Principles and Design of IoT Systems
Coursework 3 (2022-23)

Group F
Sean Strain - s1832137
Szymon Ciciala - s1834237
Yong Chen - s1909083

December 7, 2023

Abstract

Human Activity Recognition is a complex field of study focusing upon recognising human motion. This paper outlines the creation of a wearable sensor-based approach the Human Activity Recognition problem. Multiple machine learning models capable of distinguishing between 14 different kinds of ambulation activities are created with a quantitative analysis being conducted to evaluate the most successful model. The design process behind the creation of an accompanying Android mobile application is discussed. A reflection on the quality of the system is provided alongside suggestions for improvements to similar systems in the future.

Contents

1 Introduction

Human Activity Recognition (HAR) is an active field of research that focuses on interpreting human motion to identify activities, gestures, or behaviors. Be it ambulation (running, walking, etc.), transportation (riding a bus, riding a bike, etc.), gesturing (waving, raising a fist, etc.), or other activities, HAR involves using Machine Learning (ML) methods to classify raw data gathered from various sources to determine what a person is currently doing. The applications of HAR are numerous, including medical, military, and security applications. For instance, HAR could be utilised to detect abnormal activities in dementia patients and in doing so prevent them from coming to harm [43].

The sources of data used in HAR can be either *external* or *wearable* [33]. In the external case, cameras and sensors are placed in predetermined positions; the classification of activities is dependent on an individual's interaction with these sensors [30]. However, Lara and Labrador [33] note several drawbacks to these external sources. If the user is not in the area measured by the sensors the classification of their activities is impossible. Video sequences of an individual raise privacy concerns, in addition to necessitating the high computational complexity of video processing techniques.

These drawbacks motivate the use of the alternative: wearable sensors. Wearable sensors include devices that are carried by or attached to the user. Given the increase in availability, increase in power, and decrease in cost of microelectronics and computer systems in the past decade, wearable sensors have become increasingly accessible. These sensors are not limited to a certain area as external sensors are. Further, they do not require installation. This encourages the use of wearable sensors for HAR.

This paper outlines the creation of an end-to-end HAR system that utilises wearable sensors. This system comprises of three parts: wearable sensors that collect real-time accelerometer data from the user, a ML model that interprets said data and classifies it under a given list of activities, and the smartphone application that presents the outcome in a readable form for the user.

This paper was written as part of a course for the University of Edinburgh. The course organisers supplied two wearable sensors, namely the Nordic Thingy:52 [11] - a multi-sensor device with an accelerometer and magnetometer, and the RESpeck respiratory monitor [25] - a device containing a three-axis digital accelerometer, for the creation of the system. A skeleton mobile application was also provided, alongside a set of labelled movement data to be used in the training, testing, and validation of the ML model. There are 14 different labels given to this set of movement data:

- Sitting (upright, bent forward, leaning backwards)
- Lying down (left-side, right-side, on the back, on the front)
- Ascending and descending stairs
- Walking and running
- Standing
- Desk work
- General movement

This paper aims to explore the different machine learning methods that are applicable to HAR through investigating the relevant literature and to create highly accurate models based off of these methods.

The mobile application was created following the 6 stage design process recommended by Jakob Nielsen, a human-computer interaction expert [2]. This process is discussed in detail within the Methodology section. Within the *Define* stage, the following aims for the application were identified:

- Create a user-interface that users find intuitive.
- Facilitate the communication between the sensors and the smartphone.
- Provide a secure system for a user to create an account with the application.
- Allow for a user to store, view, and delete their historical data.
- Ensure a reasonable level of stability by limiting crashes and unexpected behaviours.

Finally, this paper will combine these machine learning models with the mobile application. In doing so, a full, end-to-end HAR system will be created that allows a user to connect to and use the wearable sensors to determine the user's current activity.

2 Literature Survey

The proliferation of mobile and wearable devices has enabled the development of various applications that track and monitor human activities. These applications make use of low-power Inertial Measurement Unit (IMU) sensors in mobile and wearable devices to perform human activity recognition (HAR). The sensory data collected by wearable devices during physical activities contains valuable information about the duration and intensity of the activity, which can reveal an individual's daily habits and health conditions. Activity monitoring can provide individuals with insights into their activity patterns and help them identify areas where they may need to make changes to improve their health and reduce the risk of chronic diseases. [20]
In the field of HAR, researchers have explored a diverse range of ML techniques in the past. These include algorithms such as KNNs, Decision Trees, SVMs and HMMs. While traditional techniques like SVMs and HMMs have been commonly used in HAR systems, deep-learning (DL) approaches have shown superior performance in recent years. These DL approaches have resulted in improved performance and increased accuracy in recognizing human activities, surpassing the results obtained from traditional ML techniques. A total of four types of DL approaches are investigated in this survey.

- **Autoencoder**

  Autoencoders are an unsupervised pre-training method for artificial neural networks (ANN) that consist of an encoder module and a decoder module. They have become a popular method for learning features in an unsupervised way and are often used to improve the performance of other networks and algorithms [41]. The encoder compresses the input data into a latent space, while the decoder transforms the data back into the original domain. They are commonly used for feature extraction, dimensionality reduction and anomaly detection. They are usually trained using mean squared error or mean squared error plus KL divergence loss functions.

  In Li et al.'s study, an autoencoder architecture that combined a sparse autoencoder and a denoising autoencoder was used to extract useful feature representations from accelerometer and gyroscope sensor data. Then they performed classification using support vector machines. The classification accuracy was compared with traditional methods such as Fast Fourier Transform and Principal Component Analysis. The results showed that the stacked autoencoder achieved the highest accuracy of 92.16% and provided a 7% advantage over traditional ML methods with hand-crafted features [35].

- **Deep Belief Network (DBN)**

  DBN is a type of deep learning model that is formed by stacking multiple simple unsupervised networks. The hidden layer of the preceding network serves as the input layer for the next network. The building block of a DBN is generally a Restricted Boltzmann Machine, which is an undirected generative energy-based model with a visible input layer, a hidden layer, and connections between the layers. DBN typically has connections between the layers but not between units within each layer. This structure allows for a fast and layer-wise unsupervised training procedure, where contrastive divergence can be applied to every pair of layers in the DBN architecture sequentially. In Zhang et al, a five-layer DBN was trained using acceleration data collected from mobile phones. The study showed an improvement in accuracy ranging from 1% to 32% when compared to traditional ML methods with manually extracted features [44].

- **Recurrent Neural Network (RNN)**

  RNNs are a type of neural network that process input data in a recurrent manner. RNNs have a directed graph structure, exhibit dynamic behaviour and have the ability to model temporal and sequential relationships due to a hidden layer with recurrent connections. In the context of HAR, RNNs are useful for capturing longer context information and longer temporal intervals. However, RNNs can also face challenges such as vanishing or exploding gradient problems while backpropagating gradients. To address these challenges, long short-term memory (LSTM)-based RNNs [32], and Gated Recurrent Units (GRUs) [23] have been introduced. GRUs use a reset and update gate to control the flow of inputs to a cell, while LSTMs are capable of memorizing and modelling long-term dependencies in data. LSTMs have become popular for time-series and textual data analysis and have shown high performance in wearable-based HAR [38].

- **Convolutional Neural Network (CNN)**

  CNN is a type of deep learning model that has become popular for HAR due to its ability to learn robust and informative features from sensor data. A typical CNN architecture consists of convolutional layers, pooling layers, fully connected layers, and an output layer [39].

  The convolution operation in CNNs uses a shared kernel to learn space-invariant features. Each filter in a convolutional layer has a defined receptive field, which allows the network to capture local dependencies. By stacking multiple layers, the neurons in higher layers cover a larger, more global receptive field, which allows the CNN to gather low-level local features and combine them into high-level semantic meanings. Pooling layers are also often used to compress the representation learned by the network and make it more robust to noise. Most CNNs use univariate or multivariate sensor data as input, such as raw or filtered sensor data or the magnitude of 3-axis acceleration. The choice of kernel size and the number of layers in a CNN can vary depending on the specific task and dataset [34].

  The study by Dong et al. presented an improved approach for HAR by combining the use of handcrafted time and frequency domain features with features generated from a CNN. The method was tested on accelerometer and gyroscope data from smartphones and showed improved performance in classifying six different locomotion activities [24].

In comparison, RNNs are good at modelling temporal and sequential relationships, Autoencoders are good at feature extraction and dimensionality reduction, and DBNs are good at modeling complex data distributions. However, CNNs excel in handling sensor data, specifically spatial data and are able to extract meaningful features from it. We trained CNN models for both the RESpeck and Thingy sensory data to provide predictions. CNN has the following advantages compared to other approaches when applied to HAR:

- Local Dependency:
  CNNs are good at capturing local dependencies between the input data, allowing them to learn meaningful features from sensor data. For physical activity recognition, close spatial/acceleration readings may be correlated, and CNNs are designed to take advantage of this by using convolutional layers that apply a shared kernel to the input data, which extracts local features.

- Pooling:
  the pooling layers of CNNs compress the representation being learned and make the model more robust to noise.

3 Methodology

3.1 Wearable Sensors

The RESpeck device is a small, wireless, wearable sensor [25]. It uses a 3-axis accelerometer and a 3-axis gyroscope to measure chest wall rotation and determine the wearer's respiratory rate in a non-invasive manner. It has a motion processing frequency of 25Hz and is placed on the lower left rib cage with tape. This placement on the chest is optimal for both respiratory monitoring and activity tracking as it is close to the body's center of mass and provides clear movement signals for most physical activities [27].

Thingy:52 is a "[c]ompact multi-sensor prototyping platform" [11]. It uses the nRF52832 Bluetooth 5 System-on-a-Chip (SoC) and includes a 3-axis accelerometer, 3-axis gyroscope, and 3-axis magnetometer. It has a 1440 mAh rechargeable lithium-polymer battery which can be charged via a micro-USB port. Configuration of the device can be performed through the accompanying mobile application, available for both iOS and Android. For our purposes, the Thingy was configured with a motion processing frequency of 25Hz to match the sampling rate of the RESpeck device. For data collection, it was placed in the front right-hand pocket of trousers with the RGB LED in the upright position for optimal activity recognition using the IMU sensor [19].

Both the RESpeck and Thingy:52 devices use an accelerometer and a gyroscope to measure movement, which are then transmitted via Bluetooth Low Energy (BLE). The accelerometer measures changes in the direction of gravity and the gyroscope measures the rotational rate in degrees per second of the axis of interest. However, the accelerometer alone struggles to determine lateral orientation or tilt, so it is used in conjunction with the gyroscope to gather more accurate information about the user's movements and infer their position.

3.2 Wireless communication

The communication between the mobile application and the sensors is achieved through BLE, which reduces power consumption and cost while maintaining a similar range as classic Bluetooth (up to 10 meters). BLE is natively supported across many mobile operating systems and is widely used in consumer electronics for its low cost and long battery life [28]. It is also commonly used in Internet of Things (IoT) applications for local, energy-efficient data exchange between smartphones and edge devices. The Android application used in this coursework utilizes the RXAndroidBLE library perform BLE connections [**?**]. There are several factors that can cause communication latency between mobile application and the IMU sensors:

- Connection interval: The connection interval setting determines the frequency at which the device sends data to the receiving device, a longer connection interval means that the device sends data less frequently, which increases the latency of the data transmission.

- Slave latency: Slave latency is a way to conserve energy on the slave (peripheral) device by allowing it to skip sending data to the master device during a connection event in case there is no new data available to send. This is useful in situations where the sensor data is not changing frequently or when the device is in a low power state.

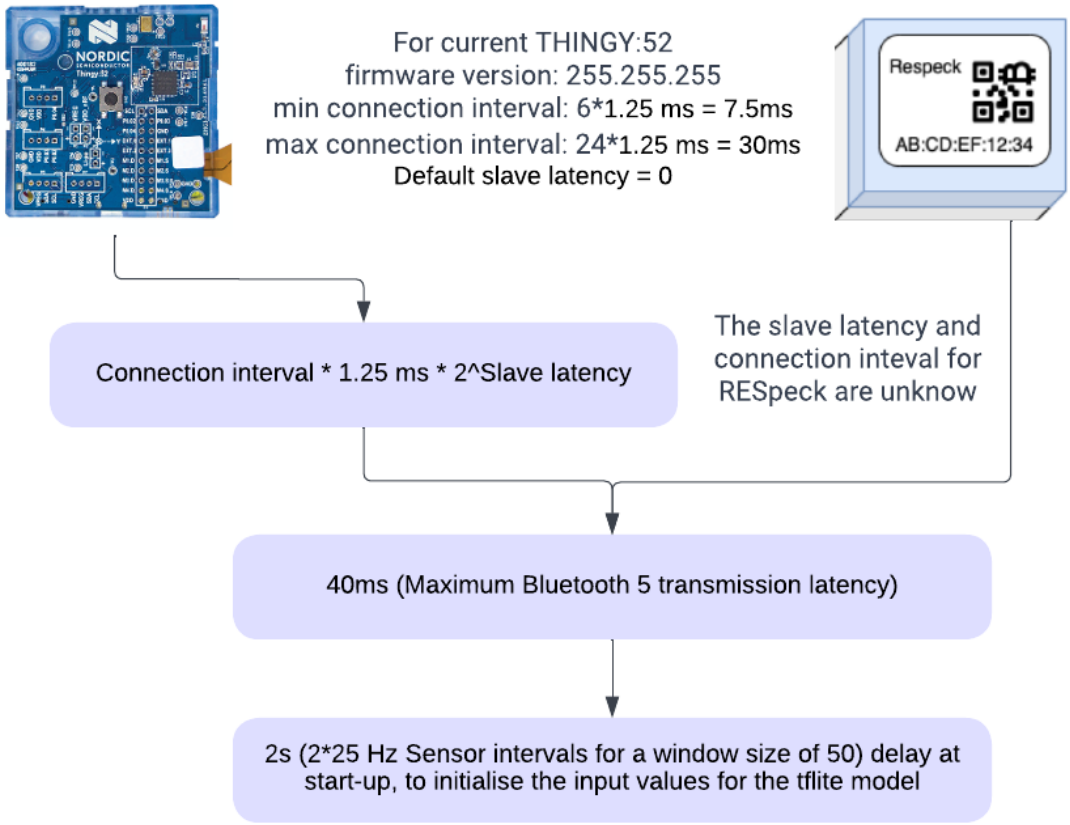


Figure 1: Diagram depicting the communication latency in our HAR system excluding advertising intervals.

