CSI Sense Zero

Realtime Wi-Fi CSI based HAR on a RPi Zero + ESP32

Term Project

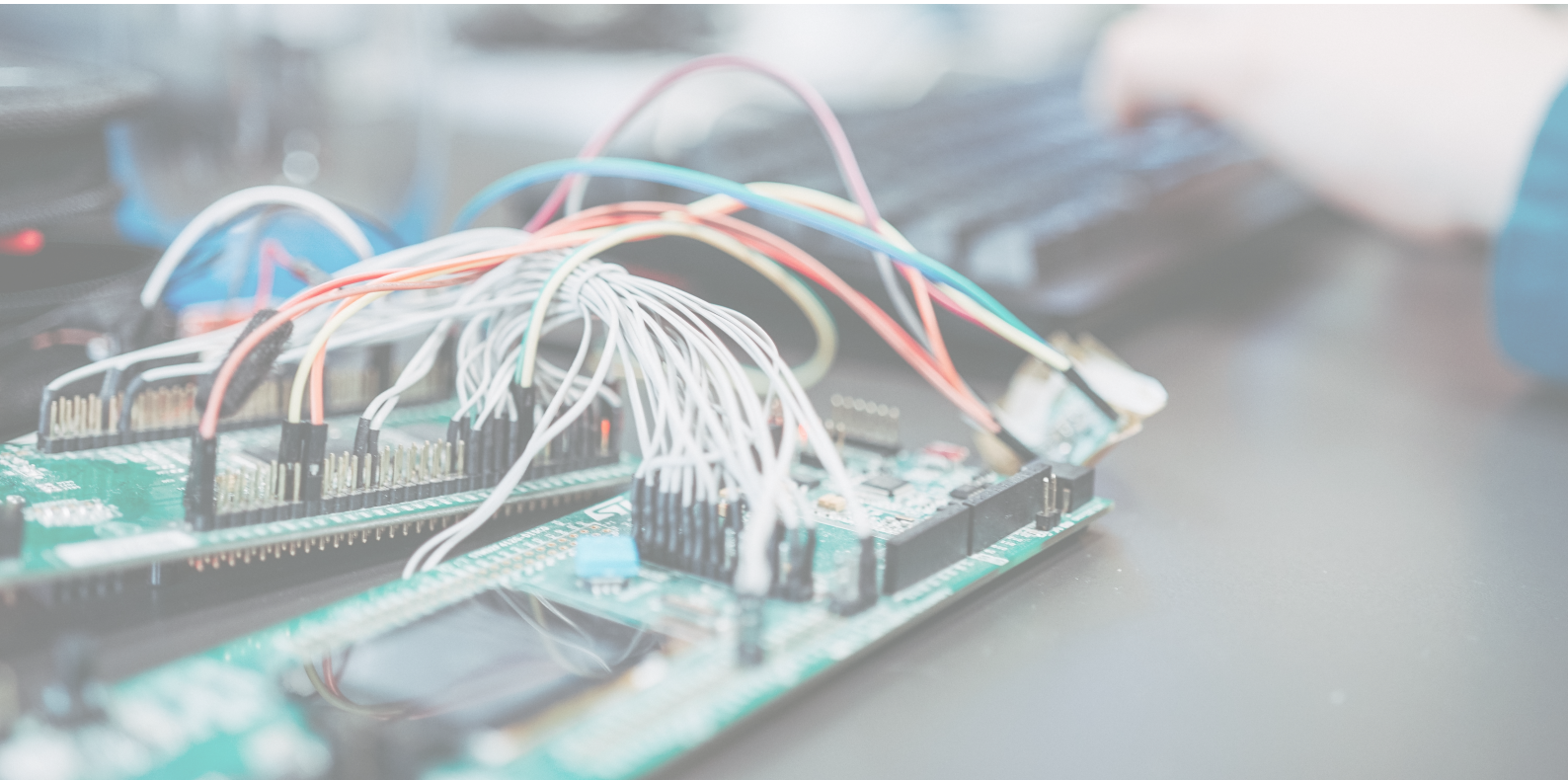
CS6650: Smart Sensing for Internet of Things

Group

Ashwin A Nayar	**Manu K Paul**	**Ankit Sanghvi**	**Sreeraj R**
NA19B001	NA19B045	CE19B028	NA20B066

Instructor: Prof. Ayon Chakraborty

May 8, 2023



Contents

Abstract

Channel State Information (CSI), paramount to modern Wi-Fi communication systems, can accurately mirror physical changes in the wireless channel. In this project, we explore the properties of CSI of Wi-Fi signals and present a cost effective and non-invasive activity recognition solution capable of running in embedded systems. The prototype utilizes nascent algorithms in the domain of time-series classification and reliably detects and identifies activity classes - *Idle*, *Walk* and *Jump*, in realtime, with over 95% accuracy. Recon performed on in-house captured CSI activity data along with experiments that back major design decisions such as choice of algorithms and system architecture are also delineated with emphasis on the insights and conclusions.

1 | Introduction

Human Activity Recognition (HAR) is a core technology that enables a wide variety of real-world applications such as health care, smart homes, fitness tracking, building surveillance etc. Channel State Information (CSI) of Wi-Fi signals can accurately mirror physical changes in the wireless channel. CSI based HAR solutions are becoming more and practical alternatives to costly vision-based approaches, known also to be privacy-intrusive.

Existing models for HAR generally have a high computational complexity, contain very large number of trainable parameters, and require extensive computational resources. These are often impractical to deploy on low-cost embedded systems which lack sufficient compute power to host such solutions. Wang et al. [1] proposes CARM, a model that can quantitatively correlate CSI dynamics and human activities. The approach involves a PCA based CSI denoising scheme followed by Discrete Wavelet Transform based feature extraction, statistical activity detection and activity prediction using Hidden Markov Models (HMM). Li et al. [2] employs a Sparse Representation Classifier (SRC) using manually chosen time domain and frequency domain features from the CSI data. Dayal et al. [3] achieved formidable sound spectrogram classification accuracy using a lightweight CNN capable of running on embedded platforms. This approach indeed looks promising to adapt towards CSI spectrogram inputs.

