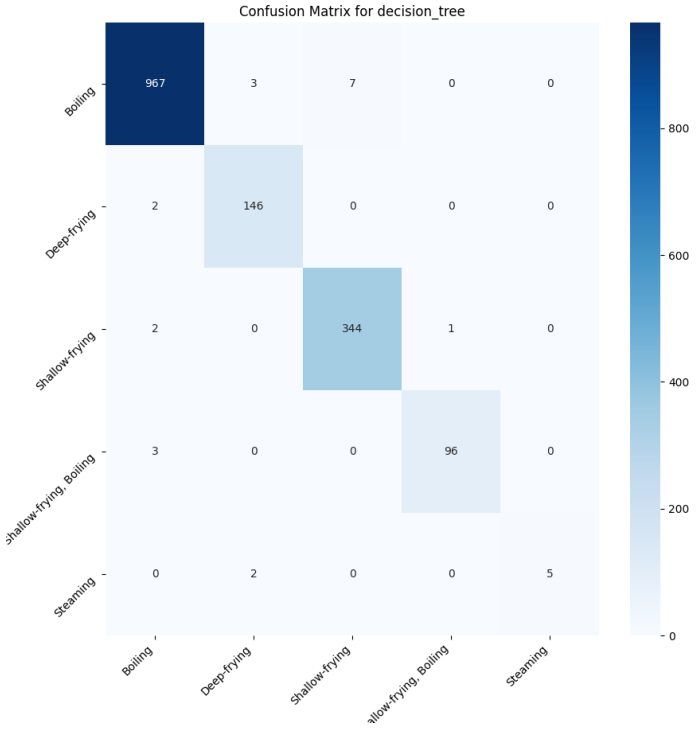



FIGURE 6.4: Cooking Type: Confusion Matrix for Decision Tree

(a) **Neural Network Implementation:**

- While previous research demonstrated high accuracy with 64-neuron hidden layers (achieving 97.7% accuracy[16]), EmLearn's C code generation limitations forced us to use a reduced architecture (10,10 neurons) for ESP32 deployment
- Despite successful deployment, this reduced architecture showed poor performance (around 45% accuracy) due to the trade-off between model complexity and hardware constraints

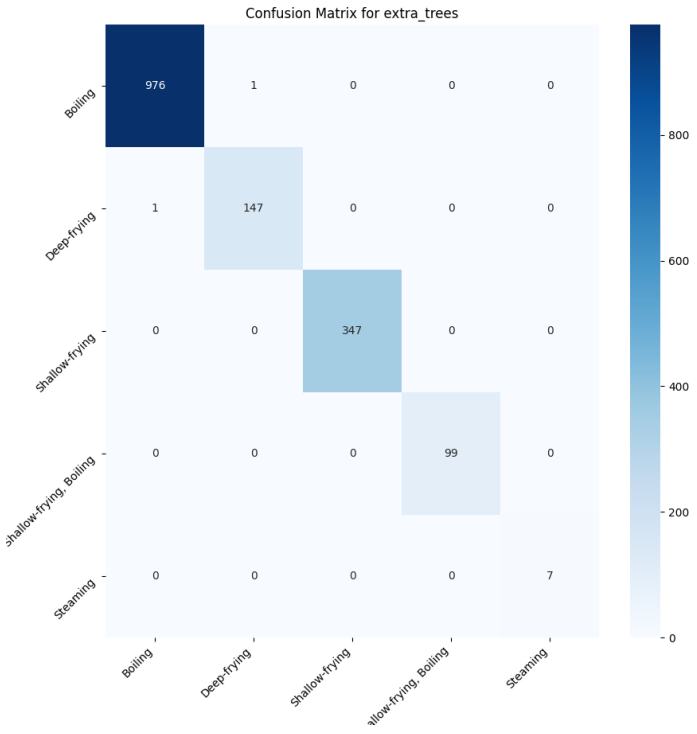



FIGURE 6.5: Cooking Type: Confusion Matrix for Extra Trees

TABLE 6.3: Model Performance for Food Type Classification

| Model | Accuracy | Avg. Time | Max Time | ESP32 Errors |
|---|---|---|---|---|
| Decision Tree | 94.17% | 7 µs | 38 µs | 0 |
| Random Forest | 99.49% | 23 µs | 289 µs | 0 |
| Extra Trees | 99.68% | 64 µs | 311 µs | 0 |
| Gaussian NB | 46.52% | 178 µs | 209 µs | 0 |
| Neural Network | 45.71% | 83 µs | 391 µs | 0 |

