From Motion to Notion: Teaching a Machine to Recognize Your Activities

Clara Wong
McMaster University
Hamilton, Canada
wongc148@mcmaster.ca

***Abstract*—This document presents a real-time human activity recognition (HAR) system executed on a Raspberry Pi, collecting accelerometer data. During preprocessing, nine statistical features were extracted from sliding windows, and a support vector machine (SVM) classifier was trained. The model was able to successfully classify sitting, walking, running, and turning with high accuracy. Visualization plots, such as PCA and feature distribution, corroborate the strong separability between the activity classes. Future work includes incorporating data from other sensors, collecting a larger and more diverse training dataset, and exploring advanced models to improve accuracy and generalizability.**

I. Introduction

Cyber-Physical Systems (CPS) integrate sensing, computation, and actuation in a closed feedback loop to enable real-time interaction between the real-world and computational network. CPS technologies are crucial in many biomedical applications where continuous physiological sensing coupled with real-time decision-making is necessary, such as wearable health monitors, rehabilitation devices, and fall-protection systems [1]. Human Activity Recognition (HAR) is the process of using sensor data from devices and machine learning algorithms to detect and classify human activities [2]. This project explores the design, implementation, and validation of a comprehensive CPS for real-time HAR using a Raspberry Pi and the Sense HAT's onboard accelerometer. The result was an end-to-end HAR pipeline, starting with data acquisition, followed by model training and evaluation, and culminating in deployment on a Raspberry Pi with LED-based activity feedback. The challenges associated with real-time classification on a computationally limited device like the Raspberry Pi were also explored.

II. Methodology

A. Data Collection and Pre-Processing

Four activities were classified for this project: sitting, walking, turning clockwise, and running. To achieve this, accelerometer data was collected with the Raspberry Pi Sense HAT v2. It utilizes the STMicro LSM9DS1 chip, an inertial measurement unit (IMU) that includes an accelerometer. The IMU has a 16 g acceleration measurement range with a 16-bit resolution [3]. To collect data, the Raspberry Pi was placed at waist level; studies found that a waist-mounted accelerometer best represented human motion, and several studies obtained an accuracy score exceeding 90% [4]. Sampling frequency is also an important consideration. Karantonis et al. reported that all measured body measurements are contained within frequency components below 20 Hz [5]. According to the Nyquist sampling theorem, this means a sampling frequency of at least 40 Hz would be sufficient to capture signal information and avoid aliasing. A sampling frequency of 50 Hz was chosen because a study by Jaen-Vargas et al. found that a window length of 1-2 seconds at 50 Hz was the optimal combination of window size and sampling rate for motion recognition, providing the best trade-off between recognition performance and speed [6]. Therefore, the data were segmented into window lengths of 2 seconds; since the sampling frequency is 50 Hz, the window length is set to 100 samples. The sliding window technique was employed, with an overlap of 75%, as a study by Nurwulan and Jiang found that this provided the highest accuracy with similar features (mean and standard deviation of acceleration) used [7]. Each window was classified based on a majority vote. For example, if 70 samples were labelled as walking and 30 samples were labelled as sitting, the overall window would be classified as walking. Data not labelled with an activity during data collection (containing 'NaN') were cleaned from the dataset.

B. Model Description

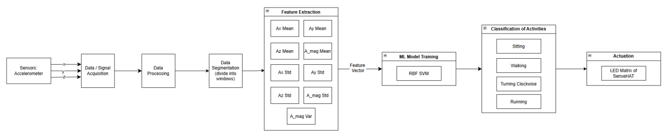


Fig. 1. System Architecture of the HAR system.

The HAR model pipeline consists of sensor data acquisition, data preprocessing, feature extraction, classification, and feedback. Data is collected from a waist-mounted Raspberry Pi SenseHAT. Data is then segmented into overlapping windows. Features are extracted for each window, and a radial basis function (RBF) support vector machine (SVM) classifies the sample into one of the following four activities: sitting, walking, turning, or running. The trained model is deployed on the Raspberry Pi for real-time classification, with predicted activities displayed on the SenseHAT's LED matrix. An overview of the system architecture is shown in Fig. 1.

More in-depth information about the model is shown in the sequence diagram (Fig. 10) and class diagram (Fig. 11) in the Supplemental Information section.

C. Training

A radial basis function (RBF) support vector machine (SVM) model was trained, specifically a one-vs-one classifier. A one-vs-one (OvO) classifier was chosen over a one-vs-rest (OvR) classifier for several reasons. First, the number of classes (four activities) is small. The OvO approach is preferred for small-class models since pairwise comparisons are done, resulting in higher accuracy for SVMs with non-linear kernels like RBF. Additionally, the training data contains an unequal number of samples for each activity. An OvO would not suffer as much from class imbalance as OvR does, which would lead to a biased model that favours the larger class. Finally, the training data is relatively small, so computational cost was not a limiting factor [8].

The RBF SVM in scikit-learn inherently uses a hinge-loss function, which penalizes misclassified samples and samples that are too close to the decision boundary. Since HAR signals are inherently noisy, the hinge-loss function is appropriate because it creates a large margin while still allowing margin violations when necessary. The C parameter in the SVM function controls the trade-off between achieving a wider margin and minimizing classification loss. A larger C parameter focuses more on correctly classifying a training sample, while a smaller C emphasizes a wider margin [9].