Figure 1 illustrates the composition of the communication latency between the wireless sensors and the Android device in our HAR system. We can use both the connection interval and slave latency settings of the sensors to reduce the power consumption and to balance the data transmission latency and energy consumption. Both the connection interval, and slave latency can be configured in the Nordic Thingy:52 application for the Thingy sensor. Unfortunately, there is no way to configure these settings for the RESpeck sensor.

If we choose to increase the connection intervals of the sensors for lower power consumption and less communication latency, then less data will be transmitted per unit of time. Some devices may have a maximum amount of data packets that may be received per connection event. A connection interval that is too long may cause some data to be lost on some devices [18]. Similarly, if the connection interval is set too short, it can lead to an excessive amount of data being transmitted, which can lead to a higher power consumption and increased CPU usage on the phone.

3.3 Machine Learning Hardware

Initially, small machine learning models were trained on a CPU (AMD Ryzen 7 3800X, 3900 Mhz, 8 Cores, 16 Logical Processors) [29] due to its compatibility with the Python's 'sklearn' library. Training models on this CPU resulted in slow training times and was found to be insufficient for creating more sophisticated models.

To remedy this, the larger 3-layered 1-D CNN models were trained on a personal GPU (GeForce RTX 2060 SUPER, 8GB of RAM) [7]. GPUs use parallel computing to break down problems into multiple smaller simultaneous calculations. This allowed training speeds to be increased significantly over the aforementioned CPU. However, due to the limited memory size, batch sizes were limited to less than 500 samples. This batch size limitation had an impact on the models' accuracy. This necessitated a different solution.

Google Colab [8] allowed the use of a significantly more powerful GPU - namely the Tesla K80, which has access to 24GB of RAM. While each user is initially given 10 free 'units' of GPU access - equivalent to roughly 5 hours of training - successful models could not be created in this short time frame. Thus, an additional 300 units were purchased at £9.72 per 100 units. Any additional models mentioned henceforth have been trained using this hardware.

The lack of sufficient hardware was a key limitation in the creation of the machine learning models. There is a positive correlation between the cost of hardware and its quality and specifications [17]. Investing in more Google Colab units was too expensive, thus, the models have been bulk trained and this comes at the sacrifice of accuracy.

3.4 Machine Learning

In order to evaluate the best ML paradigm to follow, a variety of ML models were created. This allowed a quantitative evaluation of these models. To limit the number of changing variables between each model, the dataset used was consistent across the different models. Throughout this process, three different classification methods were experimented with.

Data from the years between 2018-2021 was provided by the course organisers. This data required pre-processing in order for it to conform to the data collected in 2022. "Falling" activities were excluded from the 2021 data. Further, only the "Chest" configuration for RESpeck sensor data from 2020-2018 was included to ensure accurate data. A window size of 50 data points with a step size of 25 was used. We applied curriculum learning when training the models by gradually increasing the amount of training data available to the model. This method allows the model to adapt more easily to the training data. All real time data coming from the connected sensors is passed to CNN models which will take in sensor data individually and predict all 14 activities only if the combined confidence is >95%.

Two large CNN [40] models were trained using curriculum learning [21] for both the RESpeck and Thingy. These models work in tandem to provide a prediction. These are trained to predict all 14 activities and take in sensor data individually. This approach was inspired by similar works in fields such as Automated Speech Recognition and Natural Language Processing [22]. This method was repeated to create a single, six-layer CNN model that accepts data from both the Thingy and RESpeck.

Finally, a tree-based model was created [14]. This model uses a number of smaller CNNs, each trained on a subset of activities. A 'root' model decides what overarching category the data suggests the user is performing, such as lying down, or sitting. Using the output of the root model, the data is then passed to a specialised model trained on that category only which outputs the final classification. This approach benefits from faster training times due to less data being required per model.

3.4.1 Datasets and Pre-processing

The datasets provided were delineated by year for each year between 2018 and 2022. Each year differed in the amount of data provided. This resulted in a large disparity in the data. From 2018-2020 the data consisted of only the RESpeck sensor in a variety of positions, for example. The amount of usable data between the two years can be

seen in Table 1. These numbers are representative of the valid number of samples we were able to extract and use within our models. They exclude outlying students that did not format data correctly.

Year	Thingy Data Samples	RESpeck Data Samples
2022	74	74
2021	41	41

Table 1: depicting the number of valid user SIDs that were used per year.

One of the issues we found when attempting to use the 2018-2020 data was that not all years were equipped with gyroscopes. This would have necessitated in a large-scale reformatting of the 2018-2019 data in order for it to be usable. However, due to time constraints and possible issues due to incorrectly formatted data, we decided against this.

In our initial exploration into the large CNN models we decided to stick to using only the 2022 dataset. However, future models use both 2022-2021 for Thingy-only models and all available data for RESpeck only.

In order to get this data into a valid format for ML we first need to scrape the data from the individual folders and separate it into its activities and sensors. We do this by first checking for all folders starting with the "S" prefix standard to university SIDs and iteratively extracting file paths which contain occurrences of the activity and senor we need. This leaves us with two hash maps for each sensor where class names such as "Climbing Stairs" are mapped to an array of file paths containing valid data for that activity. "Falling" activities in the 2021 data were excluded as they are no longer included in the course.

Similarly, we needed to make sure that the correct data was extracted for the RESpeck sensor using data from 2018-2020. Only the "Chest" configuration was applicable to the current course. Other configurations were excluded. The data is then stored in the '.csv' file type. This data is packaged into a single dataframe containing all activity data per sensor and randomized to remove any artificially imposed structure created during extraction. To do this we use the 'shuffle' function from 'sklearn'. This left approximately 750-770 timestamps per data frame for a single activity recording.

Each dataframe was then formatted into 'windows'. A window is a grouping of datapoints in time that together give more context than if they were considered separately. We decided to use 50 data points per window. By selecting a step size of 25 - providing an 50% overlap between frames - roughly doubled the number of datapoints that could be used in training. This is an approach commonly used within ML [42].

Year	Thingy Data Samples Before Windowing	RESpeck Data Samples Before Windowing	Thingy Data Samples After Windowing	RESpeck Data Samples After Windowing	Thingy Total Unique Window Ids	Respec Thingy Total Unique Window Ids
2022	779072	779072	1501600	1501600	30031	30168
2021	420168	420168	821900	821900	16652	16767

Table 2: Data samples before and after windowing procedure.

Features are extracted using "tsfresh" [15]. This transforms the data into one multidimensional datapoint displaying things like the "medium", "mean" and "variance" of the gyroscope and accelerometer inside the sensor. Hence, for each window ID, we display that window as a single vector of processed sensor readings. This is done using the "MinimalFCParameters()" extraction method which is a faster variant of the default "ComprehensiveFCParameters()" extraction process.

Lastly we need a mapping from activity type to an integer such that our machine learning model can correctly predict one of our possible activities. To do this we use the paradigm depicted here [Figure 30].

Now that we have the data we split it into our training and test sets using the "sklearn" "train_test_split" function performing an 80/20 split with the majority being training. For future use and continued training over multiple days we save these training and test sets such that training can be done in iterations.

This split is later used within our results section to verify the models have been trained correctly and are not highly specialized to the training data [Section 4. We additionally use the “Unseen” dataset which consists of “RESpeck/Thingy_recordings_clean” files provided to us as a reference for data structure. This acts as an additional sanity check on the models to reflect what results will look like on offline activity recording.

3.4.2 Model Architecture

Our first model choice was a 3-layered 1D CNN [Figure 2] with 64 filters, a kernel size of 3 and a ‘relu’ activation function. Additionally, a dropout value of 0.2 was explored as - given the high variability in the data - it is reasonable to assume it may improve results. The batch sizes were increased periodically alongside the number of epochs to allow the model to grow and overcome any local maximums it may hit.

```
filters = 64
kernel_size = 3
n_features = 6
activation='relu'
n_classes = 14
dropout = 0.2

window_size = 50 # 50 datapoints for the window size, which, at 25Hz, means 2 seconds
step_size = 25 # this is 50% overlap
columns_of_interest = ['accel_x', 'accel_y', 'accel_z', 'gyro_x', 'gyro_y', 'gyro_z']

model = Sequential()

model.add(Conv1D(filters=filters, kernel_size=kernel_size, activation='linear',
                 input_shape=(window_size, n_features)))
model.add(BatchNormalization())
model.add(Activation(activation))
model.add(Dropout(dropout))

model.add(Conv1D(filters=filters, kernel_size=kernel_size, activation='linear'))
model.add(BatchNormalization())
model.add(Activation(activation))
model.add(Dropout(dropout))


model.add(Conv1D(filters=filters, kernel_size=kernel_size, activation='linear'))
model.add(BatchNormalization())
model.add(Activation(activation))

model.add(Flatten())
model.add(Dense(100, activation='relu'))
model.add(Dropout(dropout))
model.add(Dense(n_classes, activation='softmax'))
```

Figure 2: The simple 3-layered 1D CNN.

A similar process was performed to create the tree model. In this case, the number of layers and kernel size was reduced to 2. This decision was made as there are less possible activities for a given model to predict. Thus, it is possible to increase the training speed significantly without a similarly significant impact to accuracy.

```
filters = 64
kernel_size = 2
n_features = 6
activation='relu'
n_classes = 14
dropout = 0.2

window_size = 50 # 50 datapoints for the window size, which, at 25Hz, means 2 seconds
step_size = 25 # this is 50% overlap
columns_of_interest = ['accel_x', 'accel_y', 'accel_z', 'gyro_x', 'gyro_y', 'gyro_z']

model = Sequential()

model.add(Conv1D(filters=filters, kernel_size=kernel_size, activation='linear',
                 input_shape=(window_size, n_features)))
model.add(BatchNormalization())
model.add(Activation(activation))
model.add(Dropout(dropout))

model.add(Conv1D(filters=filters, kernel_size=kernel_size, activation='linear'))
model.add(BatchNormalization())
model.add(Activation(activation))
model.add(Dropout(dropout))

model.add(Flatten())
model.add(Dense(100, activation='relu'))
model.add(Dropout(dropout))
model.add(Dense(n_classes, activation='softmax'))
```

Figure 3: Tree architecture.

The final model consists of a 6 layered 1D CNN [Figure 3] with an increased dropout to 0.4, increased filters to 128, and a larger kernel size of 6. Theses changes were implemented in order to increase accuracy.

Each model is evaluated in the Results section of this paper.

```
filters = 128
kernel_size = 6
n_features = 6
activation='relu'
n_classes = 14
dropout = 0.4

window_size = 50 # 50 datapoints for the window size, which, at 25Hz, means 2 seconds
step_size = 25 # this is 50% overlap
columns_of_interest = ['accel_x', 'accel_y', 'accel_z', 'gyro_x', 'gyro_y', 'gyro_z']

model = Sequential()

model.add(Conv1D(filters=filters, kernel_size=kernel_size, activation='linear',
                 input_shape=(window_size, n_features)))
model.add(BatchNormalization())
model.add(Activation(activation))
model.add(Dropout(dropout))

model.add(Conv1D(filters=filters, kernel_size=kernel_size, activation='linear'))
model.add(BatchNormalization())
model.add(Activation(activation))
model.add(Dropout(dropout))


model.add(Conv1D(filters=filters, kernel_size=kernel_size, activation='linear'))
model.add(BatchNormalization())
model.add(Activation(activation))

model.add(Conv1D(filters=filters, kernel_size=kernel_size, activation='linear'))
model.add(BatchNormalization())
model.add(Activation(activation))

model.add(Conv1D(filters=filters, kernel_size=kernel_size, activation='linear'))
model.add(BatchNormalization())
model.add(Activation(activation))

model.add(Flatten())
model.add(Dense(100, activation='relu'))
model.add(Dropout(dropout))
model.add(Dense(n_classes, activation='softmax'))
```

Figure 4: Larger CNN trained on more data

3.4.3 Curriculum Learning strategy

Curriculum learning is a strategy commonly used within ML literature to increase accuracy by increasing the amount of training data available to the model. This is done by providing only 20% of the total data over the first 15 runs of the model, incrementing by 20% until the full 100% is reached. This allows the model to adapt to the training data more easily. This process has been shown to increase accuracy over other methods [21].

3.4.4 Tree Structured Models

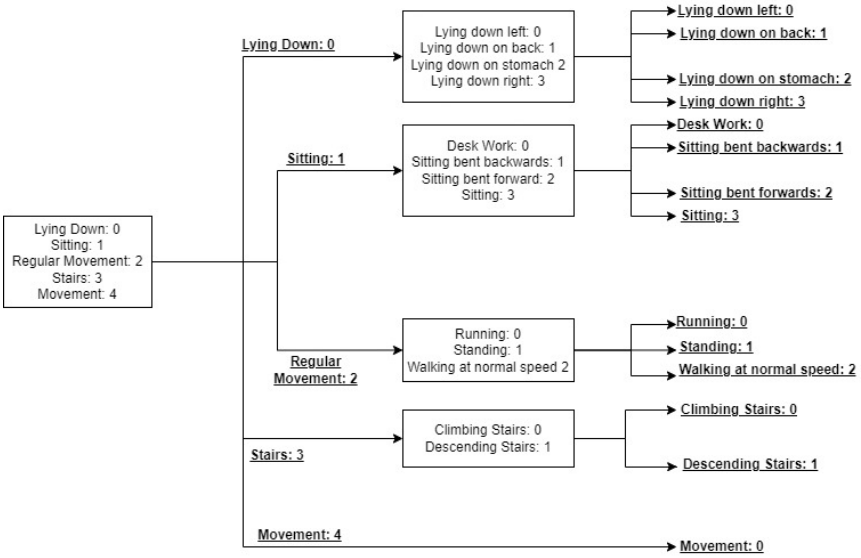


Figure 5: Tree architecture.