For realtime activity recognition coupled with end-to-end embedded hardware deployment, a lightweight approach in terms of processing, feature extraction and classification is required. Current solutions to CSI based HAR are either too resource hungry or give poor results on CSI data sampled from low-end devices such as an ESP32.

We aim to develop an end-to-end CSI based HAR solution capable of running real-time on a cost-effective and resource-limited embedded environment.

2 | Hardware

Our hardware configuration involves two ESP32 modules (configured as a transmitter and receiver) and an RPi Zero. ESP32 is highly integrated, with a dual-core 32-bit processor, and bundled with all necessary peripherals such as in-built antenna switches, RF balun, power amplifier, low-noise receiving amplifier, filters, sensors, and power management modules etc. We opted for ESP32(s) due to their ease of data acquisition through the official ESP-CSI Toolkit [4], which provides direct access to CSI data. Additionally, the low cost and low power requirements of ESP32 were also considered.

The most affordable and computationally sufficient hardware with the necessary communication interfaces was the RPi Zero. The data acquired from ESP32 CSI receiver module could be transferred to RPi via either USB serial or UART. To make sure that data transfer rate is consistent with the rate at which CSI data is being generated and to avoid any bottlenecks in the data ingestion pipeline, we carried out an analysis of the data transfer rate in both these interfaces.

We used the terminal based `pv` tool to monitor the data throughput through a unix pipe. This was repeated for UART (RPi ↔ ESP32), USB Serial (RPi ↔ ESP32) and USB Serial (Laptop ↔ ESP32).

```
# 921600 baud rate , 8 data bits , 1 stop bit , no parity
stty -F /dev/esp32 921600 cs8 -cstopb -parenb
cat < /dev/esp32 | pv > /dev/null
```

The results from all cases were fairly similar and stable (approx. 85 kbits/s), suggesting that either one would suffice for our purpose. Figure 2.1 illustrates the final RPi ↔ ESP32 connections.

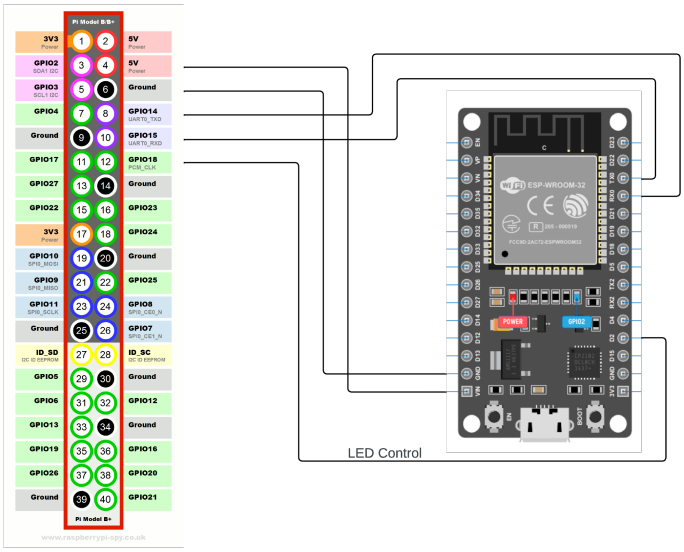


Figure 2.1: ESP32 (csi_recv) - RPi wiring

3 | Data Acquisition (DAQ)

3.1 | Setup

ESP32 is flashed with the official CSI toolkit [4] which exposes CSI data at 100 Hz. We place our ESP32(s) approximately 3m apart and perform our activities (Idle, Walk, Jump) in Line-Of-Sight. We recorded 3 hours worth of CSI data (combined across activity classes) for all the 3 activities on different days and different volunteers to introduce variation within an activity class with is necessary for learning generalization.

> *Henceforth, the ESP32 module flashed with CSI sender will be referred to as* **`csi_send`** *and the one flashed with CSI receiver will be referred to as* **`csi_recv`**.

Both `csi_send` and `csi_recv` are powered by standard power banks, the latter indirectly gets power from the RPi as illustrated in Figure 2.1. Activity is performed in LOS and the raw CSI data from the ESP is logged to disk at RPi end.

3.2 | Addressing CPU Demands of Serial Read

During initial stages of the project, we carried out raw CSI data logging using tools like picocom to configure and read serial ports.

```
picocom --baud 921600 --logfile data.csi /dev/esp32
```

However this setup caused unusual spike in CPU usage (in RPi) and we were able to boil down the cause to kernel threads under the `picocom` process. Although this didn't pose much issue during DAQ, the high CPU usage meant such tools couldn't be used for final deployment and inference.

The solution was to program the serial port manually and use GNU coreutils tools like `cat` to read the serial port as if it were a file.

```
stty -F /dev/esp32 921600 cs8 -cstopb -parenb
cat < /dev/esp32 | awk NF > data.csi
```

3.3 | Data Preprocessing

CSI in ESP32 contains the Channel Frequency Responses (CFR) of subcarriers and is calculated when packets travel from TX to RX. Each CFR of a subcarrier registers as two bytes of signed characters, the first part being the imaginary, and the second part is the real value, and we receive three CFR types from the CSI data. The in-built specification in ESP32 is for three CFR types to be received from the CSI data. These are Legacy Long Training Field (LLTF), High-Throughput LTF (HT-LTF), and Spacetime Block Code HT-LTF (STBC-HT-LTF).