To determine the best hyperparameters, a grid search was performed for two kernel options ('rbf' and 'linear') and three C values (1, 10, 100). The best parameters were determined as $kernel =' rbf'$ and $C = 10$. This indicates that non-linear decision boundaries with a moderate C value provide the best trade-off between margin size and classification performance for the collected dataset.

D. Feature Engineering

The SVM model was trained on nine features:

- mean acceleration in the x-axis, y-axis, z-axis, and overall magnitude;
- standard deviation of acceleration in the x-axis, y-axis, z-axis, and overall magnitude;
- variance in the acceleration magnitude.

The mean and standard deviation were taken for all tri-axial directions to ensure important directional information is maintained. The activities have defining directional patterns that would be lost if only the mean and standard deviation of the acceleration magnitude were considered. For example, turning and walking may produce similar magnitudes of acceleration, but they can be differentiated after breaking down each acceleration component. While walking may produce periodic, vertical oscillations in the z-axis, turning involves the body rotating around the vertical axis, which involves more lateral (x- and y-axis) acceleration changes. Only the variance of the magnitude of acceleration was considered for the model since movement intensity is captured in the magnitude value. The features were then scaled before training the model on them.

E. Validation Model

A 10-fold cross-validation model was used to validate the model. A K-fold cross-validation model was chosen because it efficiently uses a small dataset by ensuring every data point is used for both testing and training. Moreover, K-fold cross-validation provides reliable estimates, reducing the likelihood of overfitting or underfitting. Furthermore, it works well on small datasets compared to other validation models. A larger K value was chosen (K = 10 rather than K = 5) because it offers the best trade-off between bias and variance. A large K value is also more effective on small datasets, since each fold uses 90% of the data for training, leading to a less biased estimate of the model's performance [10].

F. HAR Protocol

The procedure for designing and implementing this model is as follows:

1) *Data Acquisition*: acquiring data from the SenseHAT accelerometer sensors. The data is collected by placing the Raspberry Pi at waist level and recording the four activities (sitting, walking, running, turning) consecutively, with labelling achieved by using the push button. A sampling frequency of 50 Hz was used. The activities were repeated multiple times to get a wide range of data for each activity.
2) *Data Segmentation*: divide the data into two-second windows (each containing 100 samples at a sampling rate of 50 Hz). Assign to each window the label of the activity that appears most frequently within that interval. Use a step size of 25 samples to create sliding windows, resulting in a 75% overlap between consecutive windows.
3) *Feature Extraction:* within each window, compute the features the model is trained on, as mentioned in the Feature Engineering section. Then, scale the features before using them for training.
4) *Training*: tune the hyperparameters (kernel and C parameter) via grid search; train the SVM model with the best parameters.
5) *Validation*: validate the model using the k-fold cross-validation model. A 10-fold model was used.
6) *Evaluate on Test Data*: apply the final trained model on the test set and visualize the performance using metrics such as the confusion matrix, principal component analysis (PCA) plot, and classification report.
7) *Real-time Classification*: deploy the trained model and real-time data acquisition pipeline on the Raspberry Pi to achieve real-time activity [11].

III. Results

The validation results from the 10-fold cross-validation showed high accuracy, as demonstrated by the results seen in Table I and Fig. 2 and Fig. 3. The mean accuracy was 98.9%.

The boxplot depicts the misclassified samples, with walking and running having a couple of inaccurate predictions.

TABLE I
10-Fold Cross-validation Accuracy Results

Fold Number	***Accuracy***
1	100%
2	100%
3	97.4%
4	100%
5	100%
6	97.4%
7	100%
8	100%
9	97.3%
10	97.3%
Mean Accuracy	98.9%

The test results also reflected the high accuracy seen in the validation model. In Table II, the precision, recall, and f1-scores for all activities were 100%, indicating that all windows in the test set were classified correctly.

TABLE II
Classification Report

	Precision	***Recall***	***f1-score***	***Support***
Run	1.00	1.00	1.00	18
Sit	1.00	1.00	1.00	25
Turn CW	1.00	1.00	1.00	35
Walk	1.00	1.00	1.00	17

The high accuracy is also reflected in the confusion matrix (Fig. 4), where all predicted classes match the actual outcome. The separability of the activities is also reflected in the PCA and distribution plots, as outlined in the Discussion section.

In addition to testing the model on the test data, a live demonstration of the real-time HAR classification was completed. The CPS was able to accurately detect sitting and running. However, the CPS showed some difficulty with differentiating between walking and turning clockwise.

IV. Discussion

Fig. 5 depicts a PCA plot. In the PCA plot, the activity clusters are, for the most part, separated from each other, indicating a degree of difference between the classes. Sitting has two tight clusters that are well separated from the other dynamic activities, indicating low variance within the class. The low variance is expected for sitting, and the two clusters may correspond to different sitting postures that result in different acceleration values. Running also forms a well-separated cluster from the other activities. The cluster is not as tight, indicating greater variability in acceleration during running. Walking and turning display some overlap, which is expected since they are comparable in movement intensity. However, the clusters are still distinct enough for the SVM to find a separating boundary. The walking cluster is moderately spread out, indicating a higher variability than sitting and turning. Turning creates tight clusters, indicating low variance, and the two clusters may again be due to differences in posture during data acquisition.

