3.1 Suspicious Activity Detection Using Mobile Sensor Data via Modified Subspace KNN (msK)

With the proliferation of mobile sensors, the data they capture shouldn't go un-tapped. Today, crafting black box software to a mass, such data has become an uncomplicated endeavor. Leveraging this data, it becomes feasible to unearth suspicious and unlawful events. This section introduces an innovative forensic investigation approach using a modified subspace KNN (msK) algorithm, designed for detecting suspicious activities from mobile sensor data.

Figure 3.1 shows the block diagram of the modified subspace KNN, devised to identify suspicious activity within mobile data. The data we employed was extracted from a wireless stream, comprising 29 bytes (Packet size comprising of all sensor readings) of information for each discrete time instant. This data was partitioned into smaller subsets termed subspaces. Within each subspace, a distinct K-Nearest Neighbors (KNN) algorithm was applied. The KNN, a predictive method, evaluates neighboring data points to make predictions.

Post KNN analysis across each subspace, we aggregated the outcomes and tabulated the votes from each KNN run. To result in a definitive decision, we considered the maximum vote count along with its corresponding probability. The determination of suspicious activity rested on whether the votes for such activity

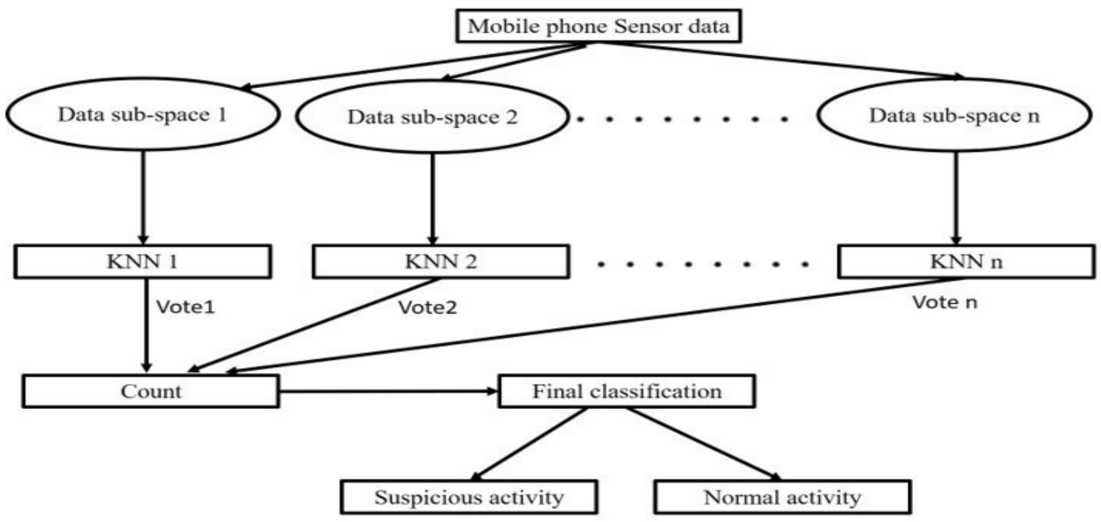


Fig. 3.1: Block diagram illustrating the application of modified subspace KNN for the detection of suspicious activities from mobile data.

The mobile data is acquired from a wireless stream, yielding 29 bytes per time instance. This data is segmented into N subspaces, each subjected to a separate KNN analysis. The outcomes from individual KNNs are analyzed and decided based on a voting. The final classification hinges on the maximum votes, accompanied by their probabilities. Suspicious activity detection occurs if the votes for suspicious activity exceed those for normal activity. Typically, the number of subspaces is maintained as an odd value to prevent equal vote distribution between classes. In this case, we initiated with 3 subspaces, incrementing subsequently to 29 in steps of 2.

Outnumbered those for normal activity. To ensure impartiality, we maintained the count of subspaces as an odd value, thereby avoiding an equal vote distribution between the two classes – suspicious and normal. In our implementation, we initially set the number of subspaces to 3, subsequently incrementing it to 29 in steps of 2.

