Machine Learning Based WiFi Tracker

Henry Blue*
henry.blue@aalto.fi
Aalto University
Helsinki, Finland

Memoona Batool
memoona.batool@aalto.fi
Aalto University
Helsinki, Finland

Santeri Kokkonen
santeri.kokkonen@aalto.fi
Aalto University
Helsinki, Finland

Abstract

This project presents a low-cost, machine learning–based system for locating a target Wi-Fi device using Channel State Information (CSI) captured passively by two synchronized ESP32-S3 microcontrollers. The primary goal is not only to detect the device's presence but to estimate its direction relative to the user—enabling guided movement toward the device. This has practical applications in scenarios such as search and rescue, where a buried or lost device must be located within rubble or obstructed environments. The system classifies the target's position into one of ten spatial zones using a real-time CSI capture pipeline and synchronized data preprocessing. Five machine learning models were evaluated—KNN, SVM, Random Forest, Deep neural networks (DNN), and Long Short-Term Memory (LSTM). The Bi-LSTM model achieved the highest accuracy at 97.97%, with the DNN close behind at 90.4%, confirming the effectiveness of both temporal and spatial feature learning. This work demonstrates that accurate, direction-aware localization is achievable using commodity Wi-Fi hardware without requiring device pairing or infrastructure changes.

CCS Concepts

• **Networks** → **Location based services**; • **Computing methodologies** → *Supervised learning*; • **Hardware** → Sensor devices and platforms.

Keywords

Channel state information (CSI), WiFi, localization, positioning, machine learning, neural network, k-nearest-neighbours, support vector machine, random forest, long short-term memory.

ACM Reference Format:
Henry Blue, Memoona Batool, and Santeri Kokkonen. 2025. Machine Learning Based WiFi Tracker.

1 Introduction

During the past 30 years, Wi-Fi networks have become widespread in society. Alongside the progress of Wi-Fi, almost every person nowadays carries a device with Wi-Fi capabilities such as a smart phone. Additionally, there exists a plethora of other common devices that are connected to the internet via Wi-Fi.

Some of the Wi-Fi signal's qualities have been shown to depend on the devices location and environment. Mainly, the channel state information (CSI) quantizes the wireless channel. The signal channel between the Wi-Fi transmitter and receiver depends on the location of the devices and the environment.

Thus, previous studies have been able to utilize the CSI of Wi-Fi for numerous applications. For example, in [6] the authors introduce ConFi, a convolutional-neural-network-based Wi-Fi localization algorithm. Other applications based on the fluctuations of CSI have also been produced. For instance, in [35] a Wi-Fi-based person identification framework is presented, and in [30] an indoor fall detection system using Wi-Fi devices is proposed.

What's more, in the past 10 years low-cost system-on-chip (SoC) devices such as the ESP32 family of microcontrollers have been brought to market. These devices, having Wi-Fi capabilities, are an appealing option for sensing Wi-Fi networks.

However, due to timing and measurement inaccuracies of the ESP microcontrollers, arithmetically deriving the Angle-Of-Arrival (AoA) of the Wi-Fi signal is inaccurate, especially with a small number of ESPs. As motivation for this study, the arithmetic approach was first attempted. As expected, the approach lead to inaccurate results, similar to a random prediction of the AoA.

The objective of this study is to use widely used supervised Machine Learning (ML) models to predict the location of a Wi-Fi device. More specifically, the study aims to predict the direction and the distance of the device relative to a ESP-pair. The ESP microcontrollers are used to passively capture the signal data. This type of system has applications in, for example, search and rescue situations where a device needs to be located.

The main contributions of this study are:

(1) Present a low-cost, simple and reproducible ESP Wi-Fi tracking system.
(2) Evaluate different supervised ML models to determine which are most suitable for Wi-Fi tracking.

Lastly, the rest of this paper is structured as follows. In section 2, the background of key concepts related to the study are presented. Section 3 covers the system architecture. In section 4, the CSI capture configuration and the data-collection pipeline of the study is introduced. The dataset preparation and feature engineering is discussed in section 5. Section 6 presents the different ML models of the study and their performance. In section 7 the performance of the ML models is evaluated and analyzed. Finally, conclusions are drawn in section 8.

2 Background

Wireless sensing leverages radio frequency(RF) signals to perceive physical environment, enabling applications like localization and

*All authors contributed equally to this research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
UbiComp / ISWC 2025, Helsinki, Finland
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/2018/06

activity recognition without detected sensors [9][31]. The usual Received Signal Strength Indicator(RSSI) provides coarse-grained information about link quality but it lacks fine-grained sensing due to environmental noise and multipath insensitivity [15]. Channel State Information(CSI) emerged as a fundamental enabler for advanced sensing due to its inherent ability to catch amplitude and phase response of the wireless channel across orthogonal frequency subcarriers. CSI captures rich multipath signatures induced by environmental dynamics; thus, offering significantly higher spatial and temporal resolution than RSSI [9][1]. The evolution of Wi-Fi standards progressively enhanced CSI availability, resolution and utility, shaping the design space for sensing applications.

2.1 CSI and Wi-Fi

Channel State Information (CSI) captures the per-subcarrier complex channel gains between every transmit and receive antenna in an 802.11 link. In our case, we use the complex CSI to capture geometric context about a link to estimate a user's relative location. Before describing the implementation, we review how successive Wi-Fi generations expose CSI and how those mechanisms shape our design choices.

IEEE 802.11n (Wi-Fi 4) was the first mainstream generation where commodity chipsets could expose CSI with only firmware or driver tweaks. It also introduced MIMO with up to four spatial streams and optional 20 or 40 MHz channels. Here, MIMO-OFDM receivers estimate the channel for every subcarrier and antenna pair, getting a CSI matrix of complex gains [15, 34]. However, cross-vendor constraints limited practical CSI exchange. Despite this, researchers modified firmware: [15] released the Intel 5300 CSI Tool, which delivers amplitude and phase for 30 subcarriers per antenna pair on 20 MHz channels (60 on 40 MHz) [15]. Similar methods later appeared for selected Atheros chipsets [34]. Wi-Fi 4 uses a 64-point OFDM symbol (52 data + pilot tones) per 20 MHz; bonding two channels doubles the number of usable subcarriers, improving frequency resolution.

Wi-Fi 5 added to the CSI collection landscape by mandating explicit CSI feedback for beamforming and adding downlink MU-MIMO. An access point (AP) sends a Null Data Packet (NDP); stations compute CSI and reply with compressed reports [4, 25]. Wi-Fi 5 also increased Channel bandwidth to 80 MHz (optionally 160 MHz), raising FFT sizes to 256 (or 512) and quadrupling subcarrier counts over 20 MHz. Together with 256-QAM and eight spatial streams, this pushes CSI accuracy requirements. Projects such as Nexmon CSI enabled raw CSI extraction on Broadcom and Qualcomm chipsets [28, 34]. Wi-Fi 5 keeps the 64-point base, bonds channels up to 160 MHz, and supports eight spatial streams. Compressed feedback (e.g., quantised angles, SVD truncation) limits its overhead while still providing thousands of coefficients per snapshot, useful for applications such as localisation and vital-sign monitoring.

Wi-Fi 6 introduces OFDMA and improves MU-MIMO. Downlink beamforming still relies on High-Efficiency (HE) sounding: the AP transmits an HE-NDP; stations return HE-compressed feedback [22]. Wi-Fi 6 also adds uplink trigger-based sounding, letting the AP collect CSI from multiple users simultaneously and exploit channel reciprocity. Larger subcarrier-grouping factors further compress reports as FFT sizes grow. Wi-Fi 6 uses 78.125 kHz subcarrier spacing: a 256-point FFT at 20 MHz and a 1024-point FFT at 80 MHz. Longer 12.8 µs symbols and flexible guard intervals improve estimation under delay spread [22]. Up to 8 × 8 MU-MIMO and per-Resource-Unit reporting expand CSI dimensionality while remaining compatible with the 11ac model. Commodity 802.11ax devices routinely exchange these beamforming frames, which are protocol-compliant, unencrypted, and sniffer-visible, giving researchers easy access to CSI without vendor firmware hacks. Wi-Fi 6 therefore makes CSI an operational requirement and also simplifies research access [33].

IEEE approved the 802.11be amendment (Wi-Fi 7) on 26 September 2024, and the Wi-Fi Alliance launched the Wi-Fi Certified 7 Release 1 programme earlier that year, in January 2024 [18, 32]. Wi-Fi 7 doubles the maximum channel width to 320 MHz (or 160+160 MHz non-contiguous), mandates Multi-Link Operation (MLO) across 2.4/5/6 GHz, and supports 4096-QAM. The specification permits up to sixteen spatial streams; Release 1 devices support eight [3]. These parameters raise a single CSI snapshot to tens of thousands of complex coefficients. To bound feedback overhead, 802.11be defines new EHT-compressed beamforming report formats with finer quantisation, subcarrier grouping, and per-segment sounding; research explores AI-assisted compression. MLO reuses compressed CSI across concurrent links, and coordinated multi-AP modes rely on sharing reports over the wired backhaul. Extended EHT-LTF sequences (up to 16 symbols) maintain estimation accuracy for large antenna arrays [18]. With latency targets below 5 ms and MAC-layer throughput goals above 30 Gbps, Wi-Fi 7 places efficient CSI acquisition and compression at the centre of high-throughput WLAN operation.

2.2 Next-Generation Positioning and Sensing (802.11az & 802.11bf)

802.11az—Next Generation Positioning. Building on the RTT/FTM procedure of 802.11mc, 802.11az enlarges sounding bandwidths to 160–320 MHz, allows optional AoA/AoD estimation, and secures ranging with Pre-Association Security Negotiation (PASN). These PHY/MAC tweaks yield sub-decimetre indoor accuracy while remaining backward-compatible with Wi-Fi 6/7 frames [20, 21, 26, 36].Fine-grained CSI is implicitly embedded in each FTM exchange: every responder transmits HE/EHT-LTFs whose per-subcarrier estimates—and optionally angle metadata—are reported back in a compressed FTM Report. Thus, 802.11az elevates CSI from a research hack to a mandatory ingredient of secure, high-rate two-way ranging.

802.11bf—Wi-Fi Sensing. Task Group bf formalises CSI-based sensing across 2.4/5/6 GHz and 60 GHz by introducing sensing PPDUs (Null-Data and Trigger-based variants), a Sensing Service Period, and new control fields for privacy, calibration and quantisation. Draft timelines target final approval around 2027, with Draft 3.0 already defining bistatic/multistatic operation and <1 cm Doppler resolution [11, 19, 24, 27].Stations may request raw or Walsh-Hadamard–compressed CSI, enabling applications from fall detection to fine-grained occupancy mapping while capping overhead to a few kilobytes per burst. By codifying acquisition, feedback, and privacy safeguards, 802.11bf turns ad-hoc CSI hacks into an

interoperable, dual-purpose communications-and-sensing PHY for future Wi-Fi generations.

The 802.11BF sensing framework and procedure is outlined in [10]. Namely, two different sensing measurement acquisition procedures that can operate in the sub-7GHz and 60GHz bands, respectively. Regarding this paper, the standard's description of the sub-7GHz sensing procedure is relevant. In short, the sensing procedure contains four phases:

(1) Sensing Capabilities Exchange.
(2) Sensing Measurement Session.
(3) Sensing Measurement Exchange.
(4) Sensing Measurement Termination.