For our project, we only focus on LLTF data. By analysing the packet meta data and with the official reference [5] we are able to identify the indices of null, pilot and data subcarriers. For training and any subsequent offline analysis on the captured data, the data is preprocessed in stages,

- Filtering out bad lines - incomplete data from serial port, usually in the beginning and end of CSI log file
- Using the first 128 bytes (LLTF) data (real + imaginary parts) to compute CSI amplitude of all 64 LLTF sub-carriers
- Filtering out invalid sub-carrier data (null + pilot carriers)

The timeseries data of a particular recording session is split into chunks (with singular stride) of size 256 (at 100Hz default sampling rate this corresponds to a duration of 2.5s). The final matrix representation is of dimension `n_samples x n_subcarriers x 256`. Our aim is to classify a given sample of size `n_subcarriers x 256` to the three activity classes - *Idle*, *Walk* and *Jump*. During liverun we implement this chunking realtime using a ring buffer.

All these steps are to be repeated each time before analysing a raw CSI logfile. This can get quite repetitive and mundane. This calls for a clean, concise and ready-to-use intermediate dataset representation. The processed data is thus exported as a MATLAB-style `.mat` file with relevant metadata using Python's `scipy` library - `scipy.io.savemat`.

Unlike most activity recognition approaches, we do not perform any major preprocessing on individual CSI subcarrier data such as filtering and smoothing.

4 | Feature Engineering and Classification Attempts

Before proceeding with any classification attempts, it is important to analyse the current data representation and whether or not it admits clustering tendencies. Such analysis will help us make informed choices at algorithms and narrow down the search for a learning/classification algorithm.

We chose to use t-distributed Stochastic Neighbour Embedding (t-SNE) over the popular PCA based approach for visualising our high-dimensional CSI data for its ability to handle outliers and the former is specifically designed to visualise high-dimensional data and is suitable for non-linearly separable datasets. Although the math behind t-SNE is quite complex [6], there are reliable interfaces in Python's `scikit-learn` ML library that could be used to run t-SNE on standard data representations.

4.1 | Visualizing the High-Dimensional CSI data

The procedure is quite simple - For each data sample (`n_subcarriers x 256`) t-SNE will associate a 2D cartesian cordinate (X_{emb}, Y_{emb}) which could be used for visualisation of the dataset as a scatter plot.

Case 1: Analysing clusterability on a day by day basis

We are looking at data collected in the same environment but on two different days. Minute changes in the environment will lead to cascading multi-path effects which will affect CSI data.

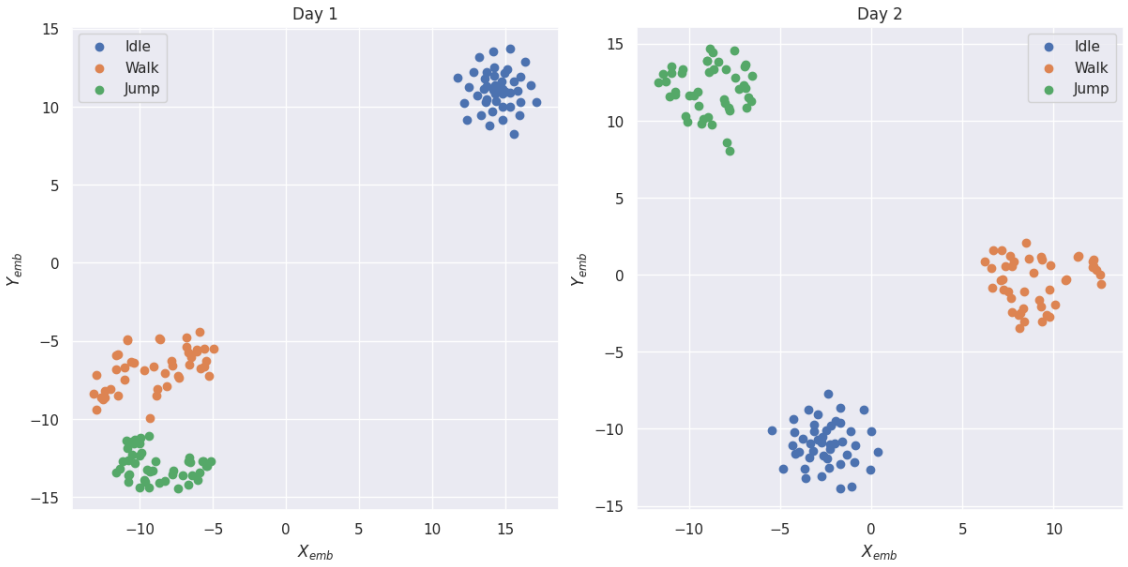


Figure 4.1: t-SNE on data, day-by-day basis

(**Observation**) We see that the activities are clustering out really well each day. But what if we perform t-SNE on the entire data? Will we see a similar clustering tendency?

Case 2: Analysing CSI data collectively

We now run t-SNE on a combined dataset, which has samples (across all 3 activity classes) collected on different days but in the same environment.