In addition to the PCA plot, the following distribution graphs for the acceleration magnitude for each activity were also created:

1) Distribution of acceleration magnitude (Fig. 6). The violin plot shows little spread in sitting acceleration, high spread in running acceleration, and moderate spread for walking and turning. The walking and turning plots show wider sections at approximately the same magnitude of acceleration. This suggests that other features are needed to differentiate between these two activities.
2) Distribution of mean acceleration magnitude (Fig. 7). Running had the highest mean acceleration, sitting had the lowest, and walking and turning had overlapping mean acceleration ranges, with turning being slightly higher. Mean acceleration shows the overall level of motion. The mean distribution plot demonstrates that running is high-intensity, turning and walking are moderate intensity, and sitting is low-intensity. It provides a baseline for activity intensity, clearly separating static and dynamic activities.
3) Distribution of acceleration magnitude standard deviation (Fig. 8). The plot shows that running had the highest standard deviation of acceleration, followed by walking, turning, and sitting with the lowest standard deviation. Walking and turning had overlapping standard deviation ranges, with walking having an overall slightly higher standard deviation. The standard deviation of acceleration indicates the spread of the acceleration values, with a higher standard deviation indicating large fluctuations (correlated with high-intensity activities) and a lower standard deviation indicating small fluctuations (correlated with low-intensity activities). These patterns are reflected in the violin plot.
4) Distribution of acceleration magnitude variance (Fig. 9). Similar to the standard deviation distribution, running had the highest variance in acceleration, followed by walking, turning, and sitting with the lowest variance. The variance of acceleration measures how much acceleration fluctuates around the mean. A high variance is expected for high-intensity activities and low variance for low-intensity activities; this is reflected in the violin plot. This reflects the complexity and variability of the activity, helping to differentiate between activities with similar mean accelerations.

Overall, activities were recognized with a high degree of accuracy, as demonstrated by the classification report results and plots of the confusion matrix, PCA, and distribution of acceleration magnitude for various features. The PCA and feature distribution plots provide insight into why walking and turning clockwise were occasionally misclassified during the real-time HAR demonstration. In both visualizations, the feature clusters for walking and turning displayed some overlap, reflecting their similar motion intensity, explaining the classifier's difficulty in distinguishing between them. The limited dataset and minor inconsistencies in the positioning of the Raspberry Pi during data acquisition may also explain the marginal differences observed in the acceleration data. To mitigate this issue in the future, other sensors could be incorporated. For instance, a gyroscope captures angular velocity and would help differentiate rotational motion from walking. Additional features should also be explored. While mean, standard deviation, and variance performed well for this dataset, a broader range of features should be evaluated to identify the most discriminative ones. Potential features to consider include jerk (derivative of acceleration), signal energy, entropy, and frequency-domain features like spectral power. Expanding the dataset to include more trials and a variety of subjects would also further improve model generalization and improve the classification accuracy across the general population.

V. Conclusion

This project demonstrates the successful development of a CPS for HAR using a Raspberry Pi, SenseHat, and a machine-learning classification pipeline. The system achieved high accuracy on the test data, strong cross-validation accuracy, and clear separation between most activities in the PCA and distribution plots; it was able to reliably classify high-intensity (running) activities from moderate-intensity (walking and turning clockwise) and low-intensity (sitting) activities. The results demonstrate that a lightweight system like the Raspberry PI can perform accurate real-time activity recognition with the right features and preprocessing steps.

However, a closer inspection of the PCA and feature distribution plots, along with the live demonstration of the HAR, highlights some limitations. Walking and turning show overlapping feature distributions, which led to some misclassifications in the demonstration. The overlap is likely due to the comparable motion intensities of walking and turning, combined with the limited discriminative capability of features based solely on acceleration. Additionally, the dataset was collected from a single subject; therefore, the model is likely not generalizable to the broader population. This highlights the importance of building an HAR system that incorporates a variety of sensors, robust features, and is trained on larger datasets to develop a universally accurate system.

These issues can be addressed in the future by incorporating a gyroscope to better capture rotation motion, which could more effectively separate the activities, especially walking and turning. Additionally, expanding the feature set to determine the most discriminative features can also enhance the model's accuracy. Moreover, collecting data from multiple subjects in different environments would improve the model's robustness and reduce bias towards a specific movement pattern. Furthermore, other advanced models aside from SVM (eg. Convolutional Neural Networks, Bayesian Neural Networks, Random Forest) should be explored to help improve the existing model. Overall, these improvements would improve the reliability and scalability of the system when used in real-world applications in the healthcare field.

References

[1] "Cyber-physical systems - an overview," https://www.sciencedirect.com/topics/engineering/cyber-physical-systems, accessed: Dec. 01, 2025.

[2] H. F. Nweke, Y. W. Teh, M. A. Al-Garadi, and U. R. Alo, "Deep learning algorithms for human activity recognition using mobile and wearable sensor networks: State of the art and research challenges," *Expert Systems with Applications*, vol. 105, pp. 233–261, Sep. 2018.

[3] "Raspberry pi sense hat v2," https://www.pishop.ca/product/raspberry-pi-sense-hat-v2/, 2025, accessed: Dec. 01, 2025.