Also, the roles of the procedure are the following:

- Sensing initiator.
- Sensing responder.
- Sensing transmitter.
- Sensing receiver.

A device, that takes a part in the procedure, is either an initiator or a responder, and a transmitter and/or a receiver. Our aim is to emulate this procedure with two EPS32 base stations, and one user equipment.

2.3 Open-Source Wi-Fi CSI Ecosystem

To emulate these positioning scenarios, researchers and hobbyists have turned to the Espressif ESP32-S3, a low-cost SoC with Wi-Fi sensing capabilities. Espressif's `esp_csi` API in ESP-IDF allows the capture of various forms of CSI such as RSSI, noise, timestamps, and receive-control flags for each Wi-Fi frame [12]. Example programs—`csi_recv`, `csi_send`, and `esp-radar`—show how to log CSI in monitor mode, run controlled TX/RX tests, and carry out presence detection on the microcontroller itself. The ESP32-S3 supports both 20 MHz and 40 MHz 802.11n channels; the wider mode doubles the number of reported subcarriers without overloading on-chip memory.

Community firmware removes most of the remaining setup work. Hernandez's `ESP32-CSI-Tool` compiles to ESP32, S2, and S3 boards, offers passive or active capture, streams raw samples over serial or to an SD card, and ships Python helpers for parsing and plotting [16]. On the other hand, researchers have used single S3 boards for through-wall presence sensing and respiration monitoring; two-board setups have been used to add a simple phase baseline for direction estimates. More recently, ESPARGOS scales to eight synchronised ESP32 radios and produces real-time 8×8 CSI suitable for angle of arrival and spatial imaging; the general design and dataset are open, however implementation details remain closed source [14]. The related Asparagus project repackages the same hardware into a classroom kit and publishes labelled datasets to support reproducible studies.

To support using multiple ESP32-S3s as a radio array, tight timing is critical. Either wired or wireless reference clocks, or periodic GPIO triggers, reduce clock error to sub-microsecond levels; WS-WiFi demonstrated this approach on earlier ESP32 hardware, and the method carries over to the S3 [29].

In contrast, commodity NIC hacks offer a different balance of effort and capability. The Intel 5300 toolset reveals up to 3×3 MIMO CSI at 20 MHz but limits output to 30 subcarriers per channel [15]. Atheros and Broadcom firmware patches, as well as Nexmon, extend bandwidth to 80 MHz and keep multiple RF chains but require a full Linux host and custom drivers [28, 34]. The ESP32-S3, in contrast, is a five-dollar microcontroller that runs fully untethered. It cannot match wide-band, multi-stream fidelity, yet its cost and self-contained design enable larger or battery-powered deployments.

The ecosystem around the S3 therefore offers a practical path from raw CSI capture to sensing experiments. Official support provides stable data access, community tools shorten the learning curve, and recent projects prove that useful localisation and activity recognition are possible even with single-antenna hardware. Because of this, the ESP32-S3 is often the choice as a low-price, small-form-factor, and simple-to-develop sensing node for Wi-Fi sensing projects.

3 System Architecture

This project aims to create a system that will locate a source device relative to the user by using available Wi-Fi CSI information from connections with the target device. Given the aforementioned benefits, we selected the ESP32-S3 as the sensing node in our passive Wi-Fi localisation platform. Figure 1 outlines the final end-to-end architecture: two synchronised ESP32-S3 boards capture channel-state information (CSI) from an 802.11n access point and stream it, together with relevant metadata, to a host workstation for alignment, preprocessing, and finally inference. This section looks more at the hardware design choices we made, along with the reasoning behind those choices.

3.1 ESP32-S3 as a CSI Receiver

Starting with the specifics on the device, the ESP32-S3 is a single antenna (1×1) IEEE 802.11n SoC that operates exclusively in the 2.4 GHz band. Espressif's ESP-IDF exposes per-subcarrier CSI estimates through a callback interface; each report contains 64 complex samples for 20 MHz frames, and 128 samples for 40 MHz. Along with the raw frames the API reports, packet-level metadata such as RSSI, noise floor, timestamp, and `rx_ctrl` flags [12]. Although limited to Wi-Fi4 and SISO operation, prior work shows that ESP32-derived CSI is sufficient for presence, motion, respiration, and coarse localisation [16].

The SDK supports two capture modes. In the first, active mode, the ESP32 transmits probe frames to a target station and records CSI for both transmitted and received packets. On the other hand, in passive mode the ESP32 operates solely as a monitor, capturing CSI for every frame observed on a specified channel. For our system, we adopt passive mode to avoid injecting additional traffic and enabling us to locate a target we are unconnected to.

3.2 Single-Node Baseline and Motivation for an Array

Originally, we considered a single-ESP32-S3 tracking system as a baseline. For this, we set up a simple single ESP in passive mode and recorded a small trial dataset with three distinct locations. The objective of the system was to use the raw CSI information to predict which location an incoming packet came from. In these trials, we

achieved mediocre results. The neural network used for classification was scoring too low for a practical localisation application. For the 3-class problem a neural network was able achieve an accuracy of around 65%. This led us to focus on a synchronised two-node ESP radio array. Using an array of ESPs increases the diversity of received signals and allows us to calculate the angle of arrival based on the phase difference of the same packet arriving at each device. This results in a more robust feature to train the machine learning models.

Nonetheless, the single ESP acts as a good baseline. After collecting our full dataset (outlined in Section 5) with our synchronised two-ESP32-S3 setup, we used the data captured from each device to separately train a neural network and measure its classification accuracy. The neural network is the same as defined in Section 6.1. Table 1 summarises the classification accuracies of the neural network using the ESPs separately. Ten different classes, in this case locations, are used. The classes are combinations of five different angles and two different distances with respect to the ESPs.

Table 1: Single-Node Baseline, neural network

	CSI	RSSI	CSI+RSSI
Master ESP	86.9%	40.1%	**89.6%**
Worker ESP	78.5%	46.2%	83.4%

The results of the single-node setup show that there is a clear difference in the classification accuracy between the Master and Worker ESPs. However, both ESPs achieve over 80% accuracy when using the captured CSI and RSSI values together as features. This is a notable jump in accuracy from our initial tests, which is further discussed in Section 7.

3.3 Time Synchronisation for a Two-Node Array

One of the primary benefits of using a radio array rather than a single ESP32-S3 device is that it enables angle-of-arrival (AoA) calculations based on the phase difference between the two devices. However, in order to accurately combine information from multiple ESP32-S3 receivers, tight time and frequency synchronisation is required. At 2.4 GHz, a 1 ns timing error translates to roughly 90° of phase, so sub-nanosecond alignment is desirable for AoA estimation [7]. While the ESP API prints the CSI of all incoming packets in order, we still need tight time synchronisation to pair the incoming packets with synchronised times to avoid packets from becoming mispaired. We evaluated four candidate synchronisation approaches:

(1) Using a shared 40 MHz reference clock effectively eliminates carrier frequency offset (CFO); however, it requires additional clock hardware and modifications for integration [29].
(2) Alternatively, using a GPIO trigger pulse aligns MCU timers within tens of nanoseconds; residual CFO can then be tracked in software [13].
(3) Network-based protocols (NTP/PTP) introduce software timestamping, which can have 10–100 µs jitter, making them unsuitable for RF-phase work requiring tight synchronisation [7].
(4) RF phase calibration injects a known tone to remove static phase offsets after deployment, which can provide nanosecond level synchronisation but requires a reference device for calibration [14].

For the present two-board prototype, we adopt a hybrid strategy: an NTP time synchronisation built into the ESP ecosystem aligns internal timers, while per-packet sequence numbers, MAC addresses, and Time Synchronisation Function timestamps allow post-hoc pairing of corresponding CSI snapshots. Although this setup does not yield phase-coherent AoA, it suffices for the learning-based angle classifier described in Section 6.

3.4 Physical Layout and Host Interface

The final two-ESP32-S3 system setup is outlined in Figure 1. Here, we have attached the two ESP32-S3 devices so that the centers of the Wi-Fi modules are exactly one half-wavelength, or 6.25 cm, apart. Each board streams the CSI over USB to a Python daemon on the host PC for data processing and sample pairing. The daemon is explained in more detail in Section 4. Finally, the hardware specifics of the setup are outlined in Table 2.

Table 2: Hardware configuration of the ESP32-S3 array.

MCUs	2 × ESP32-S3-WROOM-1 (8 MB PSRAM)
Clocking	Internal 40 MHz crystal + 1 kHz GPIO trigger
Antenna spacing	6.25 cm (½λ @ 2.4 GHz)
Channel	2.4 GHz, channel 6 (2437 MHz), 20 MHz BW
Host link	USB 115200 bps (dev) / UDP 10 Mbps (capture)

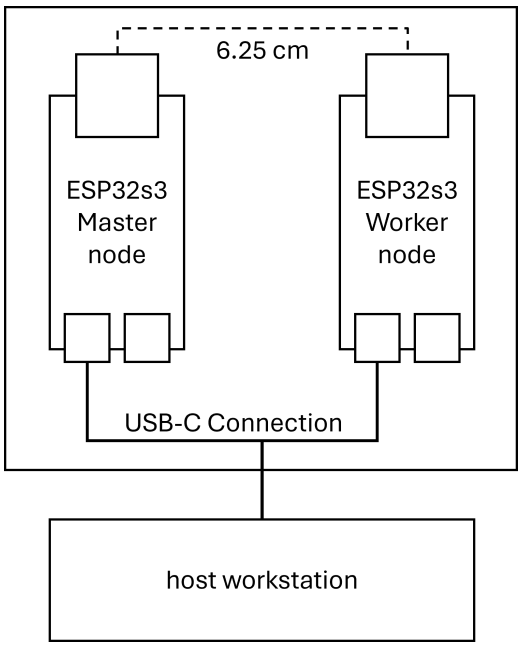

Figure 1: Physical layout of the two-ESP32-S3 setup with antenna spacing.

4 CSI Capture Configuration and Data-Collection Pipeline

Now, we explain how we process the raw CSI from two ESP32-S3 boards into a time-aligned dataset ready for angle-of-arrival (AoA) and distance analysis. The work sits on three layers: firmware patches that expose CSI, a host pipeline that pairs and processes packets, and a storage format that keeps every field we need for later study.

4.1 Software Environment

This project builds on Espressif's IoT Development Framework (ESP-IDF). To start, we used Stephan Hernandez's `ESP32-CSI-Tool` [16]. This project was intended for IDF v4.3, which is now deprecated. As such, we adjusted Wi-Fi driver calls and buffer management to make it compatible with the current IDF v5.4.1. Table 3 states the tool versions; the exact code used for this can be found in the project GitHub through the appendix.

Table 3: Toolchain.

IDE	Visual Studio Code 1.90 (Remote WSL)
ESP-IDF	v5.4.1 (release)
VS Code extension	Espressif IDF 1.10.0
Firmware language	C/C++ (FreeRTOS)
Host scripting	Python 3.12 (asyncio, numpy, h5py, pandas)

Next, to observe every Wi-Fi frame at line rate, we changed three parts of the driver:

(1) **Pre-decrypt hook**. We call `ieee80211_rx_cb()` before the driver decrypts or reassembles frames. This gives us CSI for management and data traffic, encrypted or not.
(2) **Receive ring size**. We enlarge the RX DMA ring to 256 kB, which holds roughly 20 Mbit/s of small frames without overflow.
(3) **Extended timestamp**. The default time-synchronisation field (TSF) is 40 bit. We promote it to 64-bit microsecond ticks so captures from both boards stay aligned for at least one hour.