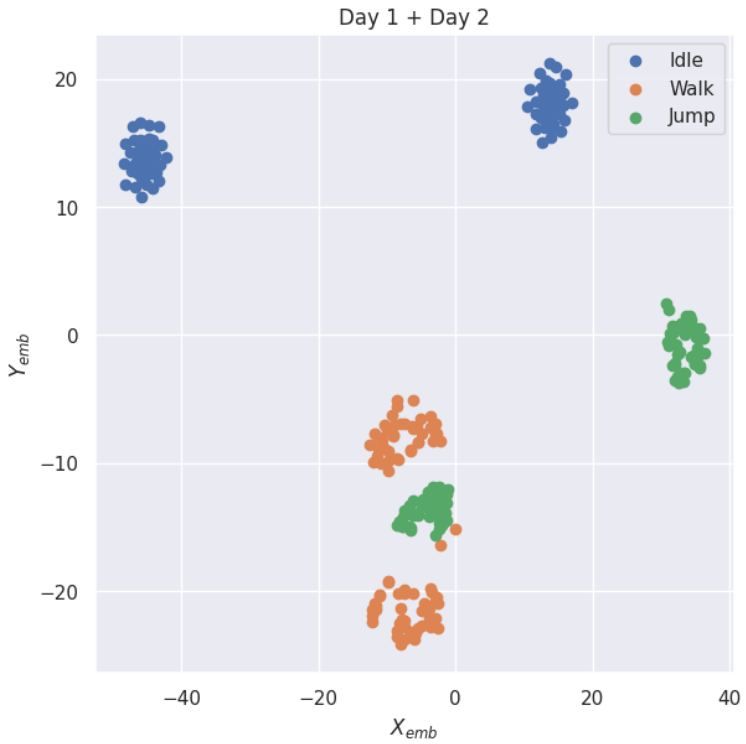


Figure 4.2: t-SNE on combined data from different days

(**Observation**) Activities from each day are clustering independently. We hypothesise this is due to a combined effect of minute changes in environment leading to cascading multi-path effects and human

error. Any learning algorithm trained on the data in the current representation will most likely perform poorly on unseen data captured on a different day.

4.2 | Validating Hypothesis - Performance of Some Classifiers

Procedure

Two datasets are prepared with uniform number of samples across each activity class.

- **D1** - Data collected on Day 1, Day 2 and Day 3
- **D2** - Data collected on Day 4

Data is collected in the exact same environment with minimal to no configuration changes. Training and evaluation of a classifier is done as follows;

- An 80-20 stratified split is done on **D1** - **D1.1** and **D1.2** respectively.
- Classifier is trained on **D1.1**, tested on **D1.2** followed by a separate test on **D2**.

The hypothesis at the end of t-SNE analysis can now be rephrased as follows - Post training, we expect satisfactory classification accuracy on **D1.2** but poor results on **D2** since the model had no access to any sample from Day 4.

The performance analysis is carried out for 3 different classifers commonly used in HAR studies - Support Vector Machines (SVM), Convolutional Neural Networks (CNN) and Gaussian Mixture Models (GMM). Model agnostic input data formats and their performances are summarised below,

> **NOTE**: Each subcarrier data is scaled to lie in [0, 1] across entire dataset. A custom `CSIMinMaxScaler` has been implemented for this purpose.

- **Support Vector Machine (SVM)**
  - SVM is initialized with a RBF kernel and One-versus-One (ovo) decision function.
  - Dataset D1.1 is fitted onto the model followed by testing on D1.2 and D2.
- **Gaussian Mixture Model (GMM)**
  - For each sample, the `n_subcarriers x 256` matrix is flattened to form a feature vector.
  - The new data matrix of shape `n_samples x len(feature_vector)` is projected onto its principal components so as to capture 95% of the total variance.
  - The new feature representation is passed onto a Gaussian Mixture Model for unsupervised classification.
- **Convolutional Neural Network (CNN)**
  - Each sample is treated as a CSI spectrogram image of size `n_subcarriers x 256`.
  - The CNN architecture is inspired from Dayal et al. - lightweight CNN for spectrogram classification [3]. The model architecture used is summarised in Table 4.1.
  - All the Conv. layers are followed by ReLU activation function and all the weights are initialized using the HeNormal initializer.

Results

Classifier	% Accuracy on D1.2	% Accuracy on D2
SVM	96.66	33.33
CNN	93.96	33.33
GMM	Mispredicted labels (AUC < 0.5)	-

Conclusions

From the experiments carried out so far, we conclude that the CSI data in its current representation is not suitable for simple classification algorithms. We require some sort of feature transform prior to training and inference.

Table 4.1: CNN - Model summary

Layer (type)	Output Shape	Param #
Input	(None, 54, 256, 1)	0
Conv1 (dilation rate = 1)	(None, 52, 253, 8)	104
Max pool	(None, 26, 126, 8)	0
BatchNorm	(None, 26, 126, 8)	32
Conv2 (dilation rate = 1)	(None, 24, 123, 16)	1552
Max pool	(None, 12, 61, 16)	0
BatchNorm	(None, 12, 61, 16)	64
Conv3 (dilation rate = 2)	(None, 4, 51, 32)	15392
Max pool	(None, 2, 25, 32)	0
BatchNorm	(None, 2, 25, 32)	128
Flatten	(None, 1600)	0
Dense	(None, 3)	4803

5 | Rocket + Ridge Majority Voting

A recent advancement in the domain of timeseries classification is ROCKET - **R**and**O**m **C**onvolutional **KE**rnel **T**ransform [7]. The idea behind rocket to classify a timeseries can be summarised as follows,

- Initialize a large pool of 1D convolution kernels with varying parameters (length, weights, bias, dilation, padding, stride).
- Convolution outputs are then represented by two values per kernel, the proportion of positive values (ppv) and the maximum value (max).
- An advantage of this feature representation approach is mapping a variable-length time series to a fixed-length feature vector, which eliminates the padding of different-length signals.
- The transformed features are then used to train a linear classifier. The most commonly used one is Ridge Regression classifier for its simplicity and significantly fast classification.

In our project, we employ rocket as follows,

- Randomly initialize a large pool of kernels (say 10,000) with the following properties,
  - Length: Randomly selected from {7, 9, 11} with uniform probability;
  - Weights: Randomly sampled from a normal distribution $\mathcal{N}(0, 1)$;
  - Bias: Randomly sampled from a uniform distribution $\mathcal{U}(-1, 1)$;
  - Dilation: Randomly sampled from an exponential scale $\lfloor 2^a \rfloor$, where $a \sim \mathcal{U}(0, log_2 \frac{l_{input}-1}{l_{kernel}-1})$, l_{kernel} is the kernel length and l_{input} is the length of the CSI subcarrier data;
  - Padding: Applied randomly with uniform probability;
  - Stride: Set to one for all kernels;
- Treat each subcarrier (say we have 54 subcarriers) as a timeseries of length 256 (in each sample).
- Convolve each subcarrier data with each of the generated kernel, per sample. This is the set of new transformed features
$$(n\_samples \times n\_subcarriers \times 256) \longrightarrow (n\_samples \times n\_subcarriers \times 2 * n\_kernels)$$
- Train a ridge classifier per subcarrier for activity recognition using the transformed kernel features.
- The final prediction is made as follows - Given a sample, use the generated kernels, transform each subcarrier time series, accumulate the activity predictions from each subcarrier agnostic ridge classifier and do a majority voting across individual predictions to get the final prediction.