[4] F. Attal, S. Mohammed, M. Dedabrishvili, F. Chamroukhi, L. Oukhellou, and Y. Amirat, "Physical human activity recognition using wearable sensors," *Sensors*, vol. 15, no. 12, pp. 31 314–31 338, 2015.

[5] "Implementation of a real-time human movement classifier using a triaxial accelerometer for ambulatory monitoring," https://ieeexplore.ieee.org/abstract/document/1573717, accessed: Oct. 26, 2025.

[6] M. Jaén-Vargas *et al.*, "Effects of sliding window variation in the performance of acceleration-based human activity recognition using deep learning models," *PeerJ Computer Science*, vol. 8, p. e1052, 2022.

[7] N. R. Nurwulan and B. C. Jiang, "Window selection impact in human activity recognition," *International Journal of Innovative Technology and Interdisciplinary Sciences*, vol. 3, no. 1, pp. 381–394, 2020.

[8] GeeksforGeeks, "Multiclass classification using support vector machines (svm)," https://www.geeksforgeeks.org/machine-learning/multi-class-classification-using-support-vector-machines-svm/, 2024.

[9] ——, "Rbf svm parameters in scikit learn," GeeksforGeeks, 07 2023. [Online]. Available: https://www.geeksforgeeks.org/python/rbf-svm-parameters-in-scikit-learn/

[10] ——, "K fold cross validation in machine learning," GeeksforGeeks, 04 2025. [Online]. Available: https://www.geeksforgeeks.org/machine-learning/k-fold-cross-validation-in-machine-learning/

[11] A. Ferrari *et al.*, "Trends in human activity recognition using smartphones," *Journal of Reliable Intelligent Environments*, vol. 7, no. 3, pp. 189–213, 2021.

VI. Supplemental Information

A. Data Plots

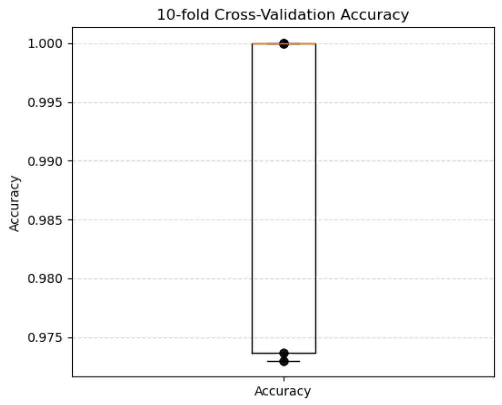


Fig. 2. Box plot of 10-fold cross-validation accuracy.

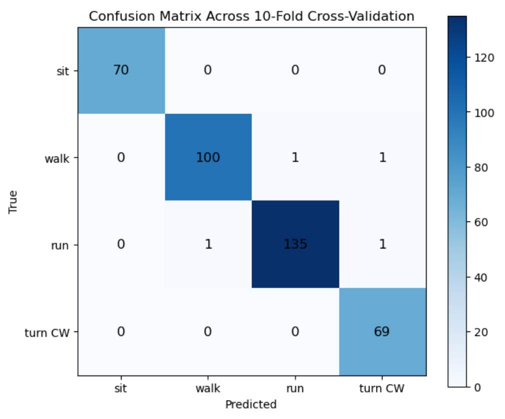


Fig. 3. Confusion matrix for 10-fold cross-validation.

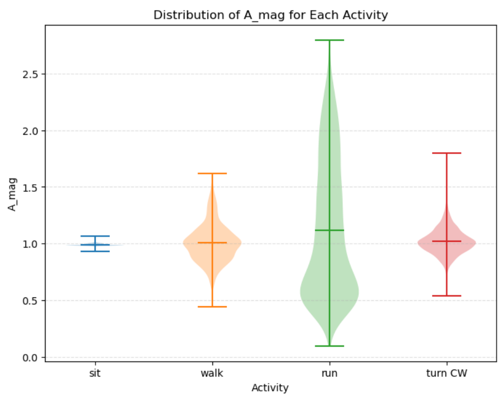


Fig. 6. Distribution of A_Mag for each activity.

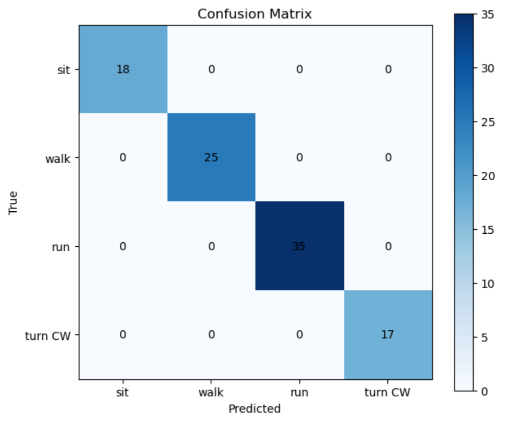


Fig. 4. Confusion matrix of test data.

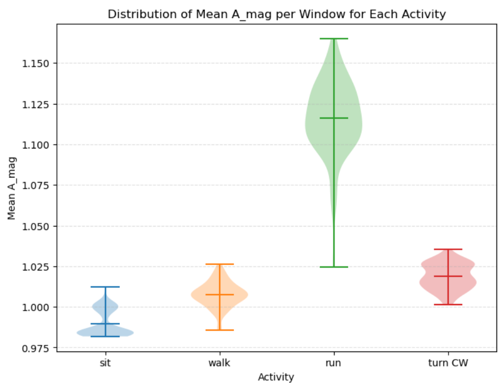


Fig. 7. Distribution of mean A_mag for each activity.