Each patch is guarded by a `CONFIG_CSI_*` flag in sdkconfig.defaults; running `idf.py menuconfig` leaves them intact.

4.2 Real-Time Host Pipeline

All host logic lives in `scripts/csi_pipeline.py`. Figure 2 shows the path of a packet through the pipeline. Listing 1 lists the four bounded queues that isolate stages.

Listing 1: Queues that decouple pipeline stages.

```
raw_queue    = asyncio.Queue(maxsize=opts.queue_size)
↪ # text lines
packet_queue = asyncio.Queue(maxsize=opts.queue_size)
↪ # CSIPacket
pair_queue   = asyncio.Queue(maxsize=opts.queue_size)
↪ # (master, worker)
result_queue = asyncio.Queue(maxsize=opts.queue_size)
↪ # AoA tuples
```

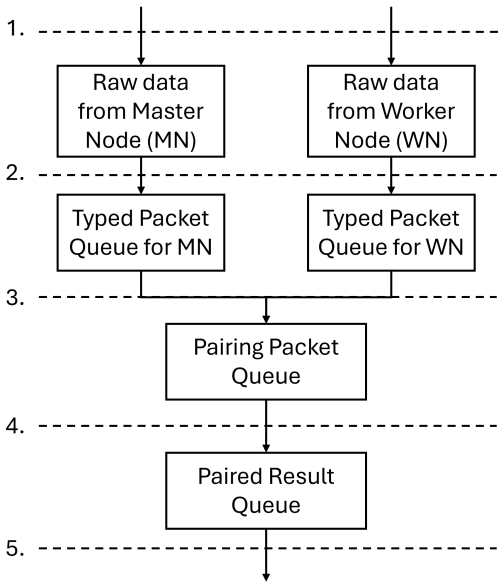

Figure 2: Overview of the real-time host pipeline. The numbering corresponds to the steps described in Section 4.2.

A `Settings` dataclass carries serial ports, baud rate, queue depth, antenna spacing, and output path. Any field can be overridden with an environment variable prefixed `CSI_PIPELINE_` or a command-line flag, making batch experiments repeatable.

Five `asyncio` tasks run in parallel; their numbering matches the steps in Figure 2:

(1) **SerialReader**: Reads bytes from each serial port in a dedicated OS thread, applies COBS framing, and pushes complete lines to `raw_queue`. A 7-bit guard marker lets it regain sync if noise corrupts a frame.
(2) **Parser**: Converts a line to a `CSIPacket`: MAC, RSSI, channel, extended TSF, sequence number, and a 128-tone complex CSI vector. The packet moves to `packet_queue`.
(3) **PairMatcher**: Holds a dictionary keyed by MAC and `seq_ctrl`. When both boards report the same frame, it forwards the pair to `pair_queue`. A garbage collector drops entries older than 5 s; the scan interval is 10 ms.
(4) **AoAEstimator**: Computes the phase difference, unwraps it, subtracts an optional calibration vector, and returns

$$\theta = \arcsin\left(\frac{\lambda\,\bar{\phi}}{2\pi d}\right),$$

where d is antenna spacing and $\lambda = 12.4$ cm at 2.412 GHz.
(5) **CSILogger**: Writes tuples

$$(t,\ \text{MAC},\ \text{seq},\ \theta,\ \text{RSS I}_M,\ \text{RSSI}_W,\ \text{IQ}_M,\ \text{IQ}_W)$$

to a rotating file (10 MB per file, five backups, flush after 8 kB).

Each queue has a fixed length. If a consumer stalls, the preceding `put()` blocks and flow slows automatically. When `raw_queue` is full, we drop a line and increment `SerialReader.dropped`. Pair drops recorded by `PairMatcher` usually point to clock drift or RF loss.

Finally, we write aligned pairs to a CSV file. Each capture group contains:

- `timestamp`: N uint64 microsecond ticks.
- `seq_ctrl`: N uint16 sequence numbers.
- `mac`: N uint48 transmitter addresses.
- `aoa`: N float32 azimuth (rad).
- `rssi`: $N \times 2$ int8, master then worker.
- `csi`: $N \times 2 \times 64$ complex64, master then worker, interleaved real/imag.

File attributes record the Wi-Fi channel, bandwidth, firmware commit hash, and pipeline version.

5 Dataset Preparation and Feature Engineering

Now with the setup explained, we turn our attention to building our dataset for our inference ML algorithms. For simplification, this paper considers a 10-class outdoor classification problem. The dataset for the classification problem is gathered by placing the two ESPs together with 6 cm separation at a fixed location, and then passively measuring Wi-Fi traffic from a target UE. As shown in Figure 3, the target UE is placed in 10 different locations around the two ESPs, with the only wall positioned roughly 10 m behind the ESPs. The device communicating with the target UE is also positioned behind the two ESPs in order to limit the potential for reflected paths. For building the dataset, we had the target device running ping traffic with a buffer size of 1000. With this, we were able to collect about 1500 samples from the target device per ESP, per measurement location. This amount of data was judged sufficient for our classification problem, as we have about 130 features, which leaves each class with about 10 times more samples than features. The placements of the target UE are at −90, −45, 0, 45, and 90 degrees with respect to the ESPs. Two measurements are gathered from each of the aforementioned angles: one from 2 meters away from the ESPs and one from 5 meters away.

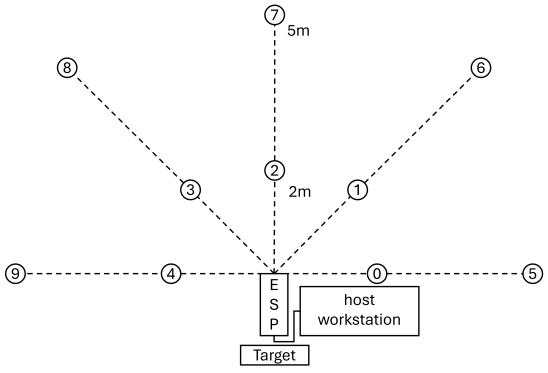

Figure 3: Experimental data-collection setup. The numbering indicates the 10 target UE locations used for dataset generation.

For each packet, the following relevant features are captured from both ESPs:

(1) 64 CSI IQ samples from both ESPs,
(2) RSSI value,
(3) Timestamp,
(4) Sequence control number,
(5) MAC address.

As explained in Section 4.2, the data from both ESPs is matched using the timestamp and the sequence control number.

6 Classification using Machine Learning

This section explores the application of various machine learning models for the classification. The objective is to classify the scenario, defined by a combination of:

- Distance: 2m or 5m
- Angle of Arrival (AoA): ±90°, ±45°, or 0°

These labels form a 10-class classification problem (labels 0–9), as outlined below:

- 2m + 90° → Label 0
- 2m + 45° → Label 1
- 2m + 0° → Label 2
- 2m - 45° → Label 3
- 2m - 90° → Label 4
- 5m + 90° → Label 5
- 5m + 45° → Label 6
- 5m + 0° → Label 7
- 5m - 45° → Label 8
- 5m - 90° → Label 9

The following five machine learning models were trained and evaluated:

- neural network (NN)
- K-Nearest Neighbors (KNN)
- Support Vector Machine (SVM)
- Random Forest (RF)
- Long Short-Term Memory (LSTM)

Each model was tested using different feature sets including raw CSI, extracted features (e.g., statistical summaries), RSSI and AoA. The primary goal was to assess how each model handles varying data modalities and complexities, compare performance across raw versus engineered features and identify the most effective model and feature combination for the classification.

6.1 neural network

This section goes over using a neural network for classification. In addition, the section gives an overview of the performance of the different available features.

A neural network (NN) is chosen as one of the machine learning models. NNs are widely used in similar problems. For example in [23] different NNs are used for localization.

The chosen NN is a Deep NN with 5 hidden layers. The input features of the NN are: the RSSI values and CSI samples from the ESPs, the calculated AoA, and their combinations. The output of the NN is the energies for each of the classes, in this case locations. The location with the highest energy is then chosen as the predicted location. In training, the ground-truth locations are used as the labels. 70/20/10 training, testing, and validation splits are used.

Some relevant parameters of the used NN:

- Layer sizes: [Length of the input feature, 256, 128, 64, 32, 16, 10].

Table 4: Performance of NN Classifier

	AoA	RSSI	CSI	CSI+RSSI+AoA
Accuracy	13.0%	56.0%	88.6%	**90.4%**

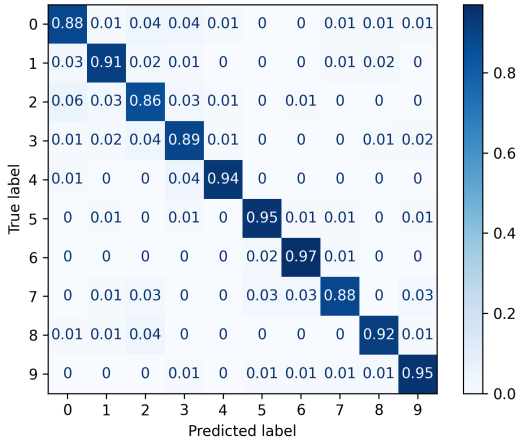

Figure 4: Confusion Matrix of NN

- Batch size: 32 features
- Loss function: Cross-Entropy loss
- Optimizer: Stochastic Gradient Descent
- Number of epochs: 100

Different hyper-parameters were considered with respect to accuracy performance. The hyper-parameters were tuned by keeping the aforementioned NN setup and tuning one parameter at a time.

For example, using different layer setups had minimal effect on the performance. Different batch sizes were found to affect the model performance. Using a small batch size of 4 is too specific as the batches can vary a lot, and thus the weight adjustments between batches are too specific. Additionally, using a larger batch size of 128, the batches appear to be too general, and thus the model is unable to learn the intricacies of the data. Two optimizers Stochastic Gradient Descent and Adaptive Moment Estimation (Adam) were evaluated. Both achieved similar performance. Minimum Square Error (MSE) was also considered as a loss function. However, with the same amount of epochs, MSE-loss performed worse, and was much more slow to learn. Finally, no significant performance gain was achieved with having more than one hundred epochs.

The accuracy values of the NN with different input features are given in table 4. The accuracy values are the averages of five separate trainings and subsequent test instances of the same NN.

The best accuracy of the NN 90.4% is achieved with using all the available features. Individual analysis of the features reveals that the CSI is the best, achieving only 1.8 percentage points worse accuracy score compared to using all the features. The calculated AoA performs worse, achieving an accuracy of 13.0%.

The normalized confusion matrix of the NN is presented in figure 4.

All classes achieve class accuracies of over 85%. The worst performing class (0°, 2m), achieves a class accuracy of 86%. The class is most often mistaken for the location (90°, 2m) happening 5% of the time.

6.1.1 ***Cross-testing with another dataset.*** To evaluate the generality of the NN model, the network was trained with the presented dataset but tested using another similar dataset. The second dataset comprised of 150 samples for each of the same ten classes. However, the location of the measurements differed. In the second dataset, the ESP32-s3s were situated outside in a parking lot.