Both the kernel generation and convolutions are embarrassingly parallel operations. The implementation is done using just-in-time compilation via `Numba` in Python, with the tasks distributed across all the available CPU cores.

5.1 | Training

- Kernels are initialized randomly as dicussed in Section 5.
- For a given dataset, post kernel transform, a ridge classifier is trained for each subcarrier to classify the respective subcarrier data into activity classes. This is a parallelizable operation and is thus implemented to be run in parallel.
- After training, the generated kernel parameters and the classifier objects are saved for loading later during inference.

For large datasets this training process requires memory and due to its aggressive parallel implementation faster and increased number of CPU cores also speeds up the training process. We carried out training on our datasets in a AWS EC2 (t2.2xlarge) instance. Training process completed in $\sim$ 15 minutes and the saved parameters were exported.

5.2 | Performance Analysis

5.2.1 | Procedure

- Initialize the convolution kernels;
- Transform dataset D1.1;
- Train a ridge classifier per subcarrier on the transformed D1.1 dataset;
- Test the classifiers for activity recognition by majority voting across subcarrier agnostic classifiers on each of D1.2 and D2;

5.2.2 | Results

The approach achieved **96.67%** accuracy on D1.2 and **91.11%** accuracy on D2, far surpassing all the previously discussed classification attempts. The confusion matrix is presented in Figure 5.1.

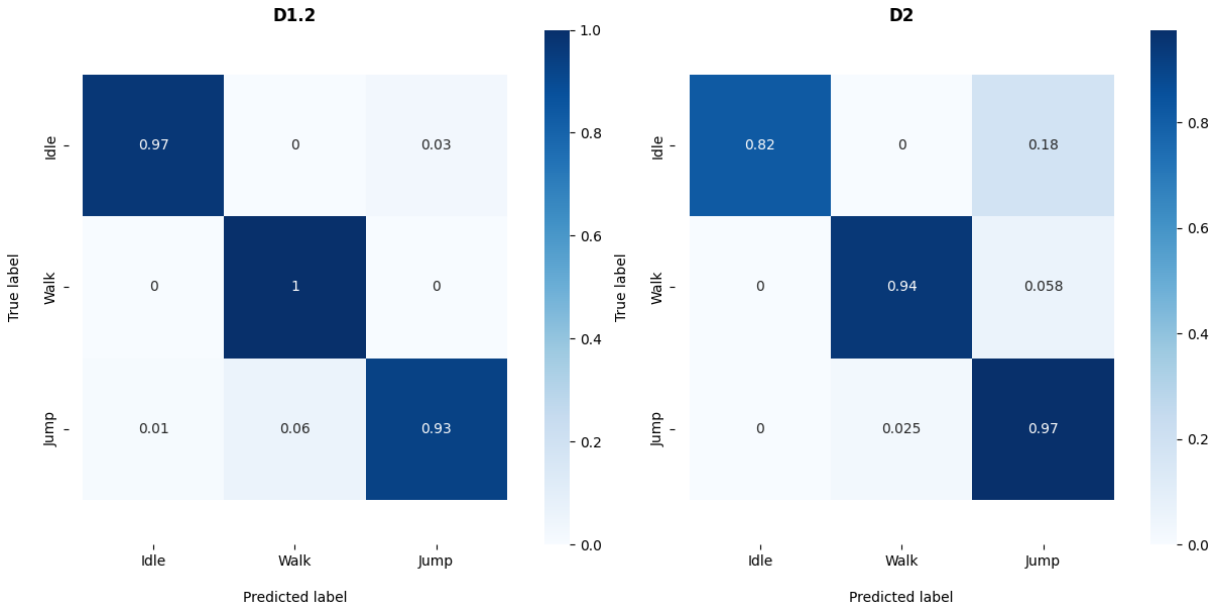


Figure 5.1: Confusion Matrix of Rocket + Ridge Voting approach on datasets D1.2 and D2

6 | Activity Indicator

Even though the proposed approach achieves satisfactory accuracy for deployment, it is not efficient to run the feature transform and ridge classification all the time. This is mainly because of the maximum use of compute available on the hardware by the algorithm (due to parallel implementation) and moreover ideally we only want to recognize an activity only when an activity/anomaly is detected in CSI data. This calls for a very simple approach to first detect an activity from the CSI data and only if an activity is detected should we be running the activity recognition pipeline.

6.1 | Correlation in CSI streams

We leverage the fact that changes introduced in all CSI streams during an activity are correlated. Given a sample of size `n_subcarriers x 256`, we run Singular Value Decomposition on this data matrix. This allows us to extract the eigenvectors in the spatial and temporal domain and optimally combine the CSI subcarriers to extract their principal orthogonal components.

We discard the first principal component $\mathbf{h}_1$ and focus on the second component $\mathbf{h}_2$ for formulating the activity indicator. This is because noise in CSI data caused by internal state changes are present in all CSI streams. Due to this high correlation, these noises are captured in $\mathbf{h}_1$ along with the activity signal. However an interesting result to note is that activity signal is captured in $\mathbf{h}_1$ as well as the other principal components because the phase of a subcarrier is a linear combination of two orthogonal components (a sine and cosine term) [1]. Since PCA components are uncorrelated, $\mathbf{h}_1$ captures only one of these orthogonal components and the other is retained in the rest of the components. Thus we can safely ignore $\mathbf{h}_1$ and focus on $\mathbf{h}_2$.