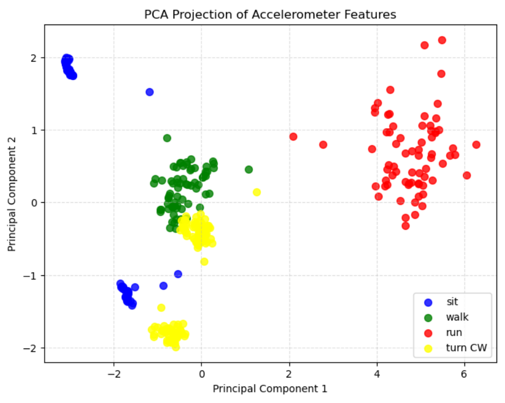


Fig. 5. PCA projection of accelerometer features.

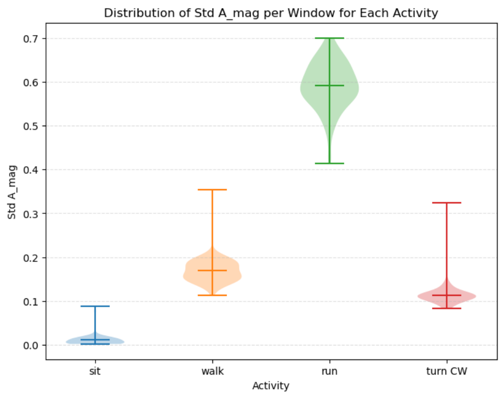


Fig. 8. Distribution of A_Mag standard deviation for each activity.

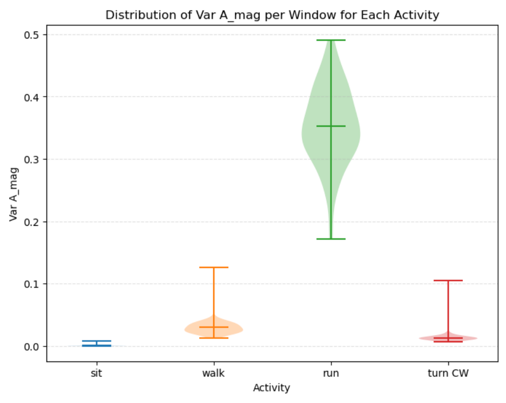
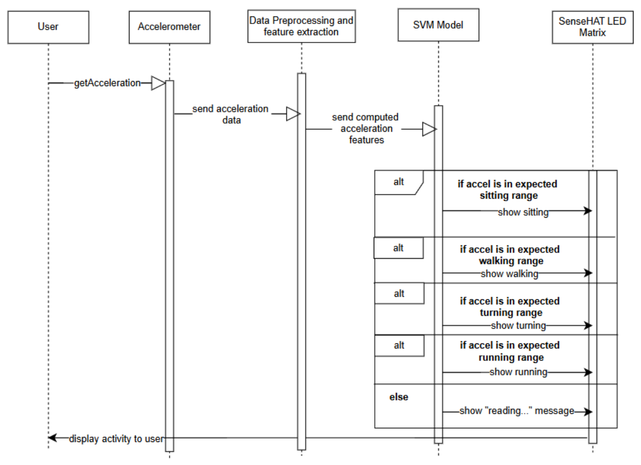


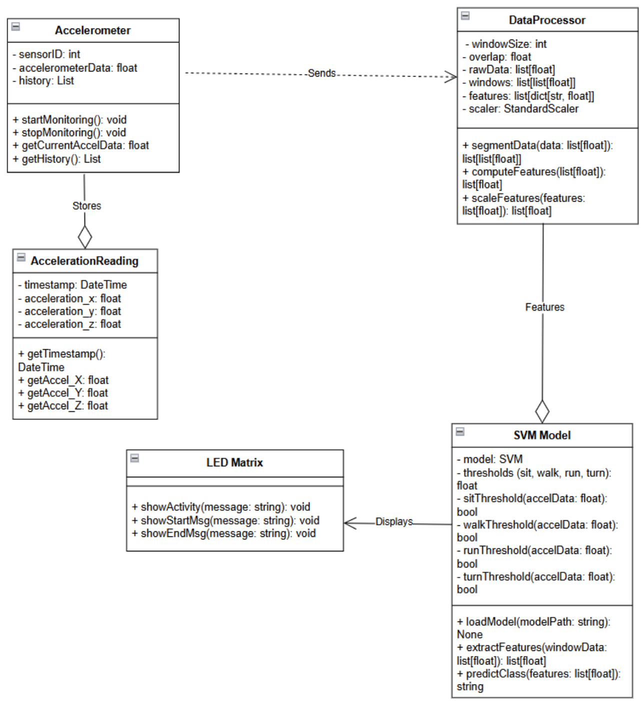
Fig. 9. Distribution of A_Mag variance for each activity.