During the creation of the various models it was found that a key issue with having a single model trained to predict all 14 activities is that high variance activities such as “Movement” or “Desk Work” dominate the prediction space. To alleviate this effect, a tree structure, demonstrated in Figure 5 was created. The root layer is a 5 class model which delineates between 5 broad categories of activities, namely lying down, sitting, stair movement, regular movement, and the general ‘movement’ activity. this classification is then used to decide which further model to use to determine the specific activity being performed, such as “Lying down left” compared to “Lying down right”.

3.5 Mobile Application

One of the requirements of the system is a supporting mobile application. This application will be primarily used to allow the user to connect to and use the sensors; it will facilitate the processing of the sensor’s data into a real-time classification of the user’s current activity which must be presented to the user. This necessitates the design of a user interface.

There are many methods defined in literature that concern how to successfully design a user-orientated system such as this. For instance, Ball’s ‘Double Diamond’ [4] or Martin and Harrington’s Universal Method of Design [37]. For this application, we decided upon using Nielsen’s 6 stage ‘Design Thinking 101’ process [2]. This decision was taken because - as Nielsen notes - it is a flexible process that ‘is meant to be iterative and cyclical’ [2]. It is possible under this system to return to and revise previous stages in the design process. The cyclic nature of this process is advantageous as we had access to a peer-review of a prototype application. The feedback of this peer review would then be used to inform a potential redesign of the application if the said feedback recommended it. The 6 stages of this design process are as follows:

1. Empathise - research is conducted into what users will need from the application.
2. Define - problems that must be solved by the application are identified and design aims are set.
3. Ideate - a range of ideas are created that aim to solve these problems.

4. Prototype - a prototype application is built that incorporates the solutions realised in the ‘ideate’ phase.
5. Test - feedback from users is obtained from the prototype.
6. Implement - the final product is materialised.

3.5.1 Empathise

Nielsen notes that the empathise process should involve observation of the intended users. This is done with the intention of giving the designers a greater sense of what is needed by the user from a successful design. Given the nature of the course this paper is created for, we - the designers - were to use the skeleton application ourselves in the earlier stages of said course. This allowed us to understand what a user will want out of the application as they use it as we were users ourselves. Thus, the empathise stage was completed through our own use of the application.

3.5.2 Define

The design process - as Nielsen notes - is about defining “where your users’ problems exist”. Through our use of the skeleton mobile application we individually identified a number of key areas that could be improved upon. We then discussed our individual thoughts as a group to identify recurring problems in our observations.

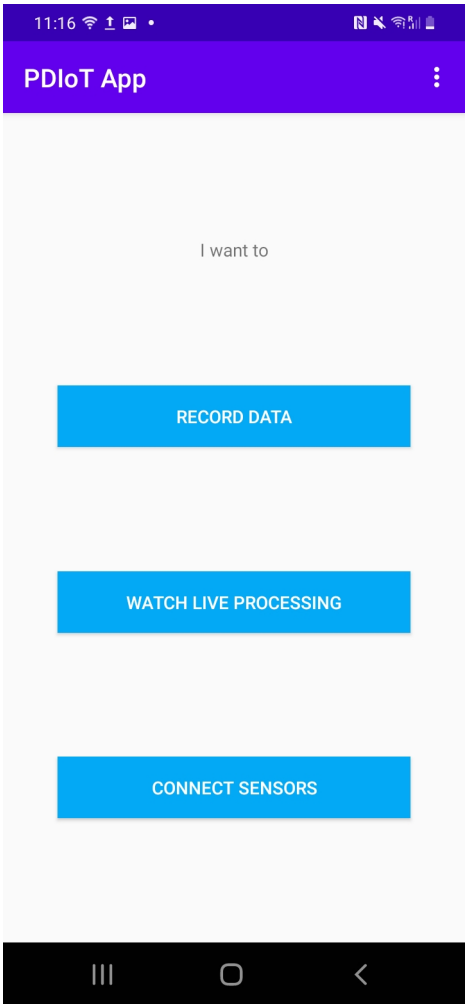


Figure 6: The landing page of the skeleton mobile application.

In Figure 9, we see the main (landing) page in the skeleton mobile application. One aspect we all noticed was the use of strong, highly saturated, ‘neon’ colours. Cousins writes that strong colours such as these should be avoided as they are ‘incredibly hard on the eyes’ [3]. Further, the Web Content Accessibility Guidelines (WCAG) [31] recommend a colour contrast ratio of at least 4.5:1 for background colour to text colour. In Figure 9, the

buttons have a contrast ratio of only 2.57:1, while the header has a ratio of 2.92:1. Thus, we sought to improve upon the colour scheme of the design.

Further, we each identified that the application uses the landing page as an intermediate step in navigating between any two sub-pages. For instance, one use case of the application is to connect the sensors and watch the output. To do this, a user would need to select 'connect sensors', come back to the landing page, and then select 'watch live processing'.

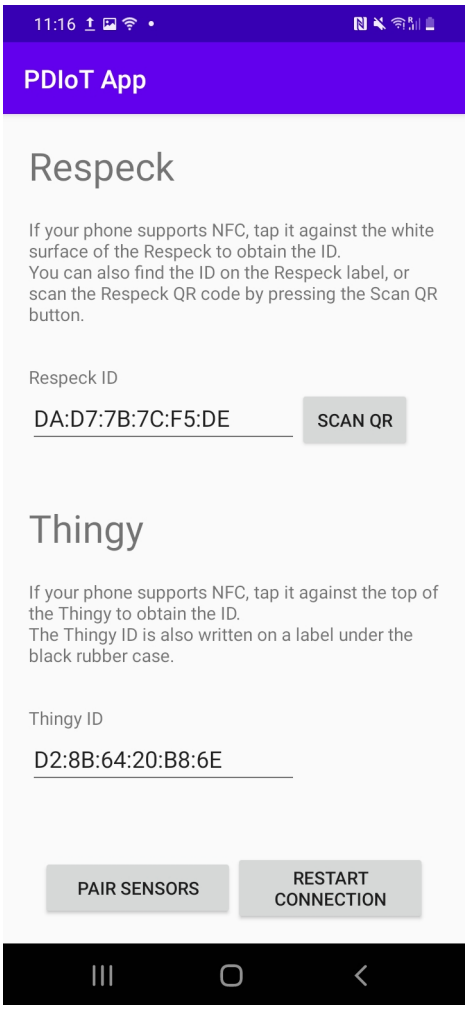


Figure 7: The connection sub-page of the skeleton mobile application.

We also noticed a number of violations of Nielsen's 10 usability heuristics for user interface design [26]. Firstly, following connecting the sensors on the 'connect sensors' sub-page, the user must navigate to the live processing or record data sub-pages to know if the connection process has been successful. We felt that this violated the 'Visibility of System Status' heuristic. Nielsen writes that designers should '[c]ommunicate clearly to users what the system's state is'. This heuristic is also violated in the header title 'PDIoT App'. This title does not change regardless of what page the user is viewing. The user is unable to visualise what page they are on without remembering that the page's content is being displayed because of the button they clicked on the landing page. Finally, the connection sub-page, as shown in Figure 7, does not make it abundantly clear to the user what a 'RESpeck' or 'Thingy' is, and may mistake them for each other. This violates the 'Match Between System and the Real World' heuristic. Nielsen warns that designers should not assume our understanding of words or concepts will match that of users.

In summary, we felt the original design created problems for users in the following ways:

- The use of colour is detrimental to accessibility, nor follows aesthetic design principles.
- The navigation throughout the application is not optimal.
- The application does not conform to the 'Visibility of System Status' heuristic.
- The application does not conform to the 'Match Between System and the Real World' heuristic.

3.5.3 Ideate

This phase involves coming up with a range of ideas that solve each of the problems found in the ‘define’ phase. The first problem we identified that we needed to address was that of colour. To solve this, we decided that we should create an accessible and pleasing colour palette for the redesigned app. Marcus writes that a successful design should use between 3-7 colours [36], thus, we kept to this restriction. Ultimately, we decided upon using a monochromatic palette for background colours, and appropriately contrasting colours for text. Ross writes that monochromatic palettes look clean and professional; this was a motivating factor in this decision [16].

Figure 8: Colour palette goes here.

Additionally, we believed navigation within the application could be simplified by making use of the ‘burger menu’ design pattern [9]. This design pattern allows for the navigation to any sub-page from any sub-page, enhancing navigation for the user.

Finally, in order to conform to the violated heuristics noted previously, we decided to include images of the RESpeck and Thingy within the application’s design. This allows the user to quickly identify each sensor. Further, we looked to add a ’connected/disconnected’ UI element on the main landing page to signify whether or not the application is connected to the sensor correctly. These two fixes allow the application to follow Nielsen’s heuristics more closely.

3.5.4 Prototype

With solutions to our problems identified, we began developing the design. A wireframe mock-up was created using Desktop Publishing (DTP) software, as shown in Figure 9.

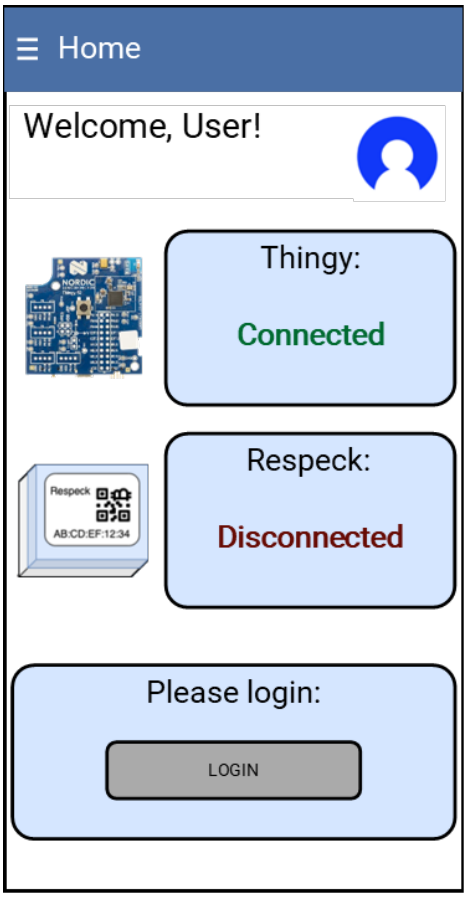


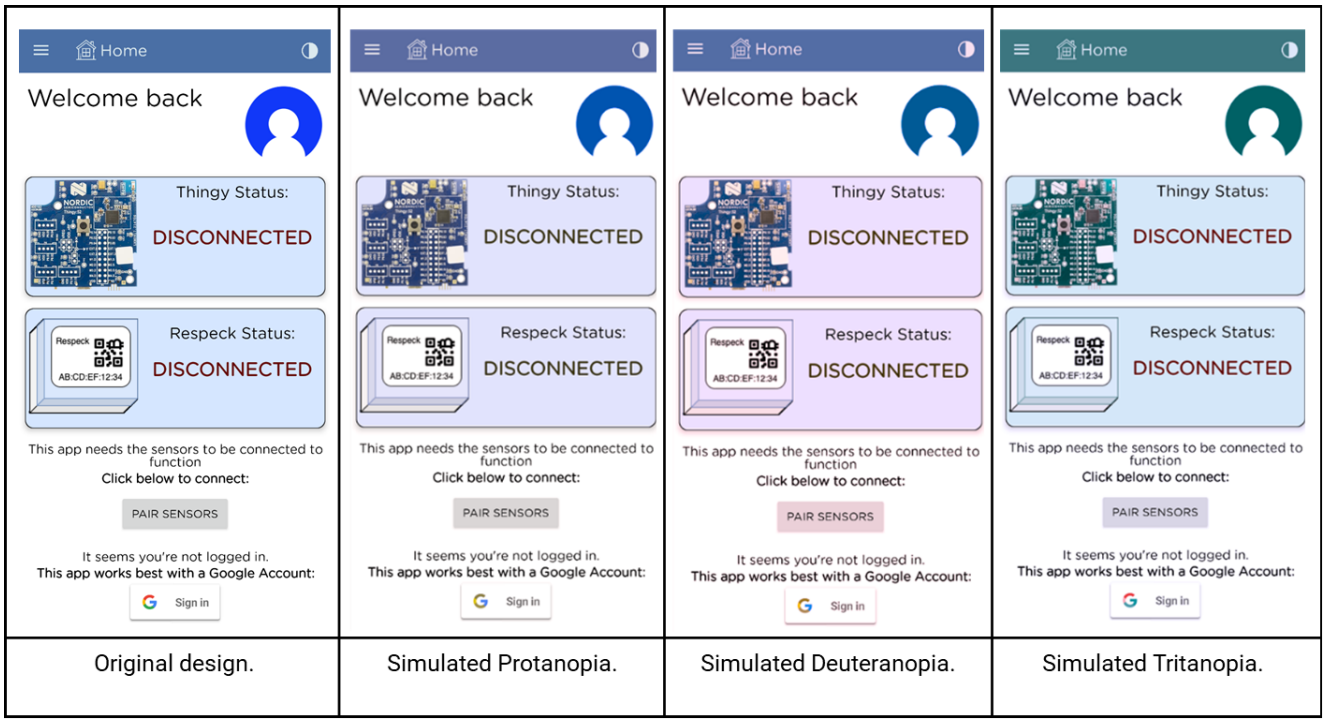
Figure 9: Design pages go here.

Using the DTP wireframe as a guide, the design was transcribed into the Android Studio integrated development environment [5]. This necessitated some design changes in order to conform to the technical limitations imposed by the programming and markup languages used. At this point, the additional technical features were added to the application, as specified in section X of this report.