This methodology facilitates comprehensive analysis of mobile data by employing subspaces, KNN algorithms, and vote counting to discern suspicious activity patterns.

3.1.1 Collection of mobile data

For the collection of mobile data, a black box application was developed, capable of acquiring various sensor readings including GPS coordinates, Accelerometer data, Gyroscope data, Magnetic field measurements, Orientation data, Linear acceleration values, Gravity readings, Rotation values, Pressure measurements, and Battery temperature. The data acquisition was facilitated through the User Datagram Protocol (UDP) stream, utilizing port 5555 for communication. Four distinct interval selection options (Capture rates) were available, each denoting the number of packets received per unit time. Each packet utilized in this study comprised 29 bytes of information.

The black box application could seamlessly operate in the background. It was derived from the open source IMU GPS Streaming app. The current investigation was carried out on Android 10 Funtouch OS 10 and Android platforms.

3.1.2 Modified subspace KNN

The mobile data was collected from a wireless stream at a rate of 29 bytes per time instance. Each time instance dataset was subdivided into N subspaces, and each KNN was applied separately. Finally, the votes from each KNN are totaled. For the final classification, the maximum number of votes with their probability is taken into account. Suspicious activity is detected if the number of votes cast is greater than the number of votes cast for normal activity. Typically, the number of subspaces was kept odd, to avoid giving equal votes to the both classes. In our case, we start with a subspace n = 3 and gradually increase it to 29 in steps of 2

i.e. n=5, 7, 9, 11....

The modified subspace KNN is a method that uses K neighborhood classification with parallel implementation. Here, to create multiple KNN models, data is sampled into subspaces. For example, out of 999 readings, any five readings are randomly chosen and then compared with the current time point. If the number of points is closer to normal activity compared to suspicious activity, then the current point is classified as normal. The main modification to the subspace KNN comes from the fact that we not only give the output class but also give the output probability that the current value belongs to that particular class.

For the probability determination, we calculate the sum of the Euclidean distance of the point from each of the K points in the KNN. The formula for the Euclidean distance is given in equation (1).

$$D_i = \sqrt{(x_i - x_c)^2 + (y_i - y_c)^2} \tag{3.1}$$

Where, x_c, y_c are the coordinates of the current data point being classified. x_i, y_i are the coordinates of the i^{th} neighbor, and i varies from 1, 2,...K D_i is the Euclidean distance for the i^{th} neighbor and i varies from 1, 2,...K.

The formula for probability computation is given in equation (2).

$$P_S = \frac{\frac{1}{L}\sum_{i=1}^{L} D_s i}{\frac{i}{K-L}\sum_{j=1}^{K-L} D_n j + \frac{i=1}{L}\sum_{1}^{L} D_s i} \tag{3.2}$$

Where, P_S is the probability of suspicious activity. L is the number of neighbors that belong to the suspicious class. D_Si is the Euclidean distance from the current point to all i points that belong to the suspicious class, and i varies from 1, 2,...L. $K - L$ is the number of neighbors that belong to the normal class. D_nj is the Euclidean distance from the current point to all j points that belong to the normal class, and j varies from 1, 2,...K-L.

3.2 Activity classification

After detecting suspicious activity using our modified subspace KNN, we decided to explore existing multiclass classification techniques to distinguish between different types of activities. One widely recognized classifier for time series data is Long Short-Term Memory (LSTM), and we also considered DenseNet.

For the LSTM based classification, we utilized standard libraries, including TensorFlow, Keras, Pandas, NumPy, and Matplotlib.