The results of the cross-testing are listed in Table 5:

Table 5: Performance of NN Classifier: Cross-testing

	AoA	RSSI	CSI	CSI+RSSI+AoA
Accuracy	8.7%	12.4%	12.5%	**13.2%**

The results show that the trained model performs poorly when tested with another dataset. This suggests that the CSI data and other features between the two datasets do not correlate well.

6.1.2 ***Shift Towards Feature-Based Representation.*** After evaluating raw CSI-based classification with deep neural networks, we introduced an alternative strategy: extracting statistical and spectral features from the complex IQ samples. This approach reduces input dimensionality, improves interpretability and enables the use of classical machine learning models with strong performance.

The transformation converts raw IQ vectors into a 69-dimensional feature vector, derived from signal amplitude, phase, and metadata. These features aim to retain critical spatial and frequency characteristics of the wireless channel while improving robustness to noise and reducing model complexity.

Feature Categories and Breakdown

a. Time-Domain Statistical Features (44 features)

To summarize the temporal behavior of the CSI signals, we compute 11 statistical descriptors over four key signal vectors, resulting in a total of $4 \times 11 = 44$ features. The four signals are:

- Amplitude (Master): $|\mathrm{IQ}_{\mathrm{master}}|$
- Amplitude (Worker): $|\mathrm{IQ}_{\mathrm{worker}}|$
- Amplitude Difference: $|\mathrm{IQ}_{\mathrm{master}}| - |\mathrm{IQ}_{\mathrm{worker}}|$
- Unwrapped Phase Difference: $\mathrm{unwrap}(\angle \mathrm{IQ}_{\mathrm{master}} - \angle \mathrm{IQ}_{\mathrm{worker}})$

For each of these signals, the following statistical features are computed:

(1) **Mean:** The average value of the signal, capturing its central tendency.
(2) **Standard Deviation:** Measures the spread of values around the mean, indicating signal variability.
(3) **Variance:** The squared standard deviation; useful in ML models sensitive to scale.
(4) **Minimum:** The smallest value in the signal window.
(5) **Maximum:** The largest value in the signal window.
(6) **Peak-to-Peak (Range):** The difference between the max and min, indicating the dynamic range.
(7) **Median:** The middle value when the signal is sorted, offering a robust central tendency.

(8) **25th Percentile (Q1):** The value below which 25% of the samples fall.
(9) **75th Percentile (Q3):** The value below which 75% of the samples fall.
(10) **Skewness:** Measures the asymmetry of the signal's distribution. A skew of 0 implies a symmetric distribution.
(11) **Kurtosis:** Measures the "tailedness" of the distribution. High kurtosis indicates outlier-prone signals.

These statistics capture both first-order and higher-order characteristics of the signal, enabling models to distinguish between different spatial patterns of Wi-Fi propagation and multipath behavior.

b. Correlation Metrics (2 features)

These features assess the similarity between signal patterns received at the Master and Worker nodes:

(1) **Amplitude Correlation:** Pearson correlation coefficient between the amplitude vectors of the Master and Worker nodes. High correlation suggests similar multipath profiles.
(2) **Phase Correlation:** Pearson correlation coefficient between the raw phase values of the Master and Worker IQ signals. This reflects alignment or disparity in phase evolution.

Correlation metrics are useful for identifying signal symmetry or divergence due to user angle and environmental geometry.

c. Frequency-Domain Features (14 features)

Using Welch's method, we estimate the Power Spectral Density (PSD) of amplitude signals to reveal periodicity and energy distribution. These features are computed separately for:

- Amplitude of Master node
- Amplitude of Worker node

For each, the following 7 features are extracted:

(1) **Mean PSD:** Average power across the spectrum.
(2) **PSD Standard Deviation:** Indicates spectral spread.
(3) **Maximum PSD Value:** Peak spectral power, often associated with dominant reflections.
(4) **Dominant Frequency Index:** Frequency bin with the highest power; may relate to user motion or interference.
(5) **Low-Frequency Energy Ratio (0–25%):** Proportion of total spectral energy concentrated in the lowest quarter of the spectrum.
(6) **High-Frequency Energy Ratio (50–100%):** Energy in the upper half of the frequency range, often linked to rapid channel changes.
(7) **Very Low-Frequency Energy Ratio (0–10%):** Concentration of near-DC components, indicative of static or slowly-varying signals.

These features provide frequency-based insights into spatial dynamics and help models learn signal periodicity and environmental clutter.

d. Phase Relationship Features (6 features)

These features describe the behavior of the unwrapped phase difference between Master and Worker signals:

(1) **Mean of Absolute Phase Difference:** Captures the average angular offset between nodes.
(2) **Standard Deviation of Absolute Phase Difference:** Indicates variation in that offset.
(3) **Mean of Cosine of Phase Difference:** Highlights alignment and directionality (in-phase vs out-of-phase).
(4) **Standard Deviation of Cosine:** Variability in angular alignment.
(5) **Mean of Sine of Phase Difference:** Captures orthogonal phase shifts.
(6) **Standard Deviation of Sine:** Measures phase jitter or fluctuation.

These features are especially relevant for detecting angle-of-arrival changes and phase-based distinctions across spatial positions.

e. Metadata Features (5 features)

These are directly extracted from packet metadata or derived from them:

(1) **Master RSSI:** Received Signal Strength Indicator at the Master node (in dBm).
(2) **Worker RSSI:** Same, but at the Worker node.
(3) **RSSI Difference (Master − Worker):** Captures relative strength which can reflect orientation or distance differences.
(4) **Sequence Control Number:** Packet sequence index; useful for time-series alignment or anomaly detection.
(5) **Angle of Arrival (AoA):** Rough angle estimate derived from CSI phase difference.

These features provide coarse spatial context and temporal identifiers to complement CSI-derived information.

6.2 K-Nearest Neighbours

The K-Nearest Neighbors (KNN) algorithm is a simple, yet effective, non-parametric method used for classification and regression. It makes predictions based on the majority class among the k closest data points in the feature space [2]. Key features of KNN used are listed below:

- Distance-based non-parametric algorithm.
- Classifier: `KNeighborsClassifier()` from `sklearn.neighbors`.
- The model's performance is optimized via `RandomizedSearchCV` over several distance metrics and neighbor settings.
- Best hyper parameters for Extracted features: `n_neighbors = 7` and weight = 'distance'.
- Best hyper parameters for Raw CSI: `n_neighbors = 3`, weight = 'distance' and p = 2.

Table 6 shows the accuracy achieved using KNN classifier.

Table 6: Performance of KNN Classifier

	AoA	RSSI	CSI	CSI+RSSI+AoA
Accuracy	12.96%	51.0%	59.94%	**66.76%**

Table 7 shows the comparison of model performance using Raw CSI and Extracted features.

Table 7: Performance of KNN using CSI extracted features

	CSI+RSSI+AoA	Extracted Features+RSSI+AoA
Accuracy	66.76%	**74.88%**

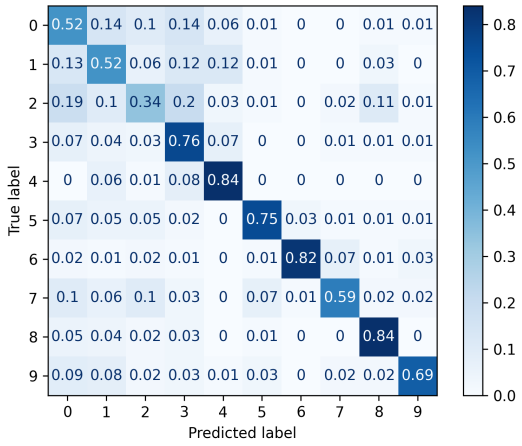

Figure 5: Confusion Matrix of KNN

The normalized confusion matrix of KNN is presented in figure 5.

The classification accuracy was higher when using extracted features (mean, standard deviation, etc.) combined with RSSI and AoA information, achieving 73.60%, compared to 66.79% when using raw CSI data with RSSI and AoA.

This improvement is likely due to the denoising and dimensionality reduction effects of feature extraction. Raw CSI data can be noisy and high-dimensional, which may obscure discriminative patterns. In contrast, statistical features summarize signal characteristics more compactly and consistently, enabling the KNN classifier to perform better in identifying class boundaries. Additionally, weighting neighbors by distance helped emphasize closer (and often more relevant) training samples in both configurations.

6.3 Support Vector Machine

Support Vector Machines (SVM) are supervised learning models that construct optimal hyperplanes to separate data into distinct classes with maximum margin. They are particularly effective in high-dimensional spaces and are robust to overfitting [8]. Key aspects of SVMs used in this study are:

- Classifier: `SVC()` from `sklearn.svm`.
- Uses kernelized version of SVM with probabilistic outputs enabled.
- Optimized using `RandomizedSearchCV` for various kernels, regularization values, and class balancing.
- Best hyper parameters found: `C=10.0`, `gamma = "scale"`, `kernel = "rbf"`, `class_weight = "balanced"`, `probability = True`.
- Set `random_state = 42` for reproducibility.

The performance achieved using SVM is shown in table 8 :

Table 9 shows a comparison of how much difference would it cause if we used extracted features for training instead of raw CSI.

The normalized confusion matrix of SVM is presented in figure 6.

Table 8: Performance of SVM Classifier

	AoA	RSSI	CSI	CSI+RSSI+AoA
Accuracy	13.42%	57.73%	84.11%	**88.85%**

Table 9: Performance of SVM using CSI extracted features

	CSI+RSSI+AoA	Extracted Features+RSSI+AoA
Accuracy	**88.85%**	86.38%

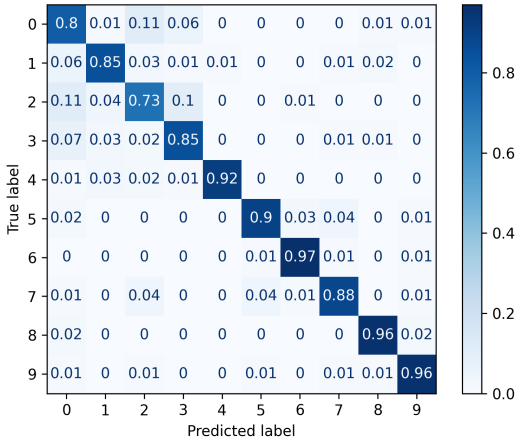

Figure 6: Confusion Matrix of SVM

Interestingly, the SVM classifier achieved higher accuracy when trained on raw CSI data combined with RSSI and AoA, reaching 88.14%, compared to 81.97% using extracted statistical features.

This result suggests that SVMs are well-suited to handling high-dimensional input spaces like raw CSI, especially when using non-linear kernels such as the Radial Basis Function (RBF). The raw CSI likely retains subtle spatial and phase-related information that is lost during feature extraction. By leveraging this detailed structure, SVM can better model complex class boundaries. In contrast, extracted features may oversimplify the data, reducing discriminative capacity in the case of SVMs.

6.4 Random Forest

Random Forest is an ensemble learning method that builds multiple decision trees and merges their outputs to improve classification accuracy and control overfitting. It is known for its robustness and ability to handle large datasets with higher dimensionality [5]. Key features of RF are:

- Classifier: `RandomForestClassifier()` from `sklearn.ensemble`.
- Ensemble of decision trees with bootstrap sampling.
- Capable of estimating feature importance.
- Optimized using `RandomizedSearchCV` across a broad hyperparameter space.