3.5.5 Test

The testing stage contained the peer review, but also testing within the team. Firstly, the application was downloaded to each of the team's mobile phones and tested with the sensors to verify it worked as intended across our devices. No issues were identified at this stage.

Further, images of the design were ran through the Coblis tool [1]. This tool simulates the various colourblindness conditions. This allowed us to verify that even those with colourblindness are able to use our application. Figure 10 displays the main landing page across several types of colourblindness. As we had prepared to accommodate colourblindness conditions through adherence to the WCAG and monochromatic colour palette in the earlier design stages, we can see that each condition does not affect the usability of the application.



Original design.	Simulated Protanopia.	Simulated Deuteranopia.	Simulated Tritanopia.

Figure 10: The application's design shown alongside versions of the design filtered with simulated colourblindness conditions.

Finally, the application was sent for peer-review. The peer-review gave the design a rating of '10/10'. The only suggested improvement was to increase the font size of the classification on the classification pages. This improvement was incorporated into the design.

3.5.6 Implement

Given the lack of actionable feedback during the testing stage, the implementation stage did not yield any significant changes in the design.

3.6 System Implementation

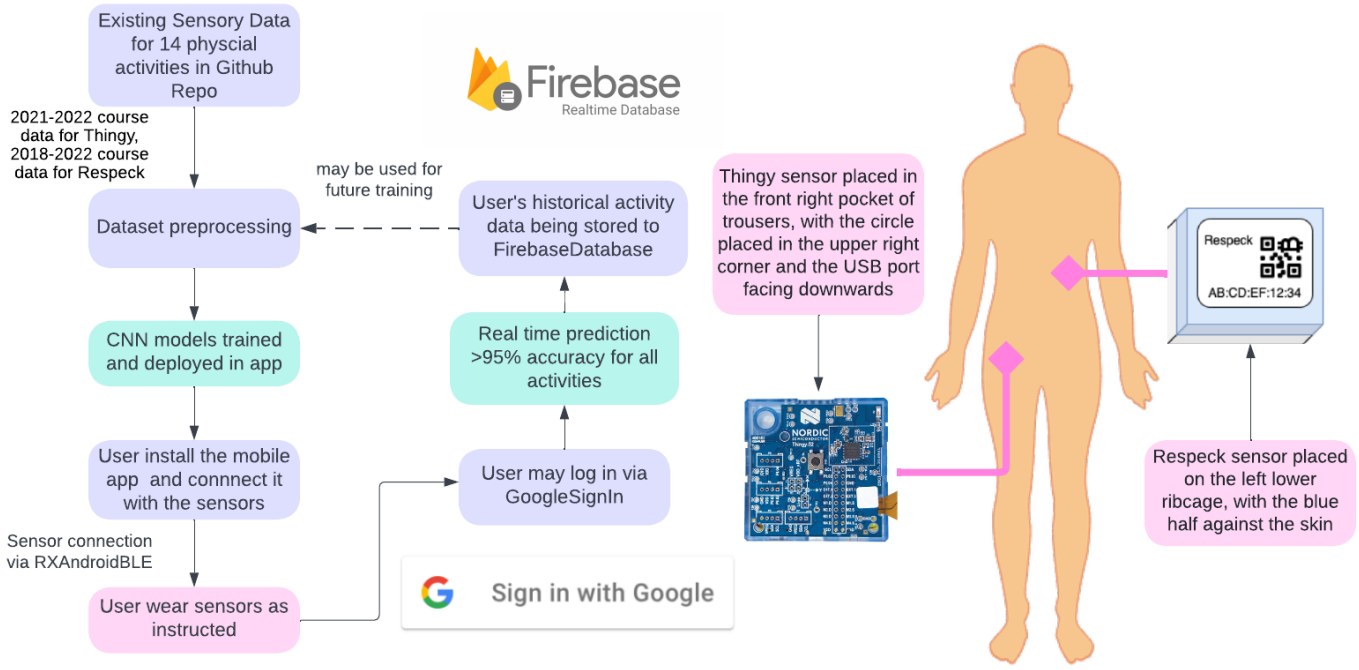


Figure 11: Workflow of the HAR end-to-end system.

The end-to-end system workflow is illustrated in Figure 11. The process begins with an active user wearing both the RESpeck and the Thingy:52 sensors while performing one of the fourteen physical activities classified by the application. The user must first adjust the sensor sampling frequency to 25Hz and establish a connection between the sensors and the Android application via BLE. There are multiple options for connecting the sensors: NFC pairing, scanning the QR code of the sensors, or manually inputting the MAC address of the sensors into the app. The sensor only needs to be paired once, the application will remember the MAC address when it is restarted. The data packets (gyroscope and accelerometer readings) from the sensors are sent over BLE to the connected smartphone. This data is then passed to the ML model for classification.

The application allows users to make an account through Google Auth, an extension of the OAuth 2.0 standard [12]. This gives each user a unique ID within the application. Through this, we can store data associated with that user. This allows us to collate a historic database of previous activities performed by that user. The application uses Google Firebase for this data collection [6]. Google Firebase allows us to store a 'Realtime Database' of information. Whenever a classification is made by a ML model within the application, if the user is signed in through Google Auth, the application will send the classification to the database under that ID. The data within the database for each user is stored in a separate document for each day. Real-time databases are optimal for use-cases such as this, for the application is constantly updating the database when in use.

To maintain compliance with data protection guidelines, the user is able to - at any point - delete their stored data. This process asks the database to delete all information stored under that ID. The user is able to re-register, but their previous data is permanently deleted. Figure 12 shows the activity within the application where the user is able to do this.

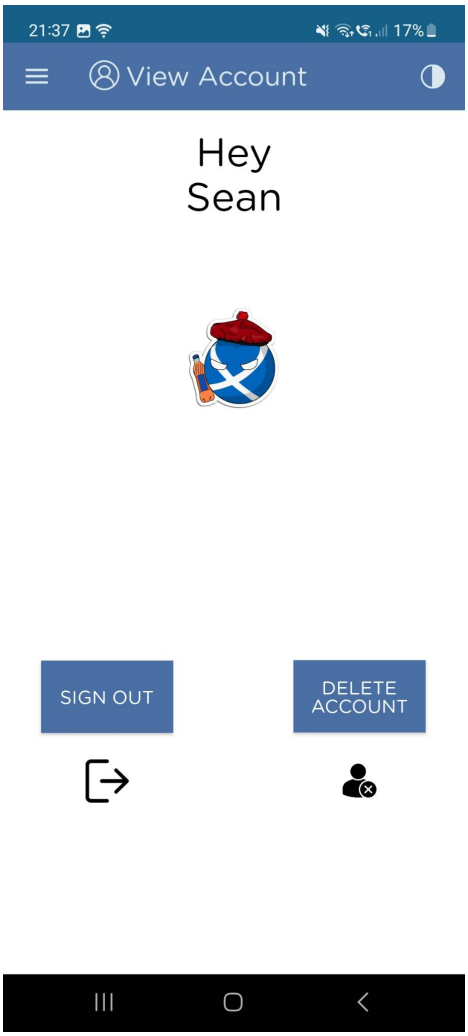


Figure 12: An example of the ‘View Account’ screen within the application.

The user is able to visualise their previous activity through the use of the ‘MPAndroidChart’ library, created by Philipp Jahoda [10]. This is a publicly available library that allows for the charting of data within Android applications. To prevent cluttering the chart, all activities that account for less than 10% of the activities performed on a given day are grouped together under the ‘others’ category. Figure 13 displays a the data of a user who has been inactive for most of the day. A DatePicker allows the user to select the day they want to view. A message is shown if the user selects a day they do not have data for. The DatePicker class is part of the standard ‘android.widget’ library. It allows a value change listener to be assosciated with the DatePicker, allowing the UI to react to the user changing the date selected.

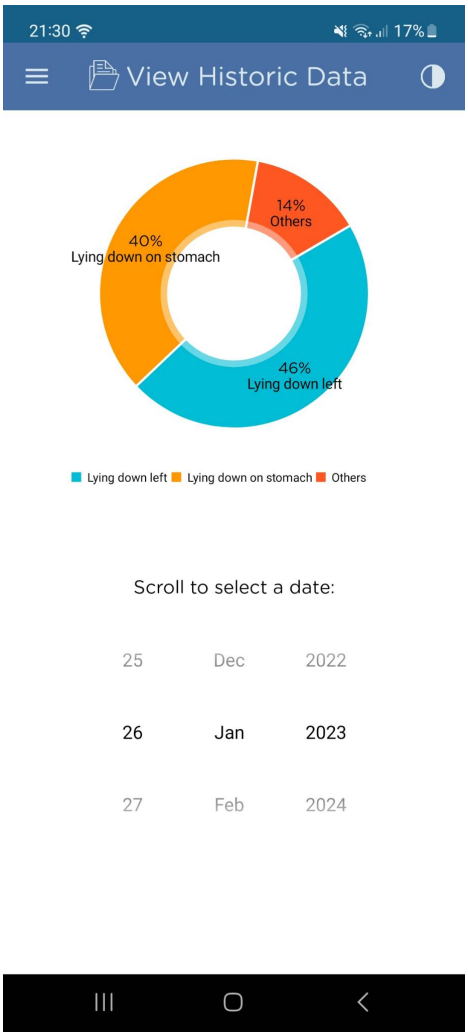


Figure 13: An example of the 'View Historic Data' screen within the application.

3.7 Software Organisation

In total, 10 additional classes were added to the application's code-base. 5 of these classes are additional activities that a user can navigate to within the app. All of these 5 new classes inherit from the abstract class 'ActivitySuperclass', which holds the logic for switching between the activities with the navigational drawer. The final 4 classes are utility classes, meaning that they are used within other classes for their functions.

Figure 14 shows the organisation of the classes:

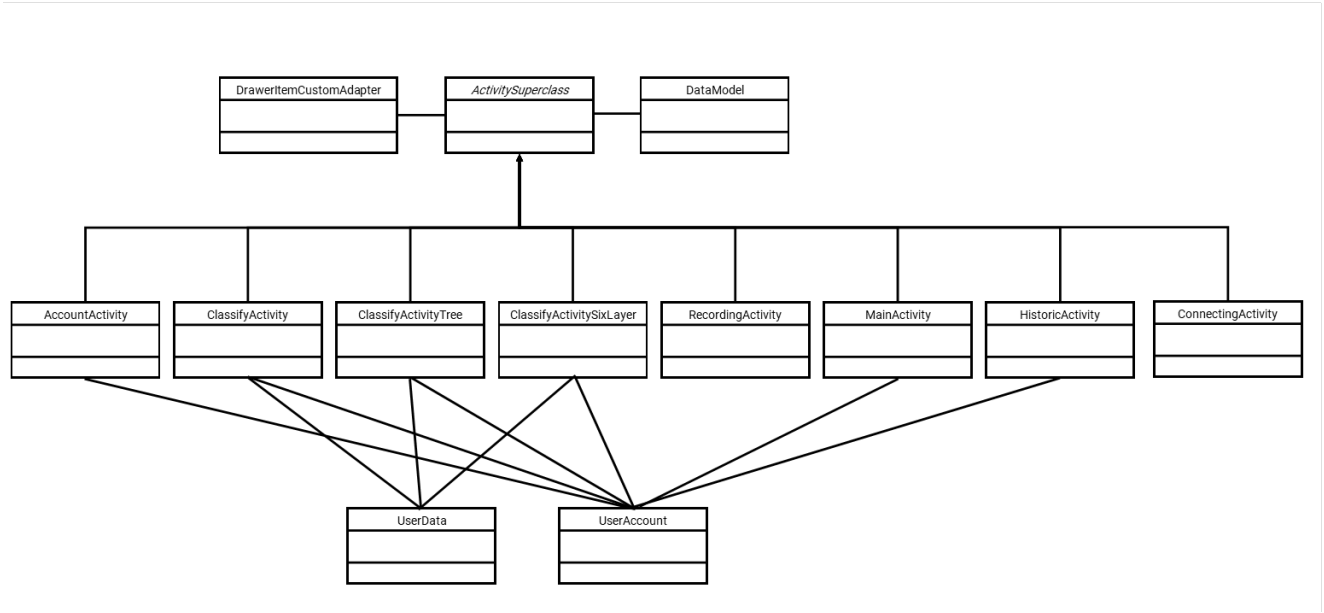


Figure 14: Simplified UML diagram of the relevant classses within the application.

Figure 15 demonstrates the intended workflow of the user within the application in relation to the classes used within the application, along with how the application utilises its various sources of information.