Figure 6.1 shows the first 5 principal components of data collected over a duration where a volunteer is walking perpendicular to LOS periodically back and forth. The shaded regions denote the LOS disturbance in CSI due to the activity.

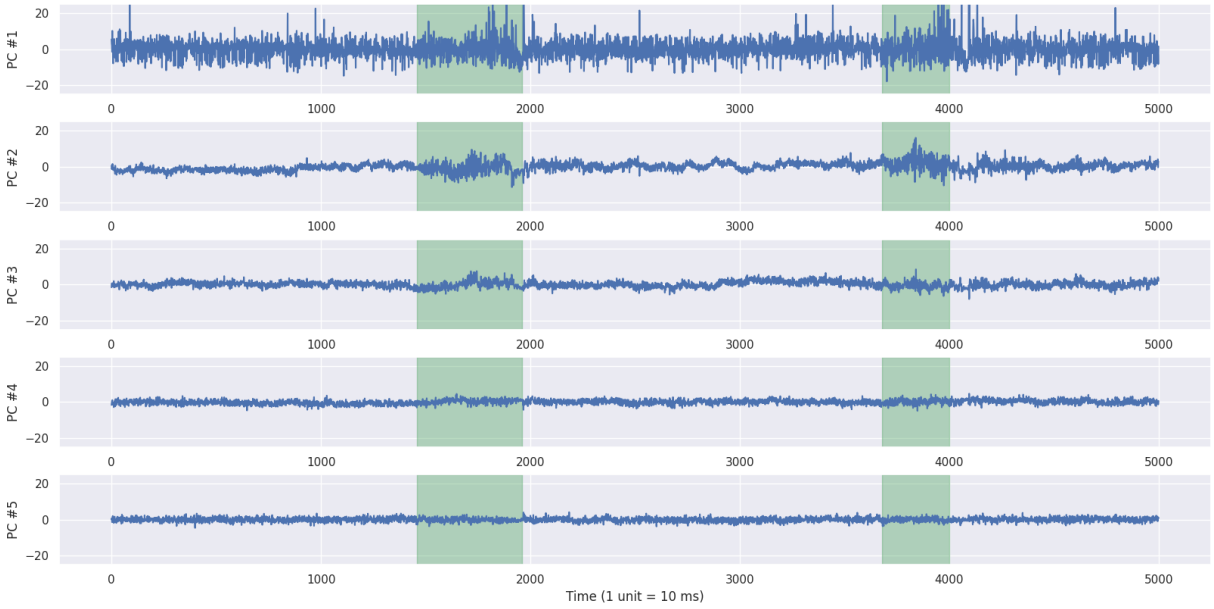


Figure 6.1: First 5 principal components of CSI data in which volunteer is periodically walking perpendicular to LOS. Shaded regions correspond to duration of disturbance in LOS.

6.2 | Experiment - Variance Thresholding

(**Motivation**) In the presence of an activity, $\mathbf{h}_2$ has a higher variance and in the absence of an activity, $\mathbf{h}_2$ has a lower variance. This is validated experimentally as illustrated in Figure 6.2 and Figure 6.3.