B. UML of CPS



User
Accelerometer
Data Preprocessing and feature extraction
SVM Model
SenseHAT LED Matrix
getAcceleration
send acceleration data
send computed acceleration features
alt
if accel is in expected sitting range
show sitting
alt
if accel is in expected walking range
show walking
alt
if accel is in expected turning range
show turning
alt
if accel is in expected running range
show running
else
show "reading..." message
display activity to user

Fig. 10. Sequence Diagram of the HAR system.



Accelerometer
- sensorID: int
- accelerometerData: float
- history: List
+ startMonitoring(): void
+ stopMonitoring(): void
+ getCurrentAccelData: float
+ getHistory(): List
Sends
DataProcessor
- windowSize: int
- overlap: float
- rawData: list[float]
- windows: list[list[float]]
- features: list[dict[str, float]]
- scaler: StandardScaler
+ segmentData(data: list[float]): list[list[float]]
+ computeFeatures(list[float]): list[float]
+ scaleFeatures(features: list[float]): list[float]
Stores
AccelerationReading
- timestamp: DateTime
- acceleration_x: float
- acceleration_y: float
- acceleration_z: float
+ getTimestamp(): DateTime
+ getAccel_X: float
+ getAccel_Y: float
+ getAccel_Z: float
Features
LED Matrix
+ showActivity(message: string): void
+ showStartMsg(message: string): void
+ showEndMsg(message: string): void
Displays
SVM Model
- model: SVM
- thresholds (sit, walk, run, turn): float
- sitThreshold(accelData: float): bool
- walkThreshold(accelData: float): bool
- runThreshold(accelData: float): bool
- turnThreshold(accelData: float): bool
+ loadModel(modelPath: string): None
+ extractFeatures(windowData: list[float]): list[float]
+ predictClass(features: list[float]): string

Fig. 11. Class Diagram of the HAR system.

C. Source Code

Python Script for Training and Validating SVM Model

1) Setup:

```python
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.multiclass import OneVsOneClassifier
from sklearn.model_selection import cross_val_score, GridSearchCV, train_test_split, cross_val_predict
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import numpy as np
import joblib
```

2) Load Data and Window Preparation:

```python
data = pd.read_csv('training_data.csv')

# Filter for the activities
data = data[data['activity'].isin(['sit', 'walk', 'run', 'turn CW'])]

# Prepare features (windowing)
WINDOW_SIZE = 100 # 2 seconds at 50Hz
def compute_window_features(ax, ay, az):
    ax = np.array(ax)
    ay = np.array(ay)
    az = np.array(az)
    a_mag = np.sqrt(ax**2 + ay**2 + az**2)
    return [np.mean(ax), np.mean(ay), np.mean(az), np.mean(a_mag),
            np.std(ax), np.std(ay), np.std(az), np.std(a_mag),
            np.var(a_mag)]

STEP_SIZE = WINDOW_SIZE // 4  # 75% overlap

windows_features = []
windows_labels = []
windows_activities = []

# Loop with step of 25 steps
for start in range(0, len(data) - WINDOW_SIZE, STEP_SIZE):
    window = data.iloc[start : start + WINDOW_SIZE]

    ax = window['Ax'].values
    ay = window['Ay'].values
    az = window['Az'].values
    acts = window['activity'].values

    # Compute features for the window
    features = compute_window_features(ax, ay, az)
    windows_features.append(features)

    # Label by most common activity in the window
    most_common = window['activity'].mode()[0]
    windows_labels.append(most_common)

    windows_activities.append(list(acts))


X = np.array(windows_features)
y = np.array(windows_labels)
```

3) Data Preparation, Tune Hyperparameters:

```python
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Hyperparameter tuning with GridSearchCV
param_grid = {
    'estimator__C': [1, 10, 100],
    'estimator__kernel': ['linear', 'rbf'],
    'estimator__probability': [True]
}

grid = GridSearchCV(
    OneVsOneClassifier(SVC(probability=True)),
    param_grid,
    cv=5,
    scoring='accuracy'
)

grid.fit(X_train_scaled, y_train)

print("Best params:", grid.best_params_)
print("Best CV score:", grid.best_score_)
```

4) Train and Validate Model:

```python
# train SVM with best params, one-vs-one classification
model = OneVsOneClassifier(SVC(kernel=grid.best_params_['estimator__kernel'], C=grid.best_params_['estimator__C'], probability=True))
model.fit(X_train_scaled, y_train)

# SVM Model Validation
# Validate with 10-fold cross-validation
# use k = 10 since smaller dataset
scores = cross_val_score(
    OneVsOneClassifier(SVC(kernel=grid.best_params_['estimator__kernel'], C=grid.best_params_['estimator__C'], probability=True)),
    X_train_scaled,
    y_train,
    cv=10,
    scoring='accuracy'
)

y_pred_cv = cross_val_predict(
    OneVsOneClassifier(SVC(kernel=grid.best_params_['estimator__kernel'], C=grid.best_params_['estimator__C'], probability=True)),
    X_train_scaled,
    y_train,
    cv=10
)

print("Cross-validation accuracy scores:", scores)
print("Mean CV accuracy:", scores.mean())

# Plot cross-validation scores
plt.figure(figsize=(6, 5))
plt.boxplot(scores, vert=True)

# Also plot individual points
plt.scatter(
    [1]*len(scores), scores,
    color='black', s=40, alpha=0.7
)

plt.title("10-fold Cross-Validation Accuracy")
plt.ylabel("Accuracy")
plt.grid(True, axis='y', linestyle='--', alpha=0.5)
plt.xticks([1], ['Accuracy'])
plt.show()

# Confusion matrix for CV predictions
cm_cv = confusion_matrix(y_train, y_pred_cv)
classes = ["sit", "walk", "run", "turn CW"]

plt.figure(figsize=(7,6))
plt.imshow(cm_cv, cmap="Blues")
plt.colorbar()

plt.xticks(np.arange(len(classes)), classes)
plt.yticks(np.arange(len(classes)), classes)

for i in range(len(classes)):
    for j in range(len(classes)):
        plt.text(j, i, cm_cv[i,j], ha='center', va='center', color='black', fontsize=12)

plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("Confusion Matrix Across 10-Fold Cross-Validation")
plt.show()
```