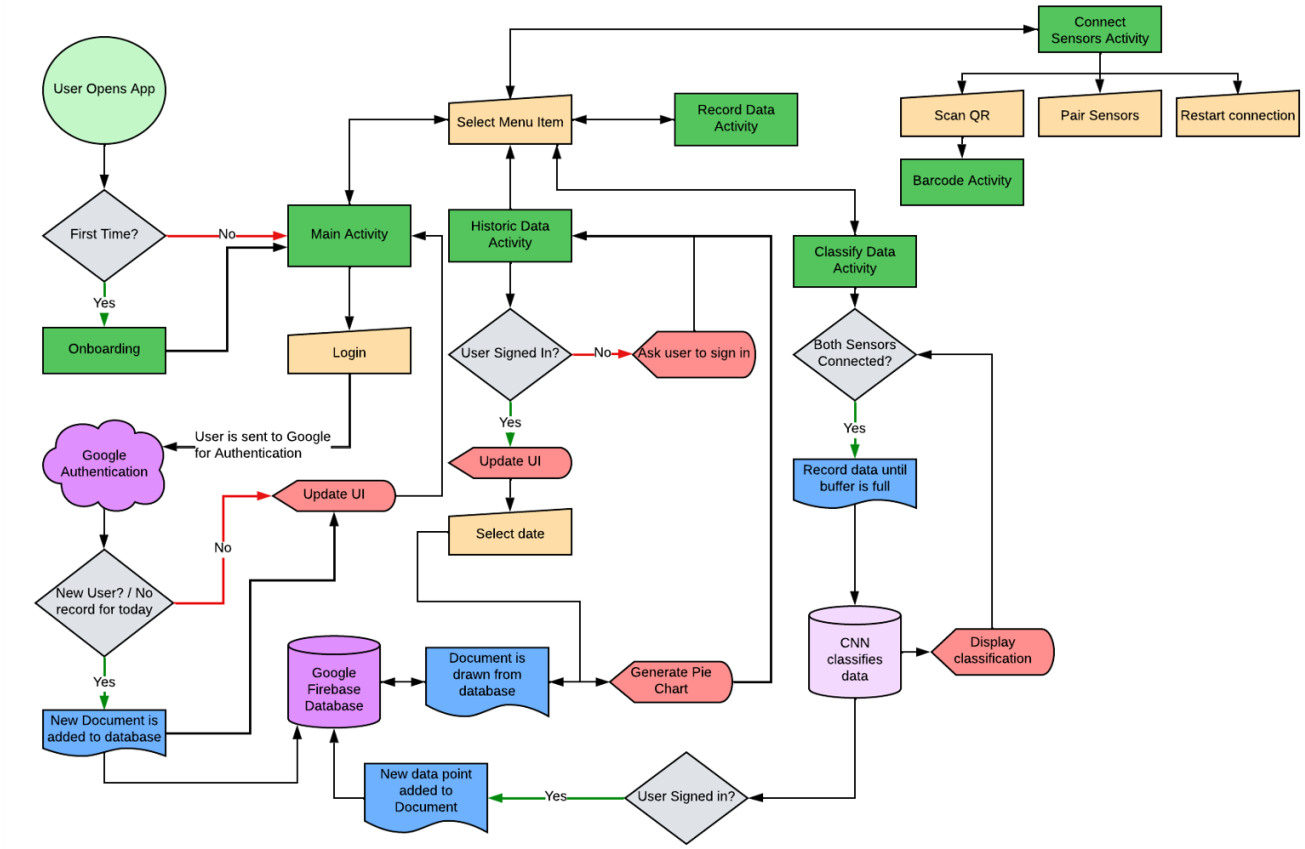


Figure 15: A flowchart showing the indended workflow of a user within the application.

Classifier	Sensor	Accuracy on Training Data (%)	Accuracy on Test Data (%)	Accuracy on Unseen Data (%)	Number of Epochs Trained
6-Layer CNN	Thingy & RESpeck	84	82	82	396210
3-Layer CNN	Thingy	84	72	65	66700
3-Layer CNN	RESpeck	92	88	69	227910
2-Layer Root Node	Thingy	92	90	91	19530
2-Layer Root Node	RESpeck	97	93	96	8200
2-Layer Sitting	Thingy	100	86	89	4350
2-Layer Sitting	RESpeck	100	93	87	1900
2-Layer Lying down	Thingy	100	99	98	2530
2-Layer Lying down	RESpeck	100	99	96	1900
2-Layer Regular movement	Thingy	99	99	99	60
2-Layer Regular movement	RESpeck	99	99	99	60
2-Layer Stairs	Thingy	100	93	57	60
2-Layer Regular movement	RESpeck	98	98	97	60

Table 3: Accuracy of the machine learning models on each of the data sets along with the total trained epochs.

4 Results

4.1 Model Accuracy Comparisons

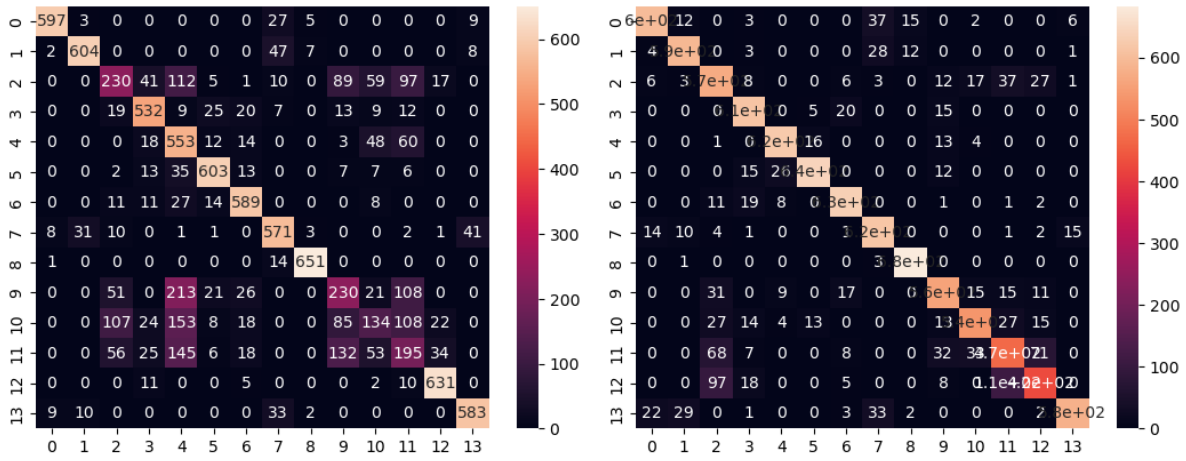
To evaluate the accuracy of each of the models previously discussed, each model was used to classify the data within the test and unseen data sets, as discussed in the section [Section 3.4.1]. All models discussed - unless stated otherwise, – have been trained to over 90% accuracy on their training sets. It is important to note that the 'overfitting' effect is mitigated by both the highly variable nature of the data produced in HAR and the low dataset size [**?**].

4.1.1 3-Layer CNN

Initially, two separate models - one for each sensor - were trained. Their outputs were combined at the prediction stage. These models were trained on a personal CPU and GPU using the 2022 dataset [Section 3.3]. Unfortunately, these models were incorrectly trained due to a class being mislabeled, leading to a significant drop in accuracy. Thus, new models were trained using the same methods. These new models did not have as much time to train as intended because of this error. As such, the Thingy model only achieved a training accuracy of 84% and the RESpeck achieved 92%, as shown in Table 3.

Within the mobile application, each model creates a prediction on the data it is provided. The application then displays the output of the more confident model. This worked in practice, however, when testing the mobile application it quickly became apparent that certain activities were more difficult to predict using only one sensor for each model. This could be alleviated by creating a model that accepts both sensors data as input at once. However, this was not completed due to the aforementioned limitations in hardware and time.

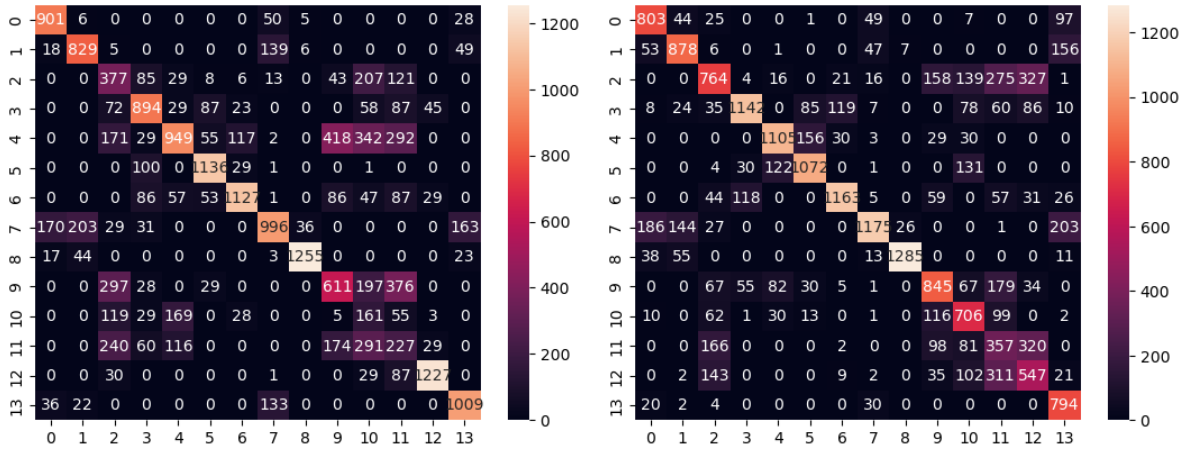
Looking at the test data examples for both sensors, shown in Figure 16, we see a strong diagonal present within the RESpeck sensor, indicating a high offline accuracy. We also observe a fall in said accuracy when approaching more difficult tasks such as climbing stairs and various forms of sitting. Similarly, a strong drop is observed in the Thingy classifications. Various sitting varieties, as well as Movement, are problematic for the model. This could be due to the precise nature of the Thingy sensor, combined with the lack of training time. On the unseen data set, this accuracy falls below 70%. Possible improvements include longer training times with better hardware, using a combination of the 2022 and 2021 data, as well as normalising the data.



(a) Thingy 3-Layered CNN on test data

(b) RESpeck 3-Layered CNN on test data

Figure 16: Test data on 3-Layered CNN for both sensors



(a) Thingy 3-Layered CNN on unseen data

(b) RESpeck 3-Layered CNN on unseen data

Figure 17: Unseen data on 3-Layered CNN for both sensors

4.1.2 Tree Models

Classifier	Number of Total Epochs Trained	Total Offline Accuracy (%)
6-Layer CNN	396210	82
3-Layer CNN	294610	67
Tree Structure	38650	90

Table 4: Total epochs trained for each model combining all sub-models into one with total offline accuracy.

As we see in Table 4, a greater classification accuracy was achieved with a much lower total training time compared to the previous models. Compared to our largest 6-Layered CNN this structure took one tenth of the time to train a model of similar accuracy. This saved a significant amount of time and provided better offline accuracy than either of the large CNN models at a fraction of the computational costs.

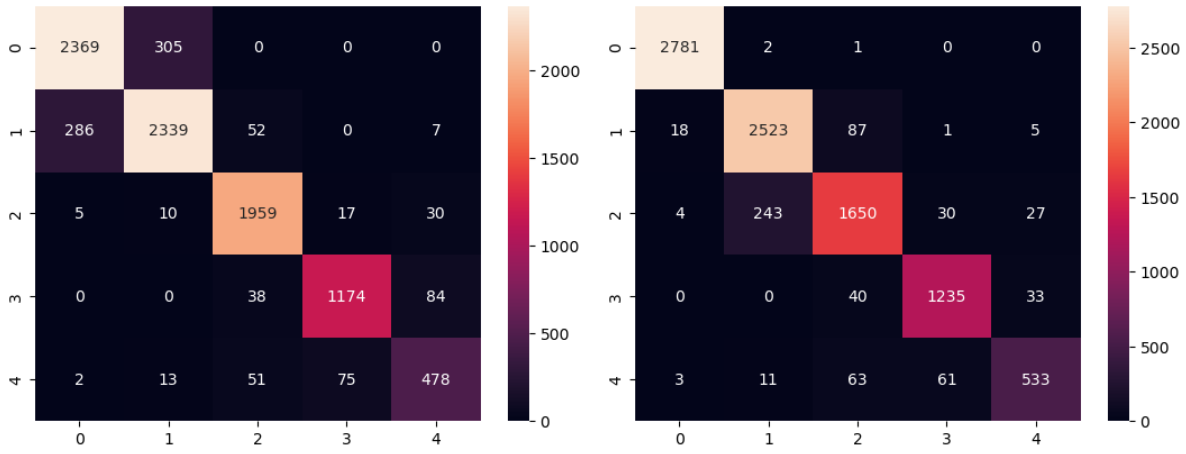
4.1.2.1 Root Models The root models explained in the previous section [4.1.2] consisted of only 2-layer CNN with reduced kernel size used to predict 5 base classes, shown in Figure 5.

In Figure 18 we see a strong diagonal across both the Thingy and the RESpeck models indicating high accuracy. We also see that the RESpeck seems to have a much greater accuracy than the Thingy, at 96% and 91% respectively. This could be down to the position of the sensors; the covariance matrix shows us that the problematic classes for the Thingy are "Lying down" and "Sitting". This logically makes sense since the relative position of the sensor between these two activities is not large hence given a small kernel size these can be easily confused. Additionally, the RESpeck sensor has a similar issue with the "Sitting" class. Experimentation should be done in the future to alleviate this issue.

Looking at the unseen data in Figure 19 we see the continued trend that the RESpeck outperforms the Thingy sensor at most activities. This difference is much larger than the test data at 96% for RESpeck and only 89% for the Thingy. Inspecting this further we see that both sensors struggle with the "Movement" class over-predicting across both instances. Hence to fix this issue we implement a flat bias against movement of -10% across all predictions which has in practice improved our accuracy offline, shown in Table 4.

We also see that within the individual training times shown in Table 3, the Thingy model was trained significantly longer for a lower overall result on all data. This again we assume is due to the nature and variable position of the sensor across students as well as the precision of the sensor.

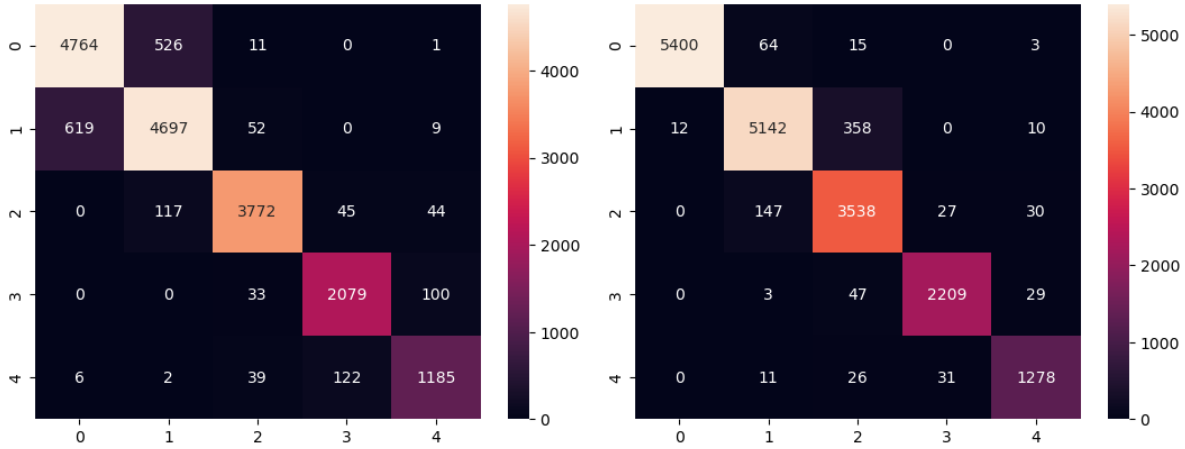
Possible improvements to this model include increasing the kernel size as well as fine-tuning and normalizing the data used.