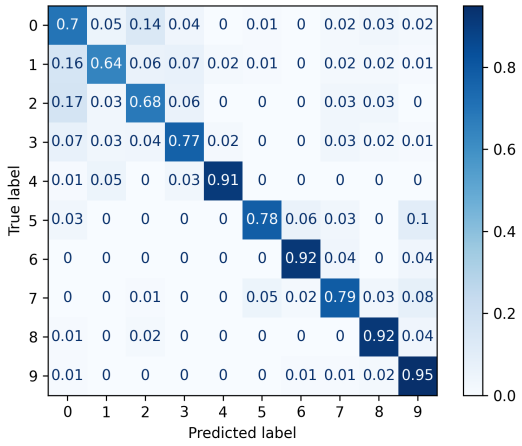

Figure 7: Confusion Matrix of RF

- Best hyper parameters found: `n_estimators = 300`, `random_state = 42`, `max_depth = 20`, `min_samples_leaf = 10`, `max_features = 0.2`, `n_jobs = -1`
- Set `random_state = 42` for reproducibility.

The performance achieved using RF is shown in table 10 :

Table 10: Performance of RF Classifier

	Aoa	RSSI	CSI	CSI+RSSI+AoA
Accuracy	12.66%	57.63%	68.85%	**81.44%**

The performance comparison between using Raw CSI and Extracted features in RF is shown in table 11 :

Table 11: Performance of RF using CSI extracted features

	CSI+RSSI+AoA	Extracted Features+RSSI+AoA
Accuracy	81.44%	**84.54%**

The normalized confusion matrix of RF is presented in figure 7.

The Random Forest classifier performed slightly better with raw CSI data (80.55%) compared to extracted features (77.00%), when both were combined with RSSI and AoA. This suggests that the ensemble of trees was able to effectively capture complex patterns in the high-dimensional raw CSI data, which may retain more spatial and temporal nuances than hand-engineered features.

However, the performance gap is narrower than in the case of SVM, likely because decision trees are inherently less sensitive to raw feature scales and more resilient to noise. While extracted features may help reduce dimensionality and noise, they might also discard subtle variations that are useful for classification in a spatially complex signal environment.

6.5 Long Short-Term Memory

Long Short-Term Memory (LSTM) networks are a type of recurrent neural network capable of learning long-term dependencies by using memory cells to store information over time. They have proven effective in sequence prediction tasks such as speech and language modeling [17].

Long Short-Term Memory (LSTM) based deep learning model is employed to classify time-series data extracted from raw sensor signals. The architecture is enhanced with attention mechanisms, batch normalization and dropout layers to improve generalization and performance. The implementation is done using `PyTorch`. Below, we describe the key components of the pipeline.

6.5.1 ***Data Preprocessing and Feature Extraction.*** The raw data is read from a CSV file, where each row contained scalar features (`master_rssi`, `worker_rssi`, and `aoa`) along with IQ sequences from master and worker devices. These are extracted and concatenated into a single feature vector per row.

To prepare the time-series input for LSTM, sequences of length 40 were generated with an overlap of 20. Each sequence corresponds to a sliding window over the data with a step size of $40 - 20 = 20$.

- Feature vectors are grouped by class.
- Sequential data samples are created from each class to maintain class balance.
- Labels are encoded using `LabelEncoder` for compatibility with `CrossEntropyLoss`.

6.5.2 ***Dataset Splitting and Scaling.*** A stratified split strategy is used to divide the dataset into 70% training, 10% validation, and 20% test subsets, ensuring class balance in each split.

Robust scaling is applied to account for outliers. The scaler is fitted only on training data and applied uniformly across validation and test sets.

6.5.3 ***Model Architecture.*** The LSTM model is implemented as a PyTorch `nn.Module`. It incorporates the following layers:

- A multi-layer bidirectional LSTM with 3 layers and 128 hidden units.
- Batch normalization applied after the LSTM output to stabilize training.
- A multi-head self-attention layer with 8 heads to allow the model to focus on relevant time steps.
- A fully connected classifier composed of:
  - Two dense layers with ReLU activation and batch normalization.
  - Dropout layers for regularization (dropout rate = 0.5).
  - Final dense layer mapping to the number of output classes.

The input to the model is a tensor of shape `(batch_size, sequence_length, feature_dim)`. The output is a probability distribution over the classes.

6.5.4 ***Training Procedure.*** The model is trained using the Adam optimizer with weight decay ($1e^{-5}$) and a learning rate of 0.0005. A learning rate scheduler (`ReduceLROnPlateau`) was used to adapt the learning rate during training.

The training loop includes:

- Gradient clipping (max norm = 1.0) to avoid exploding gradients.

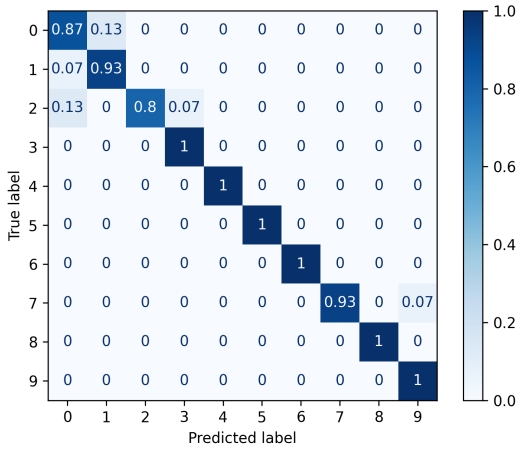

Figure 8: Confusion Matrix of LSTM

- Early stopping with a patience of 10 epochs to prevent overfitting.
- Tracking of training and validation loss, and saving the best model based on validation accuracy.

6.5.5 ***Model Evaluation***. After training, the model is evaluated on the held-out test set. The best validation model was loaded and used for inference. Accuracy was computed as the primary metric, alongside a classification report.

To detect potential data leakage or overfitting, the average cosine similarity between the first 100 sequences was computed. If high similarity was observed, warnings were issued suggesting changes to overlap, data collection strategy, or model validation.

The performance achieved using LSTMs is shown in table 12 :

Table 12: Performance of LSTM

	Aoa	CSI	RSSI	CSI+RSSI+AoA
Accuracy	15.33%	93.24%	64%	**95.27%**

Table 13 shows a comparison of LSTM performance as we used extracted features for training.

Table 13: Performance of LSTM using CSI extracted features

	CSI+RSSI+AoA	Extracted Features+RSSI+AoA
Accuracy	95.27%	**97.97%**

The normalized confusion matrix of LSTM is presented in figure 8.

7 Experimental Evaluation

It is essential to evaluate the performance of multiple classification models to determine how effectively different machine learning approaches can leverage Channel State Information (CSI), Received Signal Strength Indicator (RSSI) and Angle of Arrival (AoA) to classify a user's location across ten defined spatial classes. This section also explores the influence of feature engineering, model complexity and temporal signal structure on model training and performance.

7.1 Accuracy of different methods

The accuracies of the different models are gathered in table 14. This table lists the best accuracies gained by each model. Among all evaluated techniques, LSTM networks achieved the highest accuracy **(97.97%)**, significantly outperforming traditional ML models. The bidirectional, attention-augmented LSTM architecture exploited temporal dependencies in the raw CSI sequences that static models could not capture. This suggests that sequential dynamics—even in what seems like spatial classification—contain discriminative information possibly tied to subtle variations in the radio environment or device timing.

In contrast, the neural network (feedforward DNN) achieved a commendable **90.4%**accuracy. Though static in structure, the DNN could still exploit high-dimensional correlations in the CSI and RSSI inputs. However, it lacked the sequence modeling ability of LSTM, which likely explains the performance gap. Traditional models—KNN, SVM, and RF—lagged behind in absolute performance but merit attention for their interpretability and lower computational demands. Notably:

- KNN, despite its simplicity, achieved **74.88%** accuracy when combined with hand-engineered features. This reinforces the effectiveness of the statistical and spectral features in reducing noise and dimensionality.
- SVM reached **88.85%** accuracy using raw CSI, showing its strength in handling high-dimensional, non-linear data when tuned properly with RBF kernels.
- RF performed moderately well **(84.54%)** and proved robust to noise and overfitting, thanks to its ensemble nature.

A nuanced trend emerged in how different models responded to raw CSI versus engineered features. For KNN, engineered features improved accuracy by approximately 8%, demonstrating how noise reduction and dimensionality compression enhance performance in distance-based classifiers. Silimarly, Random Forest performed slightly better with extracted features (84.54% vs. 81.44%). In contrast, SVM performed better with raw CSI (88.85% vs. 86.38%, indicating that SVMs can exploit the richer spatial and phase-related information in unprocessed data when properly tuned. LSTM models benefited from both input types, achieving 95. 27% with raw CSI alone, and improving further to 97. 97% with engineered features, highlighting that even deep-sequence models gain from structured, interpretable inputs that emphasize key signal characteristics.

While the LSTM clearly outperforms all others, it comes at a higher computational cost due to sequence modeling, attention layers, and training time. The DNN is a strong alternative when temporal coherence is less critical or when real-time inference is needed on edge devices with limited memory. The traditional ML methods are still viable for embedded or constrained applications. Their interpretability and faster training cycles make them useful for prototyping and benchmarking, even if they sacrifice some accuracy.

Moreover, the marginal improvement from feature engineering in deep models (e.g., LSTM: 95.27% to 97.97%) suggests a hybrid path forward: using domain knowledge to craft informative features, then leveraging deep architectures to maximize generalization and pattern recognition.

Table 14: Performance Comparison

	NN	KNN	SVM	RF	LSTM
Accuracy	90.4%	74.88	88.85	84.54	**97.97**

7.2 Feature Ablation and Sensitivity Analysis across models

To evaluate the contribution of each feature type, we performed systematic feature ablation studies, removing one or more inputs—CSI, RSSI, and AoA—from the model inputs and observing the resulting impact on classification accuracy. These experiments were repeated across multiple models, including both deep and traditional machine learning approaches.

- **AoA alone has limited discriminative power.** Angle of Arrival (AoA), derived from phase differences, performed poorly as a sole feature. AoA was caluclated directly from the Raw CSI from the device using mathermatical formulas. However, the values for AoA appeared to be quite random. A simple model like KNN achieved merely **13%** accuracy using AoA in isolation—close to random guessing in a 10-class setup. However, when combined with CSI and RSSI, AoA contributed modest performance improvements by resolving directional ambiguities.
- **RSSI alone is insufficient.** When models were trained using only RSSI values from the Master and Worker nodes, accuracy dropped significantly. The neural network, for instance, only achieved **56%** with RSSI alone. This confirms that while RSSI may provide rough distance cues, it lacks the spatial resolution necessary for fine-grained classification.
- **Extracted statistical features improve interpretability and efficiency.** For models like K-Nearest Neighbors (KNN), using engineered statistical features (e.g., mean, variance, skewness) improved accuracy by approximately **7%** compared to raw CSI inputs. This suggests that dimensionality reduction and feature summarization help traditional models by reducing noise and improving generalization.
- **CSI is the most crucial feature for accurate localization and classification.** CSI captured the detailed spatial signature of the channel. While RSSI and AoA alone are insufficient, they offer useful auxiliary signals when combined with CSI. Hand-engineered features are particularly beneficial for classical models like KNN and Random Forest, and even temporal models like LSTM see improved performance with extracted features—achieving the highest accuracy of **97.97%**. This suggests that thoughtfully engineered features not only support traditional methods but also complement deep learning architectures in capturing both spatial and temporal signal characteristics.