(b) **Floating-Point Operations:**

- Both Neural Network and Gaussian NB struggled with ESP32's floating-point limitations, showing significantly higher prediction times and poor accuracy
- The necessary integer conversion of sensor data further impacted model performance

Despite these challenges, all models achieved perfect verification between Python

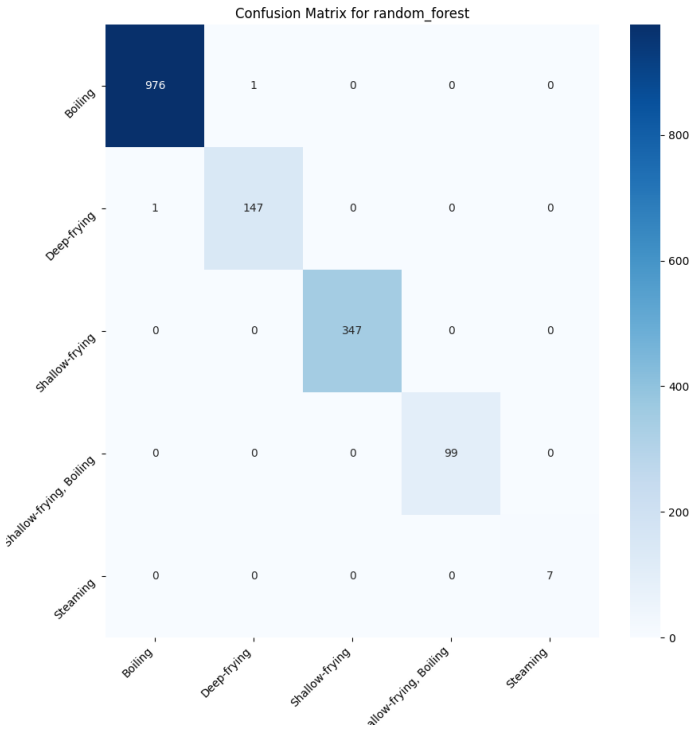



FIGURE 6.6: Cooking Type: Confusion Matrix for Random Forest

and C implementations, with zero errors reported in ESP32 deployment, confirming EmLearn's successful translation capabilities while highlighting the performance constraints of complex architectures.

### 6.3.3 Performance Analysis

Our analysis focuses on model performance in cooking activity detection tasks across both cooking type and food prediction scenarios.

#### 6.3.3.1 Tree-based Models

(a) **Model Comparison:**

- *Extra Trees*: Highest accuracy (99.81% cooking type, 99.68% food) but longer processing (43-64 µs)

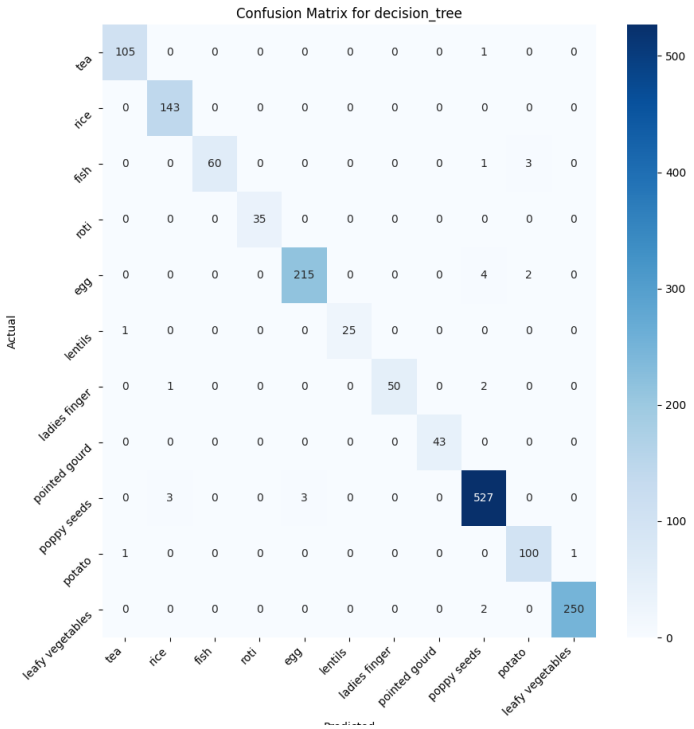



FIGURE 6.7: Food Type: Confusion Matrix for Decision Tree

- *Random Forest*: Best accuracy-speed balance (>99.4%, 16-23 µs)
- *Decision Tree*: Fastest execution (6-7 µs) with good accuracy (>94%)