5) Evaluate Test Set:

```python
# evauluate test set
y_pred = model.predict(X_test_scaled)
print("Classification Report:\n", classification_report(y_test, y_pred))

# Visualize on test data
# a) confusion matrix
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", cm)

# Visualize Confusion Matrix
plt.figure(figsize=(7,6))
plt.imshow(cm, cmap="Blues")
plt.colorbar()

classes = ["sit", "walk", "run", "turn CW"]

# Axis labels
plt.xticks(np.arange(4), classes)
plt.yticks(np.arange(4), classes)
# label squares
for i in range(4):
    for j in range(4):
        plt.text(j, i, cm[i, j], ha="center", va="center", color="black")

plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("Confusion Matrix")
plt.show()

# b) PCA Visualization of Feature Space
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_train_scaled)

plt.figure(figsize=(8, 6))
# Use a fixed activity order and explicit color mapping so plots are consistent
activity_list = ['sit', 'walk', 'run', 'turn CW']
color_map = {'sit': 'blue', 'walk': 'green', 'run': 'red', 'turn CW': 'yellow'}

for act in activity_list:
    col = color_map.get(act, 'gray')
    idx = (y_train == act)
    plt.scatter(
        X_pca[idx, 0],
        X_pca[idx, 1],
        label=act,
        color=col,
        s=50,
        alpha=0.8
    )

plt.title("PCA Projection of Accelerometer Features")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.legend()
plt.grid(True, linestyle='--', alpha=0.4)
plt.show()

# c) feature distribution per activity using A_Mag only
plt.figure(figsize=(8, 6))

for act in activity_list:
    values = data[data['activity'] == act]['A_mag']
    plt.violinplot(values, positions=[activity_list.index(act)], showmeans=True)

plt.title("Distribution of A_mag for Each Activity")
plt.xlabel("Activity")
plt.ylabel("A_mag")
plt.xticks(range(len(activity_list)), activity_list)
plt.grid(True, axis='y', linestyle='--', alpha=0.4)
plt.show()

# d) feature distribution using mean A_mag per window
plt.figure(figsize=(8, 6))
mean_a_mag_per_window = []

for features in windows_features:
    mean_a_mag_per_window.append(features[3]) # mean A_mag is the 4th feature
mean_a_mag_per_window = np.array(mean_a_mag_per_window)
for act in activity_list:
    values = mean_a_mag_per_window[np.array(windows_labels) == act]
    plt.violinplot(values, positions=[activity_list.index(act)], showmeans=True)

plt.title("Distribution of Mean A_mag per Window for Each Activity")
plt.xlabel("Activity")
plt.ylabel("Mean A_mag")
plt.xticks(range(len(activity_list)), activity_list)
plt.grid(True, axis='y', linestyle='--', alpha=0.4)
plt.show()

# e) feature distribution using std of A_mag per window
plt.figure(figsize=(8, 6))
std_a_mag_per_window = []

for features in windows_features:
    std_a_mag_per_window.append(features[7]) # std A_mag is the 8th feature
std_a_mag_per_window = np.array(std_a_mag_per_window)
for act in activity_list:
    values = std_a_mag_per_window[np.array(windows_labels) == act]
    plt.violinplot(values, positions=[activity_list.index(act)], showmeans=True)

plt.title("Distribution of Std A_mag per Window for Each Activity")
plt.xlabel("Activity")
plt.ylabel("Std A_mag")
plt.xticks(range(len(activity_list)), activity_list)
plt.grid(True, axis='y', linestyle='--', alpha=0.4)
plt.show()

# f) feature distribution using var A_mag per window
plt.figure(figsize=(8, 6))
var_a_mag_per_window = []

for features in windows_features:
    var_a_mag_per_window.append(features[8]) # var A_mag is the 9th feature
var_a_mag_per_window = np.array(var_a_mag_per_window)
for act in activity_list:
    values = var_a_mag_per_window[np.array(windows_labels) == act]
    plt.violinplot(values, positions=[activity_list.index(act)], showmeans=True)

plt.title("Distribution of Var A_mag per Window for Each Activity")
plt.xlabel("Activity")
plt.ylabel("Var A_mag")
plt.xticks(range(len(activity_list)), activity_list)
plt.grid(True, axis='y', linestyle='--', alpha=0.4)
plt.show()
```