7.3 Discussion of Limitations

Despite the high accuracies observed, several limitations were identified in our experiments:

- **Site-Specificity:** The trained models are appear signs of being site-specific. Patterns learned from the CSI/RSSI measurements in one environment do not generalize well to a different location. In cross-testing (training on dataset that was gathered on one site and evaluating on another), we observed a drastic drop in accuracy, indicating that the models might have captured environment-specific multipath signatures rather than universal features.
- **Hyperparameter Tuning Time:** Training and tuning the models, especially the NN and LSTM, required extensive computational effort. Grid searches over hyperparameters (e.g., number of layers, neurons, learning rates) were time-consuming. Randomized Search was hence used to hypertune the parameters. This poses a practical challenge for real-time deployment or rapid adaptation to new scenarios.
- **Timestamps unused in modeling:** While CSI timestamps were part of the dataset, they were not explicitly used as input features. Surprisingly, LSTM still outperformed others, suggesting it captured time-like structure from sequences of IQ samples. However, future work could explore directly incorporating temporal indices or time-deltas for potentially even better performance.
- **Noisy AoA estimates:** The AoA feature derived from CSI phase differences lacked precision due to imperfect time synchronization and non-coherent hardware.

These limitations underscore the challenges of WiFi-based localization using CSI/RSSI data. Future work should address the need for more robust features, adaptive learning for different sites and methods to reduce training complexity.

8 Conclusion

Our work presents a low-cost, machine learning–driven indoor localization system using synchronized ESP32-S3 modules to passively capture Channel State Information (CSI). A custom data pipeline enabled synchronized CSI collection from two nodes, forming the basis for a ten-class static classification task. Five machine learning models were evaluated and LSTMs achieved the highest accuracy at 97.97%, leveraging temporal dependencies in CSI sequences. Notably, the feedforward neural network also performed strongly, reaching 90.4% accuracy, highlighting its suitability for environments where temporal modeling is less critical.

While effective in static settings, the current system does not address mobility or environmental variability. Future work will focus on incorporating dynamic scenarios through temporal modeling, improving generalization across locations via domain adaptation, and enhancing synchronization for accurate angle-of-arrival estimation. Overall, this work demonstrates that high-accuracy localization is achievable using commodity Wi-Fi hardware and machine learning offering a scalable, cost-effective alternative to traditional positioning systems.

References

[1] Fadel Adib and Dina Katabi. 2013. See Through Walls with Wi-Fi!. In *Proceedings of the ACM SIGCOMM 2013 Conference (SIGCOMM '13)*. ACM, Hong Kong, China, 75–86.

[2] Naomi S Altman. 1992. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician* 46, 3 (1992), 175–185.

[3] Arista Networks. 2024. *Wi-Fi 7: A Leap Towards Time-Sensitive Networking*. Technical Report. https://www.arista.com/assets/data/pdf/Whitepapers/Arista-Wi-Fi-7-White-Paper.pdf

[4] Oscar Bejarano, Edward W. Knightly, and Minyoung Park. 2013. IEEE 802.11ac: From Channelization to Multi-User MIMO. *IEEE Communications Magazine* 51, 10 (2013), 84–90. doi:10.1109/MCOM.2013.6619570

[5] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[6] Hao Chen, Yifan Zhang, Wei Li, Xiaofeng Tao, and Ping Zhang. 2017. ConFi: Convolutional Neural Networks Based Indoor Wi-Fi Localization Using Channel State Information. *IEEE Access* 5 (2017), 18066–18074. doi:10.1109/ACCESS.2017.2749516

[7] Leif Claesson. 2021. ESP1588: IEEE-1588 Precision Time Protocol (PTP) Client for ESP8266/ESP32. https://github.com/leifclaesson/ESP1588. Accessed 2025-05-28.

[8] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine Learning* 20, 3 (1995), 273–297.

[9] Saandeep Depatla, Lucas Buckland, and Yasamin Mostofi. 2015. X-Ray Vision with Only WiFi Power Measurements Using Rytov Wave Models. *IEEE Transactions on Vehicular Technology* 64, 4 (apr 2015), 1376–1387.

[10] Rui Du, Haocheng Hua, Hailiang Xie, Xianxin Song, Zhonghao Lyu, Mengshi Hu, Narengerile, Yan Xin, Stephen McCann, Michael Montemurro, Tony Xiao Han, and Jie Xu. 2025. An Overview on IEEE 802.11bf: WLAN Sensing. *IEEE Communications Surveys & Tutorials* 27, 1 (2025), 184–217. doi:10.1109/COMST.2024.3408899

[11] Rui Du, Hailiang Xie, Mengshi Hu, Yan Xin, Stephen McCann, Michael Montemurro, Tony Xiao Han, and Jie Xu. 2022. An Overview on IEEE 802.11bf: WLAN Sensing. *arXiv preprint* arXiv:2207.04859 (2022). https://arxiv.org/abs/2207.04859

[12] Espressif Systems. 2021. ESP32 Wi-Fi Channel State Information Guide. ESP-IDF Programming Guide. Available online: docs.espressif.com (retrieved 2025-04-29).

[13] Espressif Systems. 2023. *ESP32 Hardware Design Guidelines*. Accessed 2025-05-28.

[14] Florian Euchner, Tim Schneider, Marc Gauger, and Stephan ten Brink. 2025. ESPARGOS: An Ultra-Low-Cost, Real-Time-Capable Multi-Antenna Wi-Fi Channel Sounder. *arXiv preprint arXiv:2502.09405* (2025).

[15] Daniel Halperin, Wenjun Hu, Anmol Sheth, and David Wetherall. 2011. Tool Release: Gathering 802.11n Traces with Channel State Information. *ACM SIGCOMM Computer Communication Review* 41, 1 (2011), 53.

[16] Steven M. Hernandez and Ertan Bulut. 2022. WiFi Sensing on the Edge: Signal Processing Techniques and Challenges for Real-World Systems. *IEEE Communications Surveys & Tutorials* 24, 3 (2022), 1821–1858. doi:10.1109/COMST.2022.3143810

[17] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9, 8 (1997), 1735–1780.

[18] IEEE 802.11 WG. 2024. IEEE Std 802.11be™–2024. https://www.ieee802.org/11/. Approved 26 Sep 2024; publication pending.

[19] IEEE 802.11 WG. 2025. Official IEEE 802.11 Project Timelines. https://www.ieee802.org/11/Reports/802.11_Timelines.htm. Accessed 13 May 2025.

[20] IEEE 802.11az TG. 2017. Pre-Association Security Negotiation (PASN) for 11az. https://mentor.ieee.org/802.11/dcn/17/11-17-1737-00-00az-pre-association-security-negotiation-for-11az.pptx. Accessed 13 May 2025.

[21] IEEE Standards Association. 2024. Newly Released IEEE 802.11az Standard Improving Wi-Fi Location Accuracy. https://standards.ieee.org/beyond-standards/newly-released-ieee-802-11az-standard-improving-wi-fi-location-accuracy-is-set-to-unleash-a-new-wave-of-innovation/. Accessed 13 May 2025.

[22] Evgeny Khorov, Anton Kiryanov, Andrey Lyakhov, and Giuseppe Bianchi. 2019. A Tutorial on IEEE 802.11ax High Efficiency WLANs. *IEEE Communications Surveys & Tutorials* 21, 1 (2019), 197–216. doi:10.1109/COMST.2018.2871099

[23] Xinze Li, Hanan Al-Tous, Salah Eddine Hajri, and Olav Tirkkonen. 2024. Channel Covariance based Fingerprint Localization. In *2024 IEEE 100th Vehicular Technology Conference (VTC2024-Fall)*. 1–7. doi:10.1109/VTC2024-Fall63153.2024.10757910

[24] MINTS Project. 2023. An Introduction to Wi-Fi Sensing and IEEE 802.11bf. https://b5g-mints.eu/blog36/. Accessed 13 May 2025.

[25] Eldad Perahia and Robert Stacey. 2013. *Next Generation Wireless LANs: 802.11n and 802.11ac* (2nd ed.). Cambridge University Press.

[26] Pablo Picazo-Martínez, Carlos Barroso-Fernández, José Martín-Pérez, Mikhail Groshev, and Antonio de la Oliva. 2023. IEEE 802.11az Indoor Positioning with mmWave. *arXiv preprint* arXiv:2303.05996 (2023). https://arxiv.org/abs/2303.05996

[27] Thibaut Ropitault, Sergio Blandino, Anirban Sahoo, and Nada Golmie. 2023. IEEE 802.11bf: Enabling the Widespread Adoption of Wi-Fi Sensing. *IEEE Communications Standards Magazine* (2023). https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=935175

[28] Matthias Schulz, Daniel Wegemer, and Matthias Hollick. 2017. Nexmon: Build Your Own Wi-Fi Testbeds with Low-Level MAC and PHY Access Using Firmware Patches on Off-the-Shelf Mobile Devices. In *Proc. 11th ACM Workshop on Wireless Network Testbeds, Experimental Evaluation & Characterization (WiNTECH)*. 33–40. doi:10.1145/3131473.3131476

[29] Simon Tewes and Aydin Sezgin. 2021. WS-WiFi: Wired Synchronization for CSI Extraction on COTS-WiFi Transceivers. *IEEE Internet of Things Journal* 8, 11 (2021), 9099–9108. doi:10.1109/JIOT.2021.3058179

[30] Hao Wang, Daqing Zhang, Yasha Wang, Junyi Ma, Yuxiang Wang, and Shengjie Li. 2017. RT-Fall: A Real-Time and Contactless Fall Detection System with Commodity WiFi Devices. *IEEE Transactions on Mobile Computing* 16, 2 (2017), 511–526. doi:10.1109/TMC.2016.2557795

[31] Wei Wang, Alex X. Liu, and Moustafa Shahzad. 2016. Gait Recognition Using WiFi Signals. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '16)*. ACM, Heidelberg, Germany, 363–373.

[32] Wi-Fi Alliance. 2024. *Wi-Fi Certified 7 Release 1 Launch*. https://www.fierce-network.com/tech/wi-fi-alliance-unveils-wi-fi-certified-7

[33] Chenhao Wu, Xuan Huang, Jun Huang, and Guoliang Xing. 2023. Enabling Ubiquitous Wi-Fi Sensing with Beamforming Reports. In *Proc. ACM SIGCOMM*. 20–32. doi:10.1145/3603269.3604817

[34] Yaxiong Xie, Zhenjiang Li, and Mo Li. 2015. Precise Power Delay Profiling with Commodity WiFi. In *Proc. of ACM MobiCom*. 53–64.