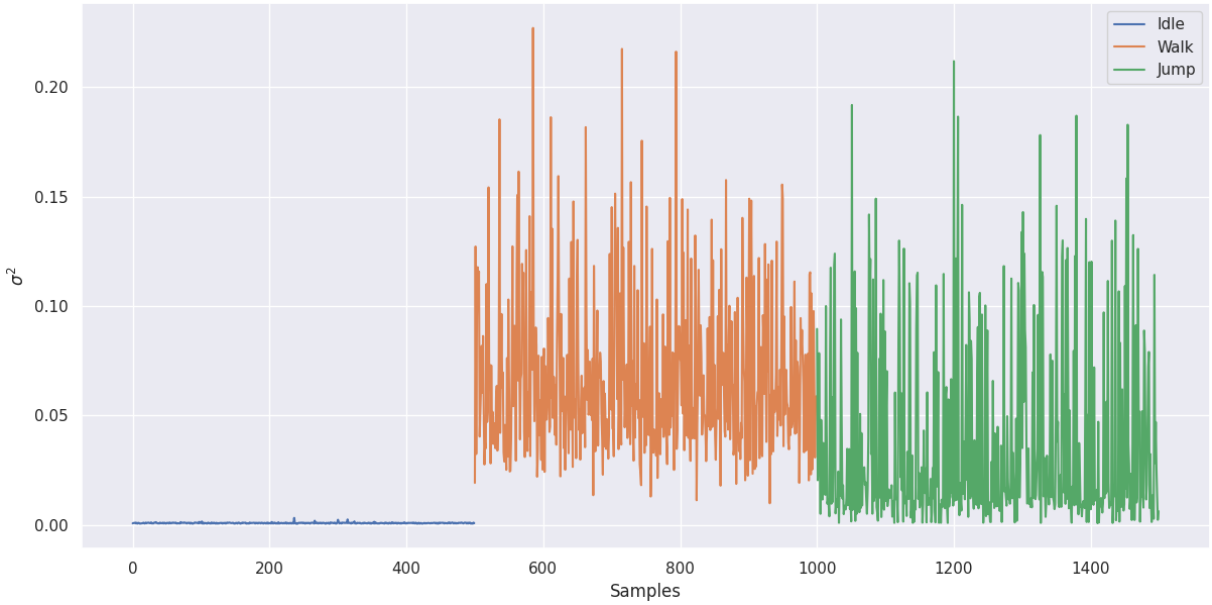


Figure 6.2: Variance of $\mathbf{h}_2$ across all samples

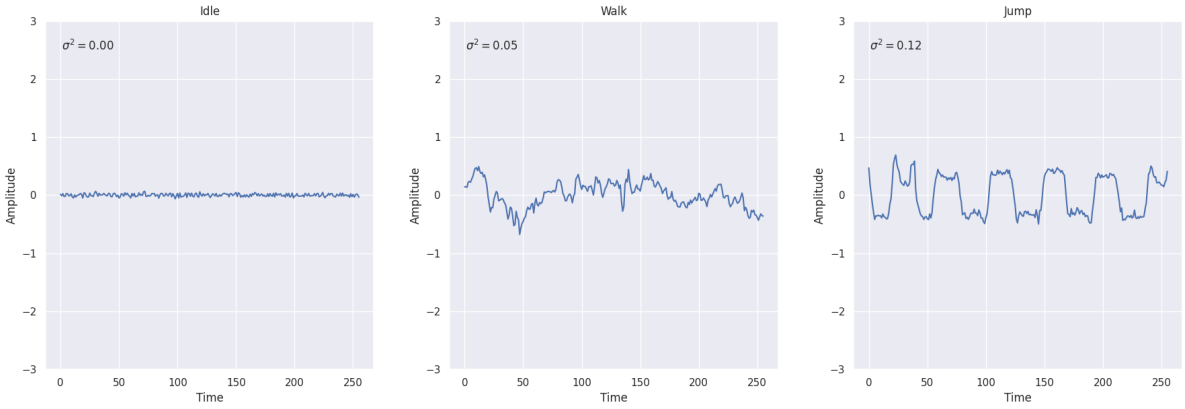


Figure 6.3: Second Principal Component ($\mathbf{h}_2$) from each activity class

(**Procedure**) The next step is to analyze the performance of a simple variance based thresholding classifier for activity detection. We define a loss function, that takes as input a threshold value and computes the "loss" defined as;

```
# true labels - 0 for no activity (Idle), 1 for activity (Walk + Jump)
y_actual = ...

# variance of all samples computed from SVD
variances = ...

def loss(th):
    y_pred = np.zeros_like(y_actual)
    y_pred[variances > th] = 1
    return 1 - accuracy_score(y_actual, y_pred)
```

This is clearly a convex function and goal is to compute the global minima. This is easily done using Python's `scipy.optimize.fmin` function.

```
th_opt = fmin(loss, np.array([0.1]))[0]  # Initial guess = 0.1
```

The pursuit is successful and the global minima for the function is at `(0.00125, 0.018667)`. This means choosing a variance threshold of 0.00125 would yield an accuracy of $1 - 0.018667 = 98.13\%$.

(**Conclusion**) The process of computing the SVD of the data matrix followed by the second principal component and its variance is significantly faster and less CPU intensive compared to the activity recognition pipeline. Thus we first check the presence of an activity using the variance-thresholding approach and when an activity is detected, we run the activity recognition pipeline involving Rocket and Ridge Classifier Pool.

7 | Hardware Deployment

7.1 | Power

csi_send - Powered by a standard power bank

csi_recv - A Raspberry Pi Zero 2 W development board is powered by a standard power bank and the ESP32 module flashed with `csi_recv` is connected to the Pi as illustrated in Figure 2.1. ESP32 is powered by the 5V GPIO pins on the Pi.

7.2 | CSI Data Ingestion

The ESP32 is registered as a serial device under `/dev/tty` subsystem on the Pi. Programming the serial port appropriately and reading the serial port will expose the raw CSI data.

However all the models and analysis on the CSI data was performed in chunks rather than the entire recording. The requirement now is to implement this ”chunking” process realtime on a stream of incoming data. This is accomplished using a circular FIFO.

7.3 | The EmLog Project

EmLog [8] is an open-source project that caught our attention while searching for ways to easily implement and configure the circular buffer. From the project readme,

> emlog is a Linux kernel module that makes it easy to access the most recent (and only the most recent) output from a process. The emlog kernel module implements simple character device driver. The driver acts like a named pipe that has a finite, circular buffer. The size of the buffer is easily configurable. As more data is written into the buffer, the oldest data is discarded. Non-blocking reads are also supported, if a process needs to get the current contents of the log without blocking to wait for new data.

It had all the characteristics we were looking for - simple, fast and easily configurable. While initializing the character device, we are required to specify the size of the buffer required. By trial and error, it was found that a buffer of size 235KB holds nearly 256 CSI entries.

The setup is shown below. A nonblocking read of `/tmp/csififo` will now yield the most recent, approx. 256 lines of CSI data received from the ESP.

```
# Load emlog kernel module
sudo insmod emlog
# Create a character device ‘/tmp/csififo‘, 235KB buffer size, owned by user
# with UID==1000 and file permissions 0644
sudo mkemlog /tmp/csififo 235 0644 1000

# Program serial port - 921600 baud, 8 data bits, 1 stop bit and no parity bit
stty -F /dev/esp32 921600 cs8 -cstopb -parenb

# Populate the FIFO with data from ESP32
cat < /dev/esp32 | awk NF > /tmp/csififo
```