(b) **Key Observations:**

- Consistent performance across both prediction tasks
- Clear trade-off between accuracy and prediction speed
- All models maintained real-time processing capability

#### 6.3.3.2 Limitations of Complex Models

(a) **Performance Issues:**

- Neural Network: Poor accuracy (44-45%) with 10,10 architecture
- Gaussian Naive Bayes: Slowest execution (157-178 µs) with similar poor accuracy

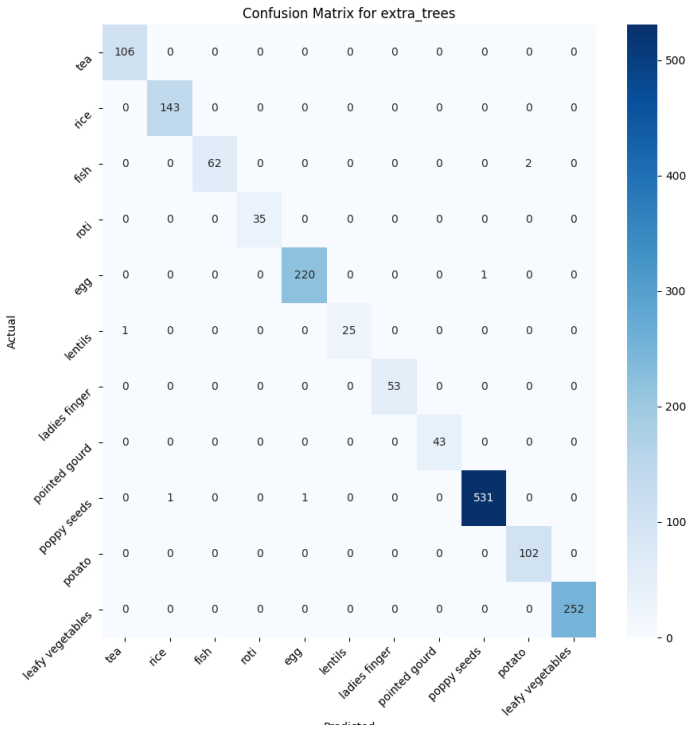



FIGURE 6.8: Food Type: Confusion Matrix for Extra Trees

(b) **Implementation Constraints:**

- Integer conversion requirements impacted neural network performance
- Floating-point operations proved unsuitable for real-time ESP32 deployment

#### 6.3.3.3 Task-Specific Performance Comparison

Our analysis reveals distinct patterns between cooking type and food prediction tasks:

- **Cooking Type Prediction** achieved faster predictions and higher accuracy, suggesting simpler classification boundaries
- **Food Type Prediction** required longer processing times and showed marginally lower accuracy, likely due to more complex feature relationships

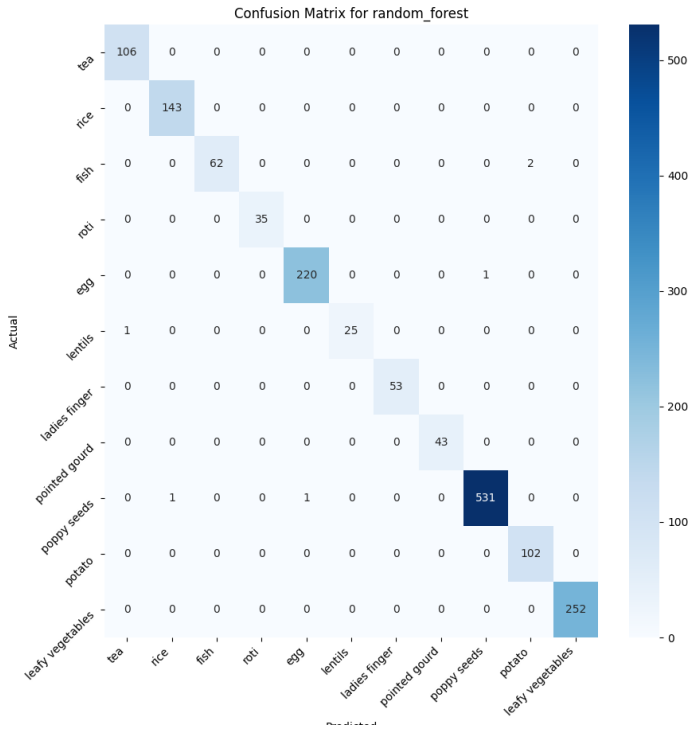



FIGURE 6.9: Food Type: Confusion Matrix for Random Forest

Overall, Random Forest emerged as the optimal choice, offering the best balance between accuracy and computation time for real-time cooking activity detection on ESP32 devices.