Python Script for Data Acquisition

```
from sense_hat import SenseHat
from datetime import datetime
import joblib
import numpy as np
import csv, time, math

sense = SenseHat()
sense.clear()
#filename = "deliverable1_log_" + datetime.now().strftime("%H%M%S") + ".csv"
filename = "final_training_data.csv"
button_labels = {"up": "walk", "down": "sit", "left": "run", "right": "turn CW"}
activity = None
active = False
msg = ""
color_map = { "sit": [0, 0, 255],
                          "walk": [0, 255, 0],
                          "run": [255, 0, 0],
                          "turn CW": [255, 255, 0]
                          }
color = [255, 255, 255]

with open(filename, 'w', newline='') as f:
        writer = csv.writer(f)
        writer.writerow(["time", "Ax", "Ay", "Az", "A_mag", "active", "activity"])
        sense.show_message("START", text_colour=[255, 255, 255], scroll_speed=0.1)

        while True:
                for e in sense.stick.get_events():
                        if e.direction == "up" and e.action == "pressed" or e.direction == "down" and e.action == "pressed" or e.direction == "left" and e.action == "pressed" or e.direction == "right" and e.action == "pressed":
                                active = True
                                activity = button_labels [ e.direction ]
                                #msg = "sit" if activity == "sit" else "walk"
                                msg = activity
                                color = color_map.get(activity, [255, 255, 255])
                                print("activity:", activity)
                                sense.show_message ( msg , text_colour = color , scroll_speed =0.05)

                        elif e.direction == "middle" and e.action == "pressed":
                                sense.show_message("END", text_colour=[255, 255, 255], scroll_speed=0.05)
                                active = False
                                print("recording stopped")
                                sense.clear()
                                f.close()
                                exit()

                a = sense.get_accelerometer_r
                ↪  aw ()
                Ax , Ay , Az = a ['x'] , a
                ↪  ['y'] , a ['z']
                A_mag = math . sqrt ( Ax **2 +
                ↪  Ay **2 + Az **2)
                writer.writerow ([
                ↪  datetime.now().isoformat()
                ↪  , Ax , Ay , Az , A_mag,
                ↪  active, activity ])
                f.flush()
                # collect sample at 50Hz
                time.sleep (0.02)
```

Python Script for Real-Time Activity Recognition

```python
from sense_hat import SenseHat
import joblib
import numpy as np
import time
import math
from collections import deque

# Load trained SVM and scaler
bundle = joblib.load('svm_model.pkl')
model = bundle['model']
scaler = bundle['scaler']

sense = SenseHat()
sense.clear()

# LED colors for each activity
colors = {
    "sit": [0, 0, 255],
    "walk": [0, 255, 0],
    "run": [255, 0, 0],
    "turn CW": [255, 255, 0]
}

WINDOW_SIZE = 100
STEP_SIZE = WINDOW_SIZE // 4   # 75% overlap
sample_counter = 0 # counts number of samples
↪  to ensure step size

window_ax = deque(maxlen=WINDOW_SIZE)
window_ay = deque(maxlen=WINDOW_SIZE)
window_az = deque(maxlen=WINDOW_SIZE)
window_mag = deque(maxlen=WINDOW_SIZE)

# Feature extraction
def compute_features():
    """Compute statistics from the sliding
    ↪  window"""
    ax = np.array(window_ax)
    ay = np.array(window_ay)
    az = np.array(window_az)
    mag = np.array(window_mag)

    features = [
        np.mean(ax), np.mean(ay), np.mean(az),
        ↪  np.mean(mag),
        np.std(ax), np.std(ay), np.std(az),
        ↪  np.std(mag),
        np.var(mag)
    ]

    return np.array(features).reshape(1, -1)

# get accelerometer data
def get_features():
    a = sense.get_accelerometer_raw()
    Ax, Ay, Az = a['x'], a['y'], a['z']
    A_mag = math.sqrt(Ax**2 + Ay**2 + Az**2)
    return Ax, Ay, Az, A_mag

# Startup message
sense.show_message("READY", text_colour=[255,
↪  255, 255])
time.sleep(1)
sense.clear()

print("Starting real-time HAR...")

last_prediction = None  # Track last activity

while True:
    Ax, Ay, Az, A_mag = get_features()

    # Add to sliding windows
    window_ax.append(Ax)
    window_ay.append(Ay)
    window_az.append(Az)
    window_mag.append(A_mag)

    sample_counter += 1

    # Prediction only after window is full and
    ↪  step size reached
    if len(window_ax) == WINDOW_SIZE and
    ↪  sample_counter >= STEP_SIZE:
        sample_counter = 0  # reset counter
        ↪  after prediction

        X = compute_features()
        X_scaled = scaler.transform(X)
        prediction =
        ↪  model.predict(X_scaled)[0]

        # Print to terminal
        print("Activity:", prediction)

        # change LED color only if prediction
        ↪  changed
        if prediction != last_prediction:
            color = colors.get(prediction,
            ↪  [255, 255, 255])
            sense.clear(color)
            last_prediction = prediction

    # Joystick press down to stop
    for e in sense.stick.get_events():
        if e.direction == "middle" and
        ↪  e.action == "pressed":
            sense.show_message("END",
            ↪  text_colour=[255, 255, 255])
            print("Measurement stopped")
            sense.clear()
            exit()

    time.sleep(0.02)
```