7.4 | System Architecture

The final system architecture, end-to-end, is illustrated in Figure 7.1

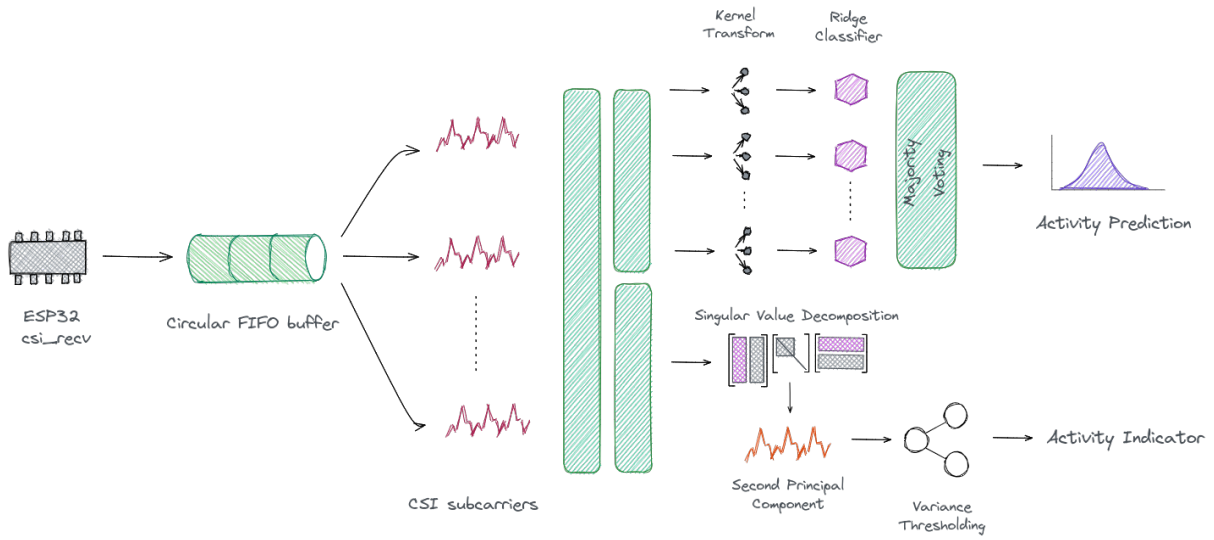


Figure 7.1: System Design, end-to-end

7.5 | Publishing Predictions

The predictions (along with useful metadata) from the pipeline illustrated in Figure 7.1 are broadcasted over websockets in JSON format to all clients via a websocket broadcast server. This feature is aimed to be used for visualizing the predictions at rendering clients that ingest the data over websockets.

8 | Results

The system was successfully deployed and validated realtime on a Raspberry Pi Zero 2 W running Raspbian OS Lite (64 bit) OS and two ESP32 modules (ESP32-WROOM-32) flashed with CSI sender and receiver programs (obtained from official sources) respectively.

The predictions were rendered and visualized on a client in the same network as the RPi and the predictions were observed to be stable, reliable and accurate.

The source code involved in the project from initial data analysis and experiments (in the form of IPython notebooks) to the final modules, scripts, ESP binaries and other program configuration files are made available open-source[1], extensively documented.

[1] https://github.com/winwinashwin/CSI-Sense-Zero

9 | References

[1] W. Wang, A. X. Liu, M. Shahzad, K. Ling, and S. Lu, "Understanding and modeling of wifi signal based human activity recognition," in *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15, (New York, NY, USA), p. 65–76, Association for Computing Machinery, 2015.

[2] M. A. A. Al-qaness, F. Li, X. Ma, Y. Zhang, and G. Liu, "Device-free indoor activity recognition system," *Applied Sciences*, vol. 6, no. 11, 2016.

[3] A. Dayal, S. R. Yeduri, B. H. Koduru, R. K. Jaiswal, J. Soumya, M. B. Srinivas, O. J. Pandey, and L. R. Cenkeramaddi, "Lightweight deep convolutional neural network for background sound classification in speech signals," *The Journal of the Acoustical Society of America*, vol. 151, pp. 2773–2786, 04 2022.

[4] espressif, "esp-csi." https://github.com/espressif/esp-csi.

[5] "Wi-fi driver - esp32 - esp-idf programming guide latest documentation." https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/wifi.html#wi-fi-channel-state-information. (Accessed on 05/06/2023).

[6] L. van der Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 11 2008.

[7] A. Dempster, F. Petitjean, and G. I. Webb, "ROCKET: exceptionally fast and accurate time series classification using random convolutional kernels," *CoRR*, vol. abs/1910.13051, 2019.

[8] N. Pavel, "Emlog." https://github.com/nicupavel/emlog.

[9] "Ieee standard for information technology–telecommunications and information exchange between systems - local and metropolitan area networks–specific requirements - part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications," *IEEE Std 802.11-2020 (Revision of IEEE Std 802.11-2016)*, pp. 1–4379, 2021.

[10] Y. Ma, G. Zhou, and S. Wang, "Wifi sensing with channel state information: A survey," *ACM Comput. Surv.*, vol. 52, jun 2019.

A | Appendix: Channel State Information (CSI)

The Wi-Fi technology, based on the IEEE 802.11n/ac standards [1], uses Orthogonal Frequency Division Multiplexing (OFDM) which decomposes the spectrum into multiple subcarriers with a symbol transmitted over each subcarrier. Channel state information (CSI) reflects how each subcarrier is affected through the wireless communication channel.

CSI in a Multiple-Input Multiple-Output (MIMO) setup has spatial diversity due to using multiple antennas as well as frequency diversity due to using multiple subcarriers per antenna. Each CSI sample, measured at the baseband, is a vector of complex variables where the size of the vector is the number of subcarriers in the OFDM signal.

More precisely, CSI represents the change of the signal from the transmitter (denote it as x) to the receiver (denote it as y) and the Wi-Fi channel in the frequency domain can be represented as;

$$y = Hx + n \tag{A.1}$$

where H is a complex matrix consisting of CSI values and n is the channel noise. The CSI is estimated for each Orthogonal Frequency Division Multiplexing (OFDM) subcarrier links [9]. OFDM splits the total frequency spectrum into 56 or 114 frequency subcarriers for a channel bandwidth of 20 and 40 MHz respectively. The CSI for each subcarrier is:

$$h = |h|e^{j\theta} \tag{A.2}$$

where $|h|$ represents the amplitude and θ the phase. To measure CSI, the transmitter sends Long Training Symbol (LTS), which contains pre-defined information for each subcarrier. When the receiver receives the LTS, it estimates the CSI having the difference between the original and received LTS. However, in real-world systems, the CSI is affected by a multi-path channel, receiver/transmit processing, hardware and software errors [10].

B | Appendix: t-distributed Stochastic Neighbour Embedding