(a) Thingy 2-Layered root model on test data

(b) RESpeck 2-Layered root model on test data

Figure 18: Test data on 2-Layered root model for both sensors



(a) Thingy 2-Layered root model on unseen data (b) RESpeck 2-Layered root model on unseen data

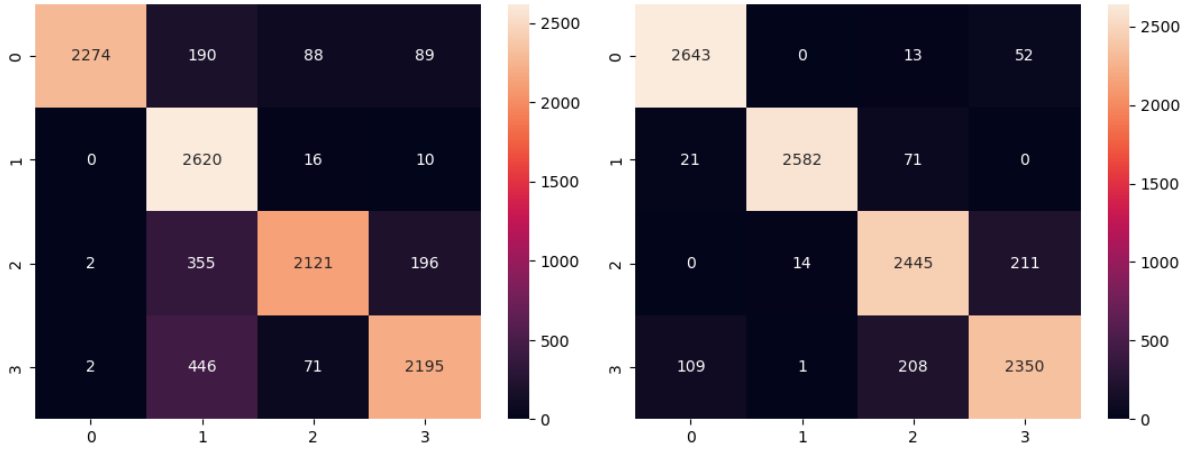
Figure 19: Unseen data on 2-Layered root model for both sensors

4.1.2.2 Sitting Models The sitting model is identical in structure to the Root model however here the model only accepts the four variations of sitting [Figure 5]. We include "Desk Work" here as the most often referred to activity during this is sitting and standing desks are not prominent around the university so we decided to impose some structure on the data via its position at the sitting node.

Looking at the test data results seen in Figure 21 we see strong diagonals for both sensors with good accuracy of 86% for Thingy and 93% for RESpeck. We see the typical evaluation of Thingy training almost twice as long as RESpeck for poorer results. However, in this case for the unseen data, we have a break in this trend as Thingy although having a lower testing accuracy outperforms the RESpeck sensor on unseen data. We see here that the Thingy sensor performs well at 89% and the RESpeck lags behind at 86%.

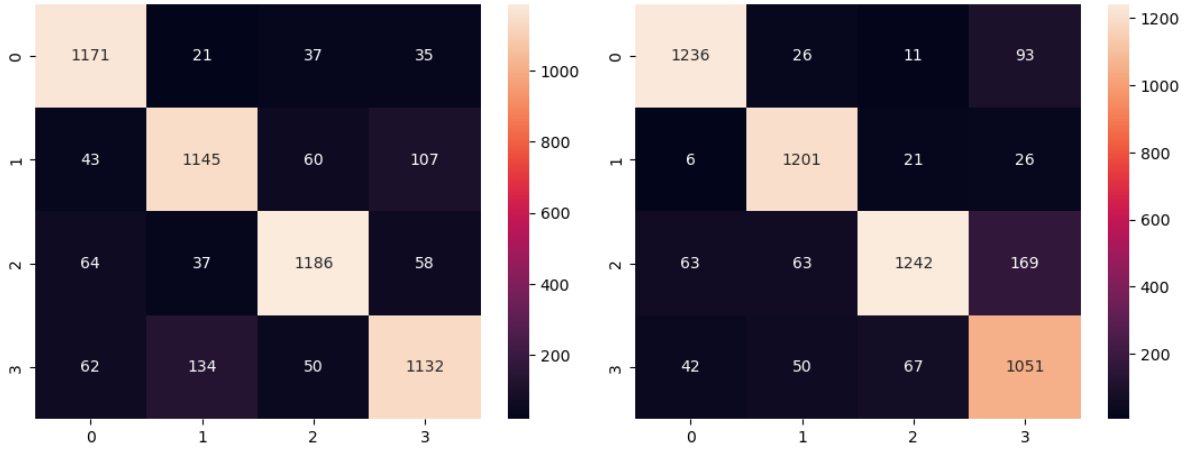
This is because the Thingy sensor typically performs better than the RESpeck under tasks that have minute differences such as "Sitting" and "Lying down" due to the precision of the sensor. Due to this fact, the converse is also true where the RESpeck sensor outperforms the Thingy on activities that require more general and varied person-to-person movements such as running and walking.

Possible improvements to this model would include an increase in kernel size for the RESpeck models to increase their competence compared to the Thingy models.



(a) Thingy 2-Layered sitting model on test data (b) RESpeck 2-Layered sitting model on test data

Figure 20: Test data on 2-Layered sitting model for both sensors

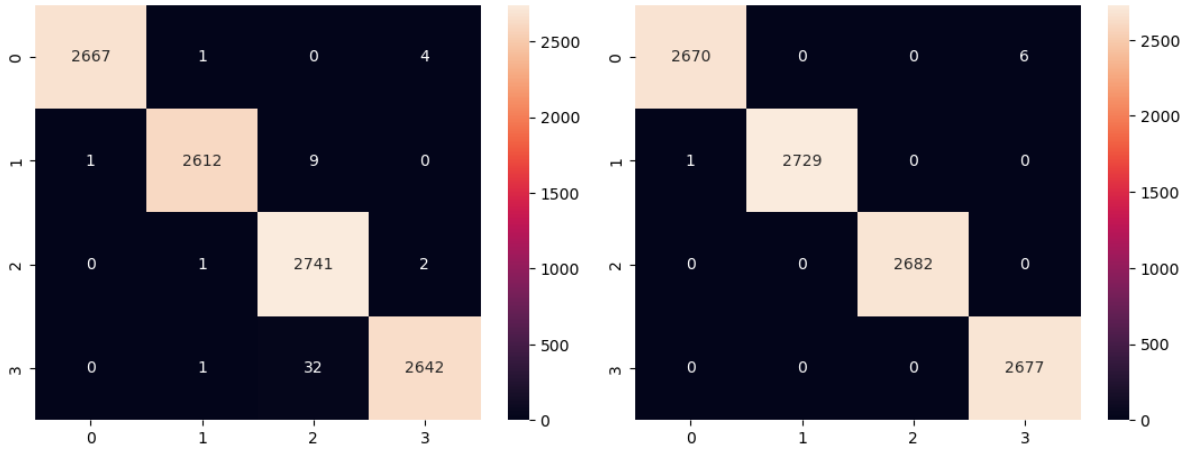


(a) Thingy 2-Layered sitting model on unseen data (b) RESpeck 2-Layered sitting model on unseen data

Figure 21: Unseen data on 2-Layered sitting model for both sensors

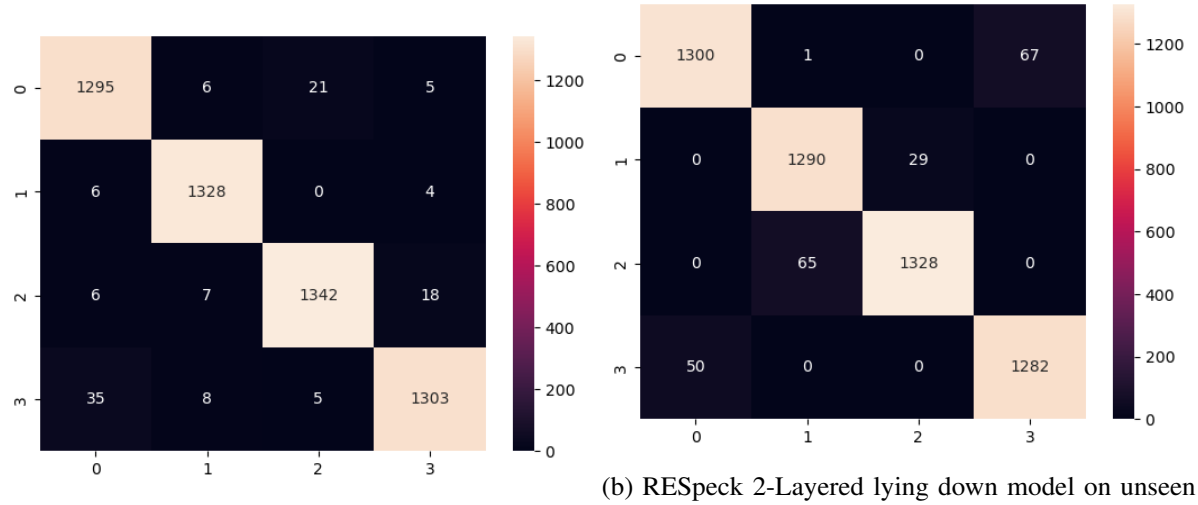
4.1.2.3 Lying Down Models Similar to the above "Sitting Model" [Section 4.1.2.2], the "Lying Down" model only predicts variants of "Lying Down", seen in Figure 5. Observing the test data, seen in Figure 22, we see strong diagonals from both sensors with excellent results of 99% for both models whilst maintaining this strong accuracy for unseen data with Thingy at 98% and RESpeck at 96%, seen in Figure 22 and Figure 23. We again see that the Thingy sensor outperforms the respec on unseen data, however, with the increased training time, shown in Table 4, and the nature of the sensor being more precise than the RESpeck such an interaction is exactly what we expected to see.

Additionally, something interesting of note is there appears to be an "inverse diagonal" present among the unseen RESpeck data. This is worrying as such an inverse correlation if an epidemic of some undiagnosed substructure within the data has been omitted when forming the training data. Solutions to fix this would be to increase the kernel size to allow the CNN to take in a wider frame of reference hence learning this substructure and adapting in turn.



(a) Thingy 2-Layered lying down model on test data (b) RESpeck 2-Layered lying down model on test data

Figure 22: Test data on 2-Layered lying down model for both sensors



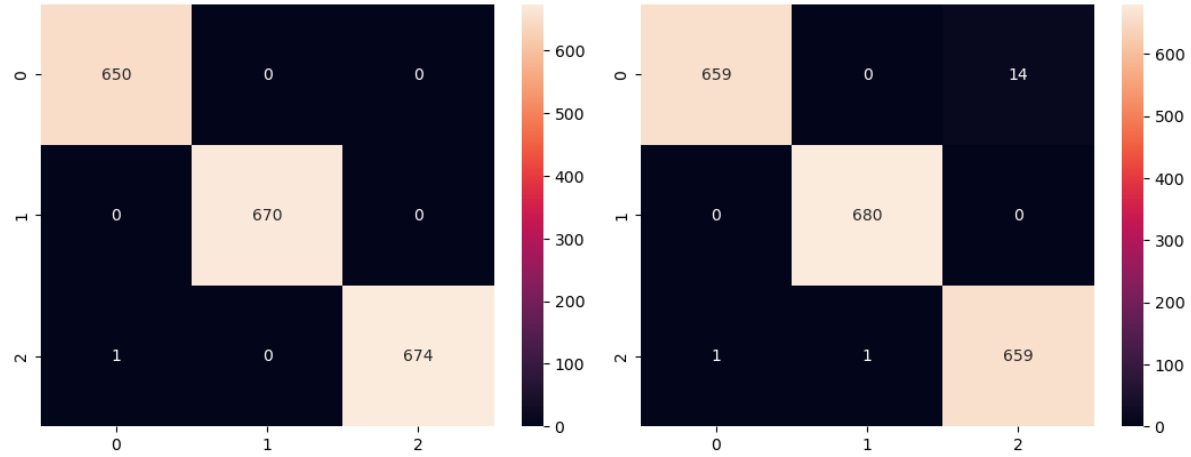
(a) Thingy 2-Layered lying down model on unseen data data

(b) RESpeck 2-Layered lying down model on unseen

Figure 23: Unseen data on 2-Layered lying down model for both sensors

4.1.2.4 Regular Movement Models The "Regular Movement" model only classifies three kinds of distinct movements due to their similarity. This does not include the "Movement" class as we have found it problematic in classification. It has been imposed onto the "Root" class with a flat bias attached. These models, due to their small size and the relatively easy task of three-class classification, have very fast training times and high accuracies at 99% for both sensors for both the test and unseen data, seen in Figure 24 and Figure 25.