[35] Yunze Zeng, Parth H. Pathak, and Prasant Mohapatra. 2016. WiWho: WiFi-Based Person Identification in Smart Spaces. In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. 1–12. doi:10.1109/IPSN.2016.7460727

[36] C. Zhang, A. Raissinia, and R. de Vegt. 2021. *Qualcomm® Wi-Fi Ranging: Delivering Ranging and Location Technologies of Tomorrow Today*. Technical Report. Qualcomm Technologies Inc. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/qualcomm-wi-fi-ranging-white-paper.pdf

9 Appendices

A Code Listings

The code used in this report is stored in the private repository:

`https://github.com/henryblu/ML-Powered-WiFi-Tracker`

Access to the repository can be provided upon request. Please contact the corresponding author at the email address provided in the report.

```
import torch.nn as nn

n_class = 10
layers = [256,128,64,32,16] # layer sizes
len_feature = 259 # 128*2 (2xCSI) + 2 (2xRSSI) + 1 (AoA)

shared_layers = nn.Sequential(
    nn.Linear(len_feature, layers[0], bias=True),
    nn.ReLU(),
    nn.BatchNorm1d(layers[0]),
    nn.Linear(layers[0], layers[1], bias=True),
    nn.ReLU(),
    nn.BatchNorm1d(layers[1]),
    nn.Linear(layers[1], layers[2], bias=True),
    nn.ReLU(),
    nn.BatchNorm1d(layers[2]),
    nn.Linear(layers[2], layers[3], bias=True),
    nn.ReLU(),
    nn.BatchNorm1d(layers[3]),
    nn.Linear(layers[3], layers[4], bias=True),
    nn.ReLU(),
    nn.BatchNorm1d(layers[4]),
    nn.Linear(layers[4], n_class, bias=True) # Last dim is
↪ the amount classes / locations
)

class My_NN(nn.Module):
```

```
        def __init__(self):
            super(My_NN, self).__init__()
            self.block = shared_layers

        def _initialize_weights(self):
            for m in self.modules():
                if isinstance(m, nn.Linear):
                    nn.init.kaiming_uniform_(m.weight)
                    nn.init.zeros_(m.bias)
                elif isinstance(m, nn.BatchNorm1d):
                    nn.init.ones_(m.weight)
                    nn.init.zeros_(m.bias)

        def forward(self, feature):
            output = self.block(feature)

            return output
# Initialize the model and other hyperparameters
my_nn = My_NN()
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(my_nn.parameters(), lr=0.001, momentum
↪ =0.9)
```

Listing 2: Neural network structure

```
#SEQUENCE HANDLING
def load_sequences(file_path, sequence_length, step):
    df = pd.read_csv(file_path)

    features_list = []
    label_list = []

    print("Extracting features from all rows...")
    for _, row in df.iterrows():
        features = load_raw_features(row)
        if features is not None:
            features_list.append(features)
            label_list.append(int(row["label"]))

    print(f"Successfully extracted {len(features_list)}
↪ feature vectors")

    # Grouping data by class
    class_data = {}
    for i, label in enumerate(label_list):
        if label not in class_data:
            class_data[label] = []
        class_data[label].append(features_list[i])

    # Print class distribution
    print("\nClass distribution:")
    for class_label in sorted(class_data.keys()):
        print(f"Class {class_label}: {len(class_data[
↪ class_label])} samples")

    # Create sequences within each class separately
    all_sequences = []
    all_labels = []

    print(f"\nCreating sequences with length={
↪ sequence_length}, step={step}")
    for class_label in sorted(class_data.keys()):
        class_features = class_data[class_label]
        class_sequences = []

        # Create sequences within this class only
        for start_idx in range(0, len(class_features) -
↪ sequence_length + 1, step):
            sequence = class_features[start_idx:start_idx +
↪ sequence_length]
            class_sequences.append(sequence)

        # Add to main lists
        all_sequences.extend(class_sequences)
        all_labels.extend([class_label] * len(
↪ class_sequences))

        print("Class {class_label}: {len(class_sequences)}
↪ sequences created")

    return np.array(all_sequences), np.array(all_labels)


def load_raw_features(row):
    """Extracts features from a single row."""
    try:
        master_rssi = float(row["master_rssi"])
        worker_rssi = float(row["worker_rssi"])
        aoa = float(row["aoa"])

        IQ_master = np.array([int(x) for x in row["IQ_master
↪ "].split()])
        IQ_worker = np.array([int(x) for x in row["IQ_worker
↪ "].split()])
        # Concatenate scalar and IQ features into one vector
        features = np.concatenate([[master_rssi, worker_rssi
↪ , aoa], IQ_master, IQ_worker])
        return features
    except Exception as e:
        print(f"Error processing row: {e}")
        return None
```

Listing 3: Raw CSI and Features

```
import numpy as np
from scipy.stats import skew, kurtosis
from scipy.signal import welch

#PARSING & FEATURE EXTRACTION
def parse_iq(iq_string):
    try:
        values = list(map(int, iq_string.strip().split()))
        iq_array = np.array(values).reshape(-1, 2)
        return iq_array[:, 0] + 1j * iq_array[:, 1]
    except Exception:
        return None

def extract_features_from_row(row):
    try:
        iq_master = parse_iq(row["IQ_master"])
        iq_worker = parse_iq(row["IQ_worker"])
```

```
        if iq_master is None or iq_worker is None:
            return None

        # Time-domain features
        amp_master = np.abs(iq_master)
        phase_master = np.angle(iq_master)
        amp_worker = np.abs(iq_worker)
        phase_worker = np.angle(iq_worker)
        phase_diff = np.unwrap(phase_master - phase_worker)
        amp_diff = amp_master - amp_worker

        features = []
        # Statistical features (4x11 = 44)
        for signal in [amp_master, amp_worker, phase_diff,
↪ amp_diff]:
            features.extend([
                np.mean(signal), np.std(signal), np.var(
↪ signal), np.min(signal),
                np.max(signal), np.ptp(signal), np.median(
↪ signal),
                np.percentile(signal, 25), np.percentile(
↪ signal, 75),
                skew(signal), kurtosis(signal)
            ])
        # Correlation features (2)
        corr_amp = np.corrcoef(amp_master, amp_worker)[0, 1]
        corr_phase = np.corrcoef(phase_master, phase_worker)
↪ [0, 1]
        features.append(corr_amp if not np.isnan(corr_amp)
↪ else 0)
        features.append(corr_phase if not np.isnan(
↪ corr_phase) else 0)

        # Frequency-domain features (2x7 = 14)
        for signal in [amp_master, amp_worker]:
            try:
                freqs, psd = welch(signal, nperseg=min(len(
↪ signal), 64))
                features.extend([
                    np.mean(psd), np.std(psd), np.max(psd),
↪ np.argmax(psd),
                    np.sum(psd[:len(psd)//4]) / np.sum(psd),
                    np.sum(psd[len(psd)//2:]) / np.sum(psd),
                    np.sum(psd[:len(psd)//10]) / np.sum(psd)
                ])
            except Exception:
                features.extend([0]*7)
        # Phase relationships(6)
        features.extend([
            np.mean(np.abs(phase_diff)), np.std(np.abs(
↪ phase_diff)),
            np.mean(np.cos(phase_diff)), np.std(np.cos(
↪ phase_diff)),
            np.mean(np.sin(phase_diff)), np.std(np.sin(
↪ phase_diff))
        ])
        # RSSI and other metadata(5)
        master_rssi = float(row.get("master_rssi", -100))
        worker_rssi = float(row.get("worker_rssi", -100))
        features.extend([
            master_rssi, worker_rssi,
            master_rssi - worker_rssi,
            float(row.get("seq_ctrl", 0)),
            float(row.get("aoa", 0))
        ])
        return features
    except Exception as e:
        print(f"Feature extraction error: {e}")
        return None
```