### 6.3.4 Implementation Insights

Our dual-task implementation revealed key performance patterns:

- Maximum prediction times stayed under 400 microseconds across all models
- Both tasks maintained zero error rates in ESP32 implementation
- Tree-based models, particularly Random Forest, provided optimal performance balance for both cooking type and food prediction tasks

This evaluation demonstrates that our implementation can successfully handle multiple prediction tasks using the same sensor data, with tree-based models proving most effective for both cooking type and food identification on resource-constrained ESP32 devices.

# Chapter 7

# Conclusion and Future Work

## 7.1 Conclusion

This thesis presents a comprehensive distributed system for indoor air quality monitoring and activity detection using ESP32 devices. We have successfully implemented and tested three key components that work together to create a robust and efficient system:

(a) **Set CvRDT Implementation**: We developed a state-based conflict-free replicated data type system that ensures consistent data sharing across distributed nodes. This implementation maintains eventual consistency even under network partitions and node failures, providing a reliable foundation for distributed data management.

(b) **Raft Consensus Algorithm**: Our adaptation of the Raft algorithm for ESP32 devices enables reliable leader election and coordinated decision-making. This ensures that the network maintains a clear hierarchy for processing and aggregating activity predictions, even in challenging network conditions.

(c) **Machine Learning Models**: We successfully implemented and validated multiple machine learning models on ESP32 devices using the EmLearn library. The most efficient model, Random Forest, achieved over 97% accuracy for lab activity detection and 99% for cooking activity detection, while maintaining minimal prediction times suitable for real-time operation.

The integration of these components creates a system where:

- Nodes independently generate activity predictions using local ML models
- Set CvRDT ensures consistent sharing of predictions across the network
- Raft-elected leader coordinates the aggregation and interpretation of these predictions

Our testing has demonstrated the feasibility of running sophisticated distributed algorithms and machine learning models on resource-constrained ESP32 devices while maintaining high accuracy and reliable operation.

## 7.2 Future Work

While we have successfully implemented and tested each component independently, several important areas remain for future development:

### 7.2.1 System Integration

The primary focus of our future work will be the complete integration of all three components:

- Implementing unified codebase combining Set CvRDT, Raft, and ML models on ESP-32

- Optimizing memory usage across all components when running simultaneously
- Developing efficient communication protocols between components
- Testing system behavior under various real-world conditions

### 7.2.2 Dataset Enhancement

To improve the system's activity detection capabilities:

- Expanding the dataset to include more diverse activities
- Collecting data from different indoor environments (homes, offices, laboratories, etc.)
- Including seasonal variations in air quality patterns
- Incorporating more complex, overlapping activities

### 7.2.3 Real-world Deployment

To validate system effectiveness in practical scenarios:

- Testing in various residential and commercial settings
- Evaluating system response to different ventilation conditions
- Analyzing performance across different building layouts
- Gathering user feedback for system refinement

These developments will move our research from individual component validation to a fully integrated, practical system for distributed indoor air quality monitoring and activity detection. The enhanced dataset and real-world testing will enable more robust and generalizable activity detection, making the system more valuable for practical applications.

# Bibliography


[1] Airthings. Airthings. https://www.airthings.com/, 2023.

[2] Amazon. Airknight 9-in-1 indoor air quality monitor, 2022. Available at: https://www.amazon.in/AIRKNIGHT-Portable-Formaldehyde-DetectorMonitoring/dp/B09LY1QBDM.

[3] Justas Brazauskas, Rohit Verma, Vadim Safronov, Matthew Danish, Jorge Merino, Xiang Xie, Ian Lewis, and Richard Mortier. Data management for building information modelling in a real-time adaptive city platform. *arXiv preprint arXiv:2103.04924*, 2021.

[4] Nan Ding, Fudan Liu, Feng Pang, Jingyu Su, Lianyu Yan, and Xi Meng. Integrating the living wall with the split air conditioner towards indoor heating environment improvement in winter. *Case Studies in Thermal Engineering*, 47:103061, 2023.