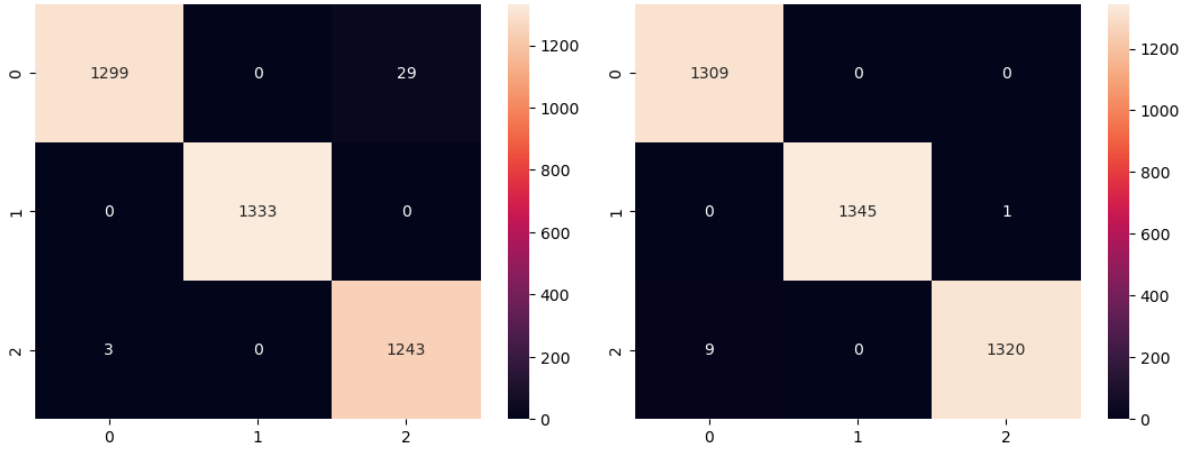
Although not significantly better than the RESpeck model, the Thingy model still has some misclassification within the "Running" Class which supports the evidence that the RESpeck sensors are better at more general activities. Although these models do not need improving some experiments could be conducted to vary the kernel size and see if this will decrease this miss-classification and the Thingy model in general. Additionally, since both of these models were trained for the same duration, increasing the training time for the Thingy model could benefit in alleviating this issue.



(a) Thingy 2-Layered regular movement model on test data

(b) RESpeck 2-Layered regular movement model on test data

Figure 24: Test data on 2-Layered regular movement model for both sensors



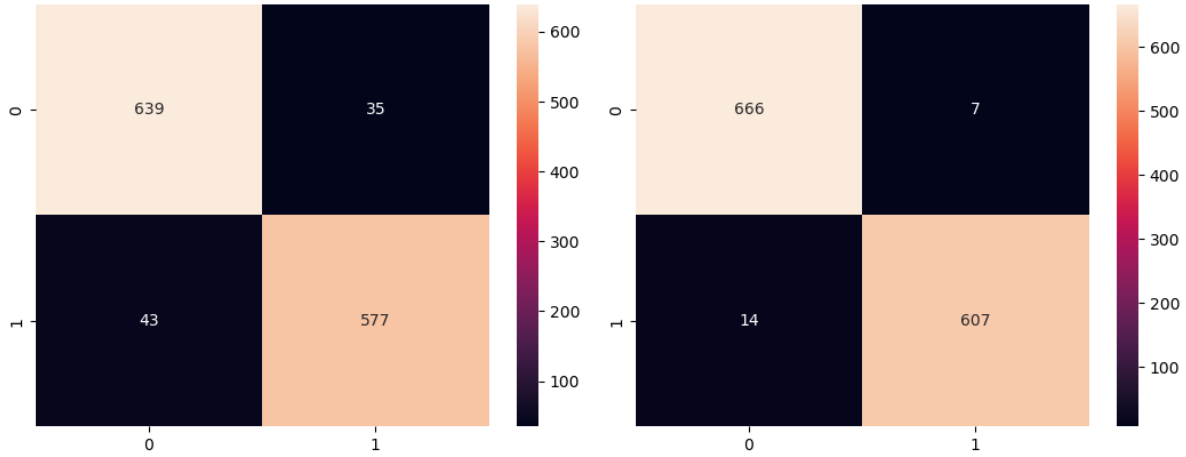
(a) Thingy 2-Layered regular movement model on unseen data

(b) RESpeck 2-Layered regular movement model on unseen data

Figure 25: Unseen data on 2-Layered regular movement model for both sensors

4.1.2.5 Stairs Models The "Stairs" model only classifies the two types of possible stair activity which involve climbing and descending. This is because the difference between these two activities is rather large and complex in itself hence a linear classifier will allow us to predict their differences effectively. Additionally, the decision to make these activities part of their own model was in part due to this complexity as in practice stairs are a frequent misclassification contributor.

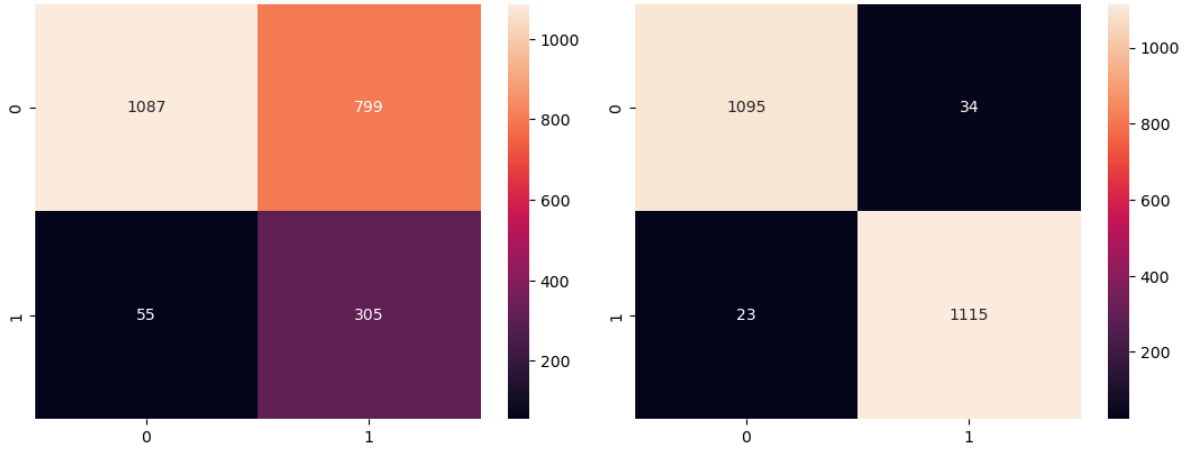
Looking at the test data, seen in Figure 26, we see strong diagonals for both sensors with the typical structure of the RESpeck sensor out-classifying the Thingy with 98% compared to only 93%. Although this is indicative of a strong classifier on unseen data, in practice we see that the covariance matrix for the Thingy model, seen in Figure 27, is unacceptable with only 57% accuracy compared to an impressive 97% for the RESpeck sensor. Although these models have been trained on the same structure and the same duration we see a polarising difference between them, one that we can only conclude is down to the sensor itself or the training data.



(a) Thingy 2-Layered stairs model on test data

(b) RESpeck 2-Layered stairs model on test data

Figure 26: Test data on 2-Layered stairs model for both sensors



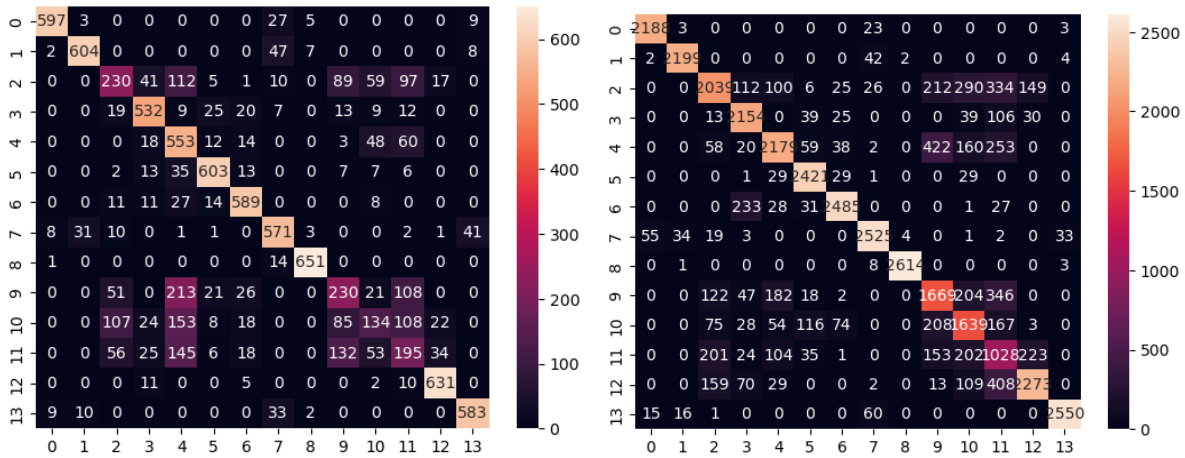
(a) Thingy 2-Layered stairs model on unseen data (b) RESpeck 2-Layered stairs model on unseen data

Figure 27: Unseen data on 2-Layered stairs model for both sensors

4.1.3 6-Layer CNN

This model was among the last that we trained due to its size and complexity. We reasoned that the main issue with the 3-Layer CNN is that there is too much confusion between ambiguous classes such as "Movement". To alleviate this we took two approaches, the first one being the tree structure [Section 4.1.2], and the second being increasing the number of layers, filters and kernel size.

This model was trained the longest out of all models in this report and even with this additional time and the appropriate GPU hardware, it was still not able to train to completion. This is due to the sheer amount of time it takes to train that many layers and the necessity of funding the hardware to do this. Despite this, the results are similar to the 3-Layered CNN [Section 4.1.1] which is trained for only around 100,000 epochs less. However, something to note that comes with this size is huge retention in accuracy across training, test and unseen data. We see this in the results, seen in Table 3, where the difference between peak accuracy with training data and unseen is only 2%. Looking more specifically at the test and unseen data, seen in Figure 28, we see both test and untrained data returned 82% accuracy which is impressive. We also note that between the two covariance matrices, there is a high similarity where the model miss-classifies activity. These are very similar areas of struggle that we have seen among the 3-Layered CNN however due to this retention we can evaluate that the accuracy once conquered on the training data would indeed reflect a solid unseen accuracy. To improve this model we could reduce the number of layers down to 5 whilst maintaining the kernel size and number of filters the same. This would improve training speeds allowing us to train to completion and maintain high retention between test data and unseen data.



(a) Combined 6-Layered CNN on test data (b) Combined 6-Layered CNN on unseen data

Figure 28: Test and unseen data on 6-Layered CNN using both sensors

4.2 Benchmarking

When designing a HAR system, there are several factors to consider such as accuracy, latency, processing power, and energy usage. In our case, the HAR application requires continuous data transmission from wearable sensors, and a reasonable latency is acceptable for offline classification. However, this also means that more resources need to be allocated for computational complexity.

More complex models may result in greater CPU usage, which means that the processor will spend more time working and less time idle. This increases both power consumption and heat output, which could lead to increased latency and energy consumption. Therefore, it is important to find the right balance between model complexity and energy efficiency when designing a HAR system.

The following sections discuss the performance of the machine learning models within the application, as well as the performance of the application itself.

4.2.1 Classifier Benchmarking

Within the system, each classifier builds a buffer of data values as information comes in from the sensors. Once this buffer is full, a classification is made. To evaluate the speed of each classifier, the time between the data buffer for each classifier being filled to a classification being made was measured. Each classifier was run until 100 classifications had been made during these tests. The timer began as soon as the buffer became full, and ended once the classification had been made with the time in nanoseconds being measured using the 'System' library. This process was repeated to measure the period between the data buffer becoming full and the database informing the application of a successful update. The table below shows the results of these tests:

	Time to classification (ms)			Time to database update (ms)		
Classifier	Minimum	Mean	Maximum	Minimum	Mean	Maximum
3-Layer CNN	2.22	6.53	37.75	24.56	33.81	66.14
6-Layer CNN	2.14	6.62	8.83	25.19	31.74	62.35
Tree Model	0.49	2.68	9.57	20.16	32.91	61.13

Table 5

From the above table we can observe that the tree model is on average 2.5 times faster in reaching a conclusion based on the data provided from the sensors. Regardless, the delay in making a classification is relatively fast across all three models. The time for the classification to reach the database was dominated by the round-trip network delay to the database being used, which is to be expected. Given the application uses Google's "Europe-West 1" database in Belgium, the times observed are normal. The difference between the 3 and 6 layer CNNs is negligible. Within the 3-Layer-CNN, we notice a large maximum delay. However, observation of the values collected during the test suggest this high value to be an outlier.

4.2.2 Application Benchmarking

An inspection into the usage statistics of the application for each classifier was performed using Android Studio's built-inn profiler. The below figure displays an example of the profiler while running the application on a Samsung A53G. This mobile uses a 2.4GHz octa-core CPU and possesses 6GB of RAM [13].

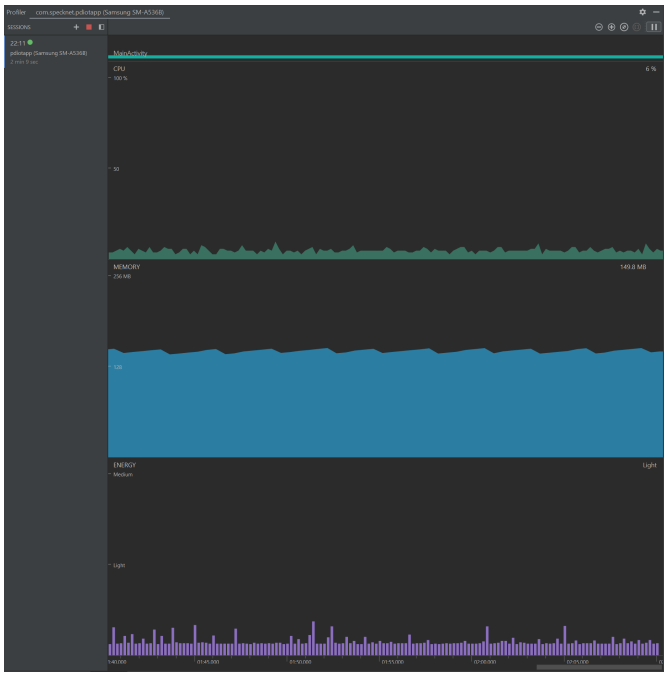


Figure 29: An example of the data provided by the in-built Android Studio profiler.