Listing 4: Extracted CSI Features

```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from sklearn.preprocessing import RobustScaler, LabelEncoder
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix,
↪ ConfusionMatrixDisplay



#CONFIGURATION
SEQUENCE_LENGTH = 40
OVERLAP = 20
BATCH_SIZE = 64
EPOCHS = 30
HIDDEN_SIZE = 128
NUM_LAYERS = 3
LEARNING_RATE = 0.0005
DROPOUT = 0.5
WEIGHT_DECAY = 1e-5
DEVICE = torch.device("cuda" if torch.cuda.is_available()
↪ else "cpu")
print(f"Using device: {DEVICE}")

#Train/Val/Test Split

def create_stratified_split(X, y, test_size=0.2, val_size
↪ =0.1): #for 70/10/20 train/val/test
    """Create stratified train/val/test split"""
    X_temp, X_test, y_temp, y_test = train_test_split(
        X, y, test_size=test_size, stratify=y, random_state
↪ =42
    )

    val_size_adjusted = val_size / (1 - test_size)
    X_train, X_val, y_train, y_val = train_test_split(
        X_temp, y_temp, test_size=val_size_adjusted,
↪ stratify=y_temp, random_state=42
    )
    return X_train, X_val, X_test, y_train, y_val, y_test
#LSTM MODEL


class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers,
↪ num_classes, dropout):
        super().__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
```

```
        self.lstm = nn.LSTM(input_size, hidden_size,
↪ num_layers,
                            batch_first=True, bidirectional=True
↪ , dropout=dropout)
        self.batch_norm = nn.BatchNorm1d(hidden_size * 2)
        self.attention = nn.MultiheadAttention(hidden_size *
↪  2, num_heads=8, dropout=dropout)
        self.classifier = nn.Sequential(
            nn.Linear(hidden_size * 2, hidden_size),
            nn.BatchNorm1d(hidden_size),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_size, hidden_size // 2),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_size // 2, num_classes)
        )

    def forward(self, x):
        batch_size = x.size(0)

        # Initialize hidden state
        h0 = torch.zeros(self.num_layers * 2, batch_size,
↪ self.hidden_size).to(x.device)
        c0 = torch.zeros(self.num_layers * 2, batch_size,
↪ self.hidden_size).to(x.device)

        # LSTM layers
        lstm_out, _ = self.lstm(x, (h0, c0))

        # Apply batch normalization (reshape for batch norm)
        lstm_out_reshaped = lstm_out.permute(0, 2, 1)  # (
↪ batch, features, seq_len)
        lstm_out_bn = self.batch_norm(lstm_out_reshaped)
        lstm_out_bn = lstm_out_bn.permute(0, 2, 1)

        # MultiheadAttention (seq_len, batch, features)
        lstm_out_transposed = lstm_out_bn.permute(1, 0, 2)
        attn_out, _ = self.attention(lstm_out_transposed,
↪ lstm_out_transposed, lstm_out_transposed)

        # Take the mean across sequence dimension for
↪ classification
        attn_out = attn_out.permute(1, 0, 2)
        pooled_out = torch.mean(attn_out, dim=1)  # (batch,
↪ features)

        # Classification
        return self.classifier(pooled_out)

#MODEL TRAINING
def train_lstm(X, y):
    print("\n" + "="*50)
    print("Starting␣LSTM␣Training␣with␣Validation")
    print(f"Input␣shape:␣{X.shape}")
    print(f"Sequence␣length:␣{SEQUENCE_LENGTH},␣Features:␣{X
↪ .shape[2]}")
    print(f"Classes:␣{len(np.unique(y))}")

    # Check class distribution
    unique, counts = np.unique(y, return_counts=True)
    print("\nSequence␣distribution␣per␣class:")
    for class_label, count in zip(unique, counts):
        print(f"Class␣{class_label}:␣{count}␣sequences")
    print("="*50)

    # Encode labels
    le = LabelEncoder()
    y_encoded = le.fit_transform(y)
    num_classes = len(le.classes_)

    # Stratified split
    X_train, X_val, X_test, y_train, y_val, y_test =
↪ create_stratified_split(
        X, y_encoded, test_size=0.2, val_size=0.1
    )

    print(f"Train:␣{X_train.shape[0]},␣Val:␣{X_val.shape
↪ [0]},␣Test:␣{X_test.shape[0]}")

    # Scale features
    n_features = X.shape[2]
    scaler = RobustScaler()

    # Fit scaler only on training data
    X_train_flat = X_train.reshape(-1, n_features)
    scaler.fit(X_train_flat)

    # Transform all sets
    X_train_scaled = scaler.transform(X_train.reshape(-1,
↪ n_features)).reshape(X_train.shape)
    X_val_scaled = scaler.transform(X_val.reshape(-1,
↪ n_features)).reshape(X_val.shape)
    X_test_scaled = scaler.transform(X_test.reshape(-1,
↪ n_features)).reshape(X_test.shape)

    # Create datasets
    train_dataset = TensorDataset(
        torch.tensor(X_train_scaled, dtype=torch.float32),
        torch.tensor(y_train, dtype=torch.long)
    )
    val_dataset = TensorDataset(
        torch.tensor(X_val_scaled, dtype=torch.float32),
        torch.tensor(y_val, dtype=torch.long)
    )
    test_dataset = TensorDataset(
        torch.tensor(X_test_scaled, dtype=torch.float32),
        torch.tensor(y_test, dtype=torch.long)
    )

    # Create data loaders
    train_loader = DataLoader(train_dataset, batch_size=
↪ BATCH_SIZE, shuffle=True)
    val_loader = DataLoader(val_dataset, batch_size=
↪ BATCH_SIZE, shuffle=False)
    test_loader = DataLoader(test_dataset, batch_size=
↪ BATCH_SIZE, shuffle=False)

    # Initialize model
    model = LSTM(
        input_size=n_features,
        hidden_size=HIDDEN_SIZE,
        num_layers=NUM_LAYERS,
        num_classes=num_classes,
```

```
        dropout=DROPOUT
    ).to(DEVICE)

    # Loss and optimizer
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=
↪ LEARNING_RATE, weight_decay=WEIGHT_DECAY)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(
↪ optimizer, mode='min', patience=5, factor=0.5)

    # Training variables
    best_val_acc = 0.0
    train_losses = []
    val_losses = []
    val_accuracies = []
    patience_counter = 0
    patience = 10

    # Training loop
    for epoch in range(EPOCHS):
        # Training phase
        model.train()
        running_loss = 0.0
        for inputs, labels in train_loader:
            inputs, labels = inputs.to(DEVICE), labels.to(
↪ DEVICE)

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters
↪ (), max_norm=1.0)
            optimizer.step()
            running_loss += loss.item()

        epoch_train_loss = running_loss / len(train_loader)
        train_losses.append(epoch_train_loss)

        # Validation phase
        model.eval()
        val_loss = 0.0
        correct = 0
        total = 0

        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs, labels = inputs.to(DEVICE), labels.
↪ to(DEVICE)
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                val_loss += loss.item()

                _, predicted = torch.max(outputs, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item
↪ ()

        epoch_val_loss = val_loss / len(val_loader)
        val_acc = correct / total
        val_losses.append(epoch_val_loss)
        val_accuracies.append(val_acc)

        # Learning rate scheduling
        scheduler.step(epoch_val_loss)

        # Early stopping and model saving
        if val_acc > best_val_acc:
            best_val_acc = val_acc
            torch.save(model.state_dict(), "
↪ Best_lstm_model_RawCSI.pt")
            patience_counter = 0
            print(f"Epoch␣{epoch+1}:␣New␣best␣validation␣
↪ accuracy:␣{val_acc:.4f}")
        else:
            patience_counter += 1

        print(f"Epoch␣{epoch+1}/{EPOCHS}␣|␣"
            f"Train␣Loss:␣{epoch_train_loss:.4f}␣|␣"
            f"Val␣Loss:␣{epoch_val_loss:.4f}␣|␣"
            f"Val␣Acc:␣{val_acc:.4f}␣|␣"
            f"LR:␣{optimizer.param_groups[0]['lr']:.6f}")

        # Early stopping
        if patience_counter >= patience:
            print(f"Early␣stopping␣triggered␣after␣{epoch+1}
↪ ␣epochs")
            break

    # Load best model for final evaluation
    model.load_state_dict(torch.load("best_lstm_model.pt"))

    # Final test evaluation
    model.eval()
    test_correct = 0
    test_total = 0
    y_true, y_pred = [], []

    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(DEVICE), labels.to(
↪ DEVICE)
            outputs = model(inputs)
            _, predicted = torch.max(outputs, 1)
            test_total += labels.size(0)
            test_correct += (predicted == labels).sum().item
↪ ()
            y_true.extend(labels.cpu().numpy())
            y_pred.extend(predicted.cpu().numpy())

    final_test_acc = test_correct / test_total
    print(f"\nFinal␣Test␣Accuracy:␣{final_test_acc:.4f}")

    # Confusion Matrix
    cm = confusion_matrix(y_true, y_pred)
    cm_normalized = cm.astype('float') / cm.sum(axis=1)[:,
↪ np.newaxis]
    cm_normalized = np.round(cm_normalized, 2)
    # Plot normalized confusion matrix
    disp_norm = ConfusionMatrixDisplay(confusion_matrix=
↪ cm_normalized, display_labels=le.classes_)
    disp_norm.plot(cmap=plt.cm.Blues)
    #plt.title("Confusion Matrix - Normalized")
    plt.savefig('LSTM_Raw.png', dpi=300)
```

```
    plt.show()

    return model
```

Listing 5: LSTM Structure

```
# Installing all dependencies
import numpy as np
import pandas as pd
import joblib
from sklearn.model_selection import (train_test_split,
                                     StratifiedKFold,
↪ GridSearchCV)
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score,
↪ classification_report
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
import warnings
from sklearn.metrics import (
    accuracy_score,
    classification_report
)
warnings.filterwarnings('ignore')

def load_data_from_csv(file_path):
    df = pd.read_csv(file_path)
    X = []
    y = []
    for _, row in df.iterrows():
        features = extract_features_from_row(row)
        if features is not None:
            X.append(features[:-1])
            y.append(int(row["label"]))
    return np.array(X), np.array(y)

# Model Training
def train_model(X_train, y_train, X_test, y_test, model,
↪ model_name, param_grid=None):
    print(f"\nTraining␣{model_name}...")

    if param_grid:
        cv = StratifiedKFold(n_splits=3, shuffle=True,
↪ random_state=42)
        grid = GridSearchCV(model, param_grid, cv=cv, n_jobs
↪ =-1, verbose=1)
        grid.fit(X_train, y_train)
        model = grid.best_estimator_
        print(f"Best␣params:␣{grid.best_params_}")
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    print(f"{model_name}␣Accuracy:␣{acc:.4f}")
    print(classification_report(y_test, y_pred))
    joblib.dump(model, f"{model_name.lower().replace('␣','␣'_
↪ ')}_model.pkl")
    return model, acc


# Main block
if __name__ == "__main__":
    # Load data
    file_path = "C:\\Users\\memoo\\esp\\Project\\Training.
↪ csv"
    print("Loading␣data...")
    X, y = load_data_from_csv(file_path)

    print(f"Data␣loaded:␣{X.shape[0]}␣samples,␣{X.shape[1]}␣
↪ features")
    print(f"Class␣distribution:␣{np.bincount(y)}")

    # Split data (70% train, 10% validation, 20% test)
    X_train, X_temp, y_train, y_temp = train_test_split(
        X, y, test_size=0.3, random_state=42, stratify=y
    )
    X_val, X_test, y_val, y_test = train_test_split(
        X_temp, y_temp, test_size=0.667, random_state=42,
↪ stratify=y_temp
    )
    # Scale features
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_val_scaled = scaler.transform(X_val)
    X_test_scaled = scaler.transform(X_test)
    print(f"Train␣set:␣{X_train.shape[0]}␣samples")
    print(f"Validation␣set:␣{X_val.shape[0]}␣samples")
    print(f"Test␣set:␣{X_test.shape[0]}␣samples")

    models = {
        "Random␣Forest": RandomForestClassifier(
            n_estimators=300,
            random_state=42,
            max_depth=20,
            min_samples_leaf=10,
            max_features=0.2,
            n_jobs=-1,
        ),
        "KNN": KNeighborsClassifier(n_neighbors=3, weights="
↪ distance", p=2),
        "SVM": SVC(
            C=10.0,
            gamma="scale",
            kernel="rbf",
            class_weight="balanced",
            probability=True,
            random_state=42,
        ),
    }
    results = {}
    for name, model in models.items():
        print(f"\nTraining␣{name}...")

        # Train model
        model.fit(X_train_scaled, y_train)

        # Validate model
        y_val_pred = model.predict(X_val_scaled)
        val_accuracy = accuracy_score(y_val, y_val_pred)
        y_test_pred = model.predict(X_test_scaled)
        test_accuracy = accuracy_score(y_test, y_test_pred)

        results[name] = {
            "model": model,
```

```
            "val_accuracy": val_accuracy,
            "test_accuracy": test_accuracy,
            "test_predictions": y_test_pred,
        }
        print(f"{name} Validation Accuracy: {val_accuracy:.4
↪ f}")
        print(f"{name} Test Accuracy: {test_accuracy:.4f}")
```

Listing 6: Structure of KNN, SVM and RF

B Using the Dataset

The dataset used in this report is stored in the private repository:

`https://github.com/henryblu/ML-Powered-WiFi-Tracker`

The primary dataset file is located at:

`training_data/11-07-2025-training-data-set.csv`

Access to the repository can be provided upon request. Please contact the corresponding author at the email address provided in the report.

B.1 Dataset Format

Each row in the CSV file corresponds to a single CSI capture, with the following columns:

- `timestamp` (`uint64`): Unix timestamp of the packet.
- `mac` (`string`): Transmitter MAC address.
- `seq_ctrl` (`uint16`): Wi-Fi sequence control number.
- `aoa` (`float32`): Calculated Angle-of-arrival in degrees. Warning! This value is often wrong.
- `master_rssi`, `worker_rssi` (`int8`): RSSI values from the master and worker ESP32-S3 devices.
- `IQ_master`, `IQ_worker` (`string`): CSI samples from the master and worker devices as interleaved real/imaginary pairs.
- `distance_m` (`float32`): Distance of the UE from the ESP array in meters.
- `angle_deg` (`float32`): Angle of the UE relative to the ESP array in degrees.

B.2 Loading the Dataset

The following Python snippet shows how to load and parse the dataset, including splitting the CSI IQ samples into flattened integer arrays:

```
import pandas as pd
import numpy as np

# Load dataset
df = pd.read_csv(
    'training_data/11-07-2025-training-data-set.csv'
)

# Parse IQ samples from master and worker into flattened
↪ integer arrays
def parse_iq(iq_str):
    return [int(x) for x in iq_str.strip().split()]

df['IQ_master_array'] = df['IQ_master'].apply(parse_iq)
df['IQ_worker_array'] = df['IQ_worker'].apply(parse_iq)

# Example: print AoA and RSSI
print(df[['aoa', 'master_rssi', 'worker_rssi']])
```

Listing 7: Loading and parsing the dataset CSV.

This structure allows users to extract features, preprocess CSI data, and train machine learning models as described in Sections 4.2 and 5.