[5] Biyi Fang, Qiumin Xu, Taiwoo Park, and Mi Zhang. Airsense: an intelligent home-based sensing system for indoor air quality analytics. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 109–119. ACM, 2016.

[6] Xinran Fang, Wei Feng, Yunfei Chen, Ning Ge, and Yan Zhang. Joint communication and sensing toward 6g: Models and potential of using mimo. *IEEE Internet of Things Journal*, 10(5):4093–4116, 2022.

[7] Future Market Insights. Indoor air quality monitor market outlook. Technical report, Future Market Insights, 2023.

[8] Ennio Gambi, Giulia Temperini, Rossana Galassi, Linda Senigagliesi, and Adelmo De Santis. Adl recognition through machine learning algorithms on iot air quality sensor dataset. *IEEE Sensors Journal*, 20(22):13562–13570, 2020.

[9] Muhammad Attique Khan, Kashif Javed, Sajid Ali Khan, Tanzila Saba, Usman Habib, Junaid Ali Khan, and Aaqif Afzaal Abbasi. Human action recognition using fusion of multiview and deep features: an application to video surveillance. *Multimedia Tools and Applications*, 83(5):14885–14911, 2024.

[10] Amit Kumar. Github repo link: `https://github.com/Amitkumar9199/MTP-1`.

[11] Gierad Laput, Karan Ahuja, Mayank Goel, and Chris Harrison. Ubicoustics: Plug-and-play acoustic activity recognition. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, pages 213–224. ACM, 2018.

[12] Carlos Baquero Marek Zawirski Marc Shapiro, Nuno Preguiça. A comprehensive study of convergent and commutative replicated data types. *RR-7506, Inria – Centre Paris-Rocquencourt; INRIA.*, pp.50., 2011.

[13] Kavous Salehzadeh Niksirat, Lev Velykoivanenko, Noé Zufferey, Mauro Cherubini, Kévin Huguenin, and Mathias Humbert. Wearable activity trackers: A survey on utility, privacy, and security. *ACM Computing Surveys*, 56(7):1–40, 2024.

[14] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference*, pages 305–319. USENIX Association, 2014.

[15] Prana Air. Parnaair sensible+ air quality monitor. `https://www.pranaair.com/air-quality-monitor/sensible-plus-air-monitor/`, 2023.

[16] Sandip Chakraborty Prasenjit Karmakar, Swadhin Pradhan. Exploiting air quality monitors to perform indoor surveillance: Academic setting. *26th International Conference on Mobile Human-Computer Interaction*, v3(1):6, 2024.

[17] Sandip Chakraborty Prasenjit Karmakar, Swadhin Pradhan. Indoor air quality dataset with activities of daily living in low to middle-income communities. *arXiv:2407.14501*, v3, 2024.

[18] Argha Sen, Anirban Das, Swadhin Pradhan, and Sandip Chakraborty. Continuous multi-user activity tracking via room-scale mmwave sensing. In *2024 23rd ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 163–175. IEEE, 2024.

[19] The New York Times. Cambridge analytica and facebook: The scandal and the fallout so far. *The New York Times*, 2018.

[20] Rohit Verma, Justas Brazauskas, Vadim Safronov, Matthew Danish, Ian Lewis, and Richard Mortier. Racer: Real-time automated complex event recognition in smart environments. In *29th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 634–637. ACM, 2021.

[21] Chenshu Wu, Feng Zhang, Beibei Wang, and K J Ray Liu. msense: Towards mobile material sensing with a single millimeter-wave radio. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 4(3):1–20, 2020.

[22] Weihai Yu and Sigbjørn Rostad. A low-cost set crdt based on causal lengths. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, pages 1–6. ACM, 2020.

[23] Hang Yuan, Shing Chan, Andrew P Creagh, Catherine Tong, Aidan Acquah, David A Clifton, and Aiden Doherty. Self-supervised learning for human activity recognition using 700,000 person-days of wearable data. *NPJ Digital Medicine*, 7(1):91, 2024.

[24] Sailin Zhong, Hamed S Alavi, and Denis Lalanne. Hilo-wear: Exploring wearable interaction with indoor air quality forecast. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–8. ACM, 2020.

[25] Hao-Cheng Zhu, Chen Ren, and Shi-Jie Cao. Dynamic sensing and control system using artificial intelligent techniques for non-uniform indoor environment. *Building and Environment*, 226:109702, 2022.