The application fluctuated significantly in its usage statistics when measured. Memory and CPU usage increased as more activities were selected in a given session. This value returned to baseline when the application was restarted. Upon inspection, the profiler apportioned this increase entirely to the 'Native' category. This category includes RAM used by the Android framework on the phone itself. Thus, we believe this increase in RAM usage is due to the framework caching each activity to facilitate better performance in switching activities. This effect necessitated a consistent approach to testing usage statistics across the application's pages; if one were to test a given page of the application, the usage statistics may be altered by previously visited activities. Thus, before testing each activity the application was restarted and only the activity being measured was selected.

As a baseline, the application was run on the landing page only for a period of one minute while connected to the sensors. During this time, the usage statistics remained consistent, with the only fluctuations being observed due to garbage-collection routines. The application was measured to use, on average (mean), 2% of the CPU, 128.1MB of RAM, and remained on the 'light' category of energy usage throughout.

	CPU Usage (%)			RAM usage (MB)		
Classifier	Minimum	Mean	Maximum	Minimum	Mean	Maximum
3-Layer CNN	8	14	27	169.2	175.3	186.0
6-Layer CNN	7	11	21	166.6	175.1	201.6
Tree Model	12	16	22	179.9	183.8	200.2

Table 6

It was observed that the energy usage of the app very rarely used a 'medium' amount of energy according to the profiler, so this statistic was not measured. In Table 6, we observe that the 6-layer CNN is the most efficient in both CPU and RAM usage. In contrast, the tree model performed the worst. These results are in line with our expectations, as the 6-layer CNN only requires one model file, whereas the tree model makes use of many.

5 Conclusion

In conclusion, we have successfully

We have successfully established a progression of CNN machine learning models. We determined that the small 3-Layered CNN does not provide us satisfactory results. Instead, we establish that a more sophisticated structure is needed. This paper, we proposed two methodologies to create this structure, a human-implied structure implemented in the tree model, and the other entirely learned in the case of the 6-Layer-CNN.

A sophisticated approach to the redesign of the mobile application was conducted using methods found in user design literature. The new design successfully conforms to the goals and aims determined. The application provides the user with 3 different options for classifying their activities, alongside the ability to store and view their previously collected data. We believe the design is an improvement upon the original, skeleton application provided and solves a number of the usability issues encountered.

Both the application and the machine learning models were evaluated using a number of techniques. The machine learning models were evaluated using a standard training, testing, and validation approach. The mobile application was profiled and found to be acceptably performant.

In all, this paper outlines the design, creation, testing, and benchmarking of an end-to-end, full-stack Human Activity Recognition application.

6 Future Work

During this course, we learned to handle time-series data from different IMU sensors and gained knowledge about the latest human activity recognition algorithms. We also engaged in extensive research and evaluations when selecting the appropriate machine learning model structure, which helped us to improve both our research and technical analysis abilities.

One potential future improvement could be to re-evaluate the trade-off between prediction latency, prediction accuracy, and data security. Currently, the primary goal of the HAR app is to accurately predict all 14 physical activities for health monitoring purposes, so computing power and storage are of paramount importance. In contrast, embedding the machine learning model on the IMU sensors results in lower prediction latency, reduced power consumption, and increased privacy as the sensor data is processed locally. This approach has some downsides such as limited storage and computing power on the sensors, as the nRF52832 only has 512 kB of flash memory and 64 kB of RAM. Additionally, scalability may be limited as the users can only update the model through a USB connection to build and flash the Zephyr applications, or by replacing the sensors. Ultimately, the decision of whether to embed the model on the sensors or in the android app will depend on the specific use case and requirements. This course has taught us the importance of carefully considering trade-offs when making system design choices, and the need to confirm the priorities of our application through thorough research and group discussion.

All models created in this paper could be improved by utilising data simultaneously from both sensors in a single model. This would allow the model to be better placed in understanding the current activity being performed by the user by taking into account the context of both sensors at once. Further, in future, a major aspect that could improve this system is support from the University in providing sufficient hardware to facilitate successful machine learning models. In total, nearly £30 was spent on Google Colab which we felt was still insufficient.

One hypothesis that was raised through our creation of the machine learning models was the potential for increased accuracy through the removal of the ‘desk work’ and/or ‘movement’ from the set of class labels. These high variance activities are hard for ML models to delineate. In ignoring these activities it may be possible to improve accuracy on the other activities, however this hypothesis requires experimentation to determine whether it is a valid concern.

The user interface could be further improved by more iterations of the design cycle outlined in the Methodology Section. Further, user studies could be conducted to determine usability issues not identified in this paper.

The application and its associated interface was only tested upon a small set of devices. In future, similar applications would be best placed in testing across a larger range of devices to maximise the number of issues found.

References

[1] Coblis — Color Blindness Simulator – Colblindor. URL: https://www.color-blindness.com/coblis-color-blindness-simulator/. Online; Accessed 22-January-2023.

[2] Design Thinking 101. URL: https://www.nngroup.com/articles/design-thinking. Online; Accessed 13-January-2023.

[3] Designing for the Web: Are There Colors You Should Avoid? URL: https://designshack.net/articles/ux-design/designing-for-the-web-are-there-colors-you-should-avoid. Online; Accessed 14-January-2023.

[4] The Double Diamond: A universally accepted depiction of the design process. URL: https://www.designcouncil.org.uk/our-work/news-opinion/double-diamond-universally-accepted-depiction-design-process. Online; Accessed 13-January-2023.

[5] Download Android Studio & App Tools. URL: https://developer.android.com/studio. Online; Accessed 19-January-2023.

[6] Firebase. URL: https://firebase.google.com. Online; Accessed 26-January-2023.

[7] GeForce RTX 2070 SUPER and GeForce RTX 2060 SUPER. URL: https://www.nvidia.com/en-us/geforce/news/nvidia-geforce-rtx-2060-super-rtx-2070-super-out-now. Online; Accessed 27-January-2023.

[8] Google Colaboratory. URL: https://colab.research.google.com. Online; Accessed 27-January-2023.

[9] Guide to hamburger menu design. URL: https://www.justinmind.com/blog/hamburger-menu. Online; Accessed 14-January-2023.

[10] MPAndroidChart/PieChartRenderer.java at master · PhilJay/MPAndroidChart. URL: https://github.com/PhilJay/MPAndroidChart. Online; Accessed 26-January-2023.

[11] Nordic Thingy:52 Prototyping platform. URL: https://www.nordicsemi.com/Products/Development-hardware/Nordic-Thingy-52. Online; Accessed 26-January-2023.

[12] OAuth 2.0 — OAuth. URL: https://oauth.net/2. Online; Accessed 26-January-2023.

[13] Samsung Galaxy A53 | Specs, Battery & Camera. URL: https://www.samsung.com/uk/smartphones/galaxy-a/galaxy-a53-5g-awesome-blue-128gb-sm-a536blbneub. Online; Accessed 26-January-2023.

[14] Tree-Based Models. URL: https://c3.ai/glossary/data-science/tree-based-models. Online; Accessed 26-January-2023.

[15] tsfresh — documentation. URL: https://tsfresh.readthedocs.io/en/latest. Online; Accessed 27-January-2023.

[16] What is monochromatic art, how does it impact accessibility, and why top UI designers love it | The Design Project. URL: https://designproject.io/blog/monochromatic-art. Online; Accessed 17-January-2023.

[17] Cost Per Frame: Best Value Graphics Cards Right Now. URL: https://www.techspot.com/article/2454-cost-per-frame-best-value-gpu, April 2022. Online; Accessed 26-January-2023.

[18] Mohammad Afaneh. BLE connection intervals and events in under 5 minutes. URL: https://novelbits.io/ble-connection-intervals, May 2016. Online; Accessed 26-January-2023.

[19] Ling Bao and Stephen S Intille. Activity recognition from user-annotated acceleration data. In *Lecture Notes in Computer Science*, Lecture notes in computer science, pages 1–17. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[20] Adrian E Bauman, Rodrigo S Reis, James F Sallis, Jonathan C Wells, Ruth J F Loos, Brian W Martin, and Lancet Physical Activity Series Working Group. Correlates of physical activity: why are some people physically active and others not? *Lancet*, 380(9838):258–271, July 2012.

[21] Y. Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. volume 60, page 6, 06 2009.

[22] William Chan, Navdeep Jaitly, Quoc V. Le, and Oriol Vinyals. Listen, attend and spell. *CoRR*, abs/1508.01211, 2015.

[23] J Chung, Ç Gülçehre, K Cho, and Y Bengio. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. arXiv*. 2014.

[24] Mingtao Dong, Jindong Han, Yuan He, and Xiaojun Jing. HAR-net: Fusing deep representation and hand-crafted features for human activity recognition. In *Lecture Notes in Electrical Engineering*, Lecture notes in electrical engineering, pages 32–40. Springer Singapore, Singapore, 2019.

[25] G. B. Drummond, A. Bates, J. Mann, and D. K. Arvind. Characterization of breathing patterns during patient-controlled opioid analgesia. *British Journal of Anaesthesia*, 111(6):971–978, December 2013. Publisher: Elsevier.

[26] World Leaders in Research-Based User Experience. 10 Usability Heuristics for User Interface Design. URL: `ps://www.nngroup.com/articles/ten-usability-heuristics/`. Online; Accessed 14-January-2023.

[27] Teodora Georgescu. A remote pulmonary rehabilitation system using the wearable RESpeck monitor. `https://project-archive.inf.ed.ac.uk/ug4/20201856/ug4_proj.pdf`.

[28] Carles Gomez. Overview and evaluation of bluetooth low energy: An emerging Low-Power wireless technology. " sensors. *Sensors*, 12:11734–11753, 2012.

[29] AMD Group. Amd ryzen™ 7 3800x | ryzen™ desktop processors | amd.

[30] Jihoon Hong and Tomoaki Ohtsuki. A state classification method based on space-time signal processing using svm for wireless monitoring systems. In *2011 IEEE 22nd International Symposium on Personal, Indoor and Mobile Radio Communications*, pages 2229–2233, 2011.

[31] W3C Web Accessibility Initiative (WAI). WCAG 2 Overview. URL: `https://www.w3.org/WAI/standards-guidelines/wcag`. Online; Accessed 14-January-2023.

[32] Frederik Kratzert, Martin Gauch, Grey Nearing, Sepp Hochreiter, and Daniel Klotz. Niederschlags-Abfluss-Modellierung mit long Short-Term memory (LSTM). *Österr. Wasser- Abfallwirtsch.*, 73(7-8):270–280, August 2021.

[33] Oscar D. Lara and Miguel A. Labrador. A survey on human activity recognition using wearable sensors. *IEEE Communications Surveys & Tutorials*, 15(3):1192–1209, 2013.

[34] Song-Mi Lee, Sang Min Yoon, and Heeryon Cho. Human activity recognition from accelerometer data using convolutional neural network. In *2017 IEEE International Conference on Big Data and Smart Computing (BigComp)*. IEEE, February 2017.

[35] Yongmou Li, Dianxi Shi, Bo Ding, and Dongbo Liu. Unsupervised feature learning for human activity recognition using smartphone sensors. In *Mining Intelligence and Knowledge Exploration*, Lecture notes in computer science, pages 99–107. Springer International Publishing, Cham, 2014.

[36] Aaron Marcus. Principles of Effective Visual Communication for Graphical User Interface Design. In *Readings in human–computer interaction*, pages 425–441. Elsevier, 1995.

[37] Bella Martin and Bruce Hanington. *Universal Methods of Design: 100 Ways to Research Complex Problems, Develop Innovative Ideas, and Design Effective Solutions*. Rockport Publishers, 2012.

[38] Jeremiah Okai, Stylianos Paraschiakos, Marian Beekman, Arno Knobbe, and Claudio Rebelo de Sa. Building robust models for human activity recognition from raw accelerometers data using gated recurrent units and long short term memory neural networks. In *2019 41st Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. IEEE, July 2019.

[39] Sadiq Sani, Stewart Massie, Nirmalie Wiratunga, and Kay Cooper. Learning deep and shallow features for human activity recognition. In *Knowledge Science, Engineering and Management*, Lecture notes in computer science, pages 469–482. Springer International Publishing, Cham, 2017.

[40] Keras Team. Keras documentation: Conv1d layer.

[41] Alireza Abedin Varamin, Ehsan Abbasnejad, Qinfeng Shi, Damith Ranasinghe, and Hamid Rezatofighi. Deep auto-set: A deep auto-encoder-set network for activity recognition using wearables. November 2018.

[42] Jie Xie, Kai Hu, Guofa Li, and Ya Guo. Cnn-based driving maneuver classification using multi-sliding window fusion. *Expert Systems with Applications*, 169:114442, 2021.

[43] Jie Yin, Qiang Yang, and Jeffrey Junfeng Pan. Sensor-based abnormal human-activity detection. *IEEE Transactions on Knowledge and Data Engineering*, 20(8):1082–1090, 2008.

[44] Licheng Zhang, Xihong Wu, and Dingsheng Luo. Real-time activity recognition on smartphones using deep neural networks. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*. IEEE, August 2015.

A Class Labels

```
#### Define Lables ####
class_labels = {
  "Climbing stairs": 0,
  "Descending stairs": 1,
  "Desk work": 2,
  "Lying down left": 3,
  "Lying down on back": 4,
  "Lying down on stomach": 5,
  "Lying down right": 6,
  "Movement": 7,
  "Running": 8,
  "Sitting bent backward": 9,
  "Sitting bent forward": 10,
  "_Sitting_": 11,
  "Standing": 12,
  "Walking": 13
}
```

Figure 30: Dictionary used for 14 class CNN