We will discuss about these libraries in short:

- TensorFlow, an open source machine learning library developed by Google, serves as a powerful platform for creating and training deep learning models. It employs data flow graphs to represent computations, where nodes represent mathematical operations and edges represent multidimensional data arrays (tensors) communicated between nodes. Tensors, serving as the central unit of data in TensorFlow, are pivotal for modeling and training. Tensor Flow's efficiency in building and training deep learning models is notable.
- It allows for the creation of computational graphs, which define the flow of operations. These graphs can be executed on various hardware platforms, including CPUs and GPUs, making TensorFlow a versatile and powerful tool for machine learning applications.
- Keras, an open source high level neural network library written in Python by Francois Chollet, a Google engineer, is tailored for building and training neural networks. It operates atop deep learning frameworks (like TensorFlow, Theano, or CNTK), fostering rapid experimentation and iteration to stream line deep learning model development. Keras provides a user friendly API, abstracting away low level intricacies and emphasizing model architecture and experimentation. It supports multiple backends, enabling developers to select the most suitable one for their needs. Keras models can be developed in Python or R and executed with TensorFlow, Theano, CNTK, or MXNet, on various platforms like CPU, NVIDIA GPU, AMD GPU, TPU, Android, iOS, and Raspberry Pi. With wide industry adoption by companies like Netflix, Yelp, and Uber, Keras boasts a robust

research community and contributions from tech giants like Microsoft, Google, NVIDIA, and Amazon. Its ease of learning and prototyping, coupled with its flexibility and multi backend support, make Keras a preferred choice for neural network development, catering to both beginners and experts alike.

- Pandas, a versatile Python library, is integral for data manipulation and analysis, simplifying tasks related to structured data. It is adept at handling tabular data, such as spreadsheets or SQL tables, providing data structures and functions for efficient operations. Built on NumPy, Pandas is widely used by data analysts, scientists, and engineers. It introduces two primary data structures: Series, for one-dimensional labeled arrays, and DataFrame, for two dimensional labeled data structures resembling tables or spreadsheets. These structures enable easy manipulation, indexing, and operations involving data. Common use cases include data cleaning, merging, and joining datasets, grouping and aggregation, data visualization, and statistical analysis and machine learning when used alongside SciPy and Scikitlearn.
- NumPy (Numerical Python) is a fundamental, open source Python library that plays a pivotal role in scientific computing, data analysis, and numerical computations. It boasts several key features:
  - Multidimensional Arrays: NumPy offers a powerful array processing package, with its primary data structure being the ndarray (n-dimensional array). These arrays can hold diverse data types (integers, floats, etc.) and enable efficient operations on extensive datasets.
  - Efficient Numerical Operations: NumPy is optimized for numerical computations, ensuring faster performance than standard Python lists.
  - Broadcasting: Facilitates element wise operations on arrays of various shapes.
  - Linear Algebra: Provides robust functions for matrix operations, eigen-values, and more.
  - Random Number Generation: Equipped with tools for random sampling and distributions.
  - Indexing and Slicing: Enables easy access and manipulation of array elements.

- Integration with Other Libraries: Often used in conjunction with libraries like SciPy, Matplotlib, and Pandas for comprehensive data analysis and visualization.

- Matplotlib is a powerful visualization library in Python primarily utilized for creating 2D plots of arrays. It serves as a visualization utility for data analysis and exploration, enabling the creation of high quality visualizations and graphs. Built on NumPy arrays, Matplotlib is designed to work seamlessly with the broader SciPy stack. Introduced by John D. Hunter in 2002, it offers several types of plots, making it versatile for various tasks, such as line plots, bar plots, histogram plots, scatter plots, pie charts, and area plots.

Specifically, we made use of the Matplotlib library for pie plots. Data importation was done through Google Drive. In the Keras framework, we employed traditional LSTM in combination with sequence preprocessing and DenseNet for classification. Our primary contribution in designing a forensic network was the sequential connection of DenseNet with LSTM. We employed the Adam optimizer to minimize errors in this deep learning model. The Adam optimizer (Adaptive Moment Estimation) is an advanced gradient descent optimization algorithm that adjusts the learning rate for each parameter dynamically, using estimates of first and second moments of gradients. It combines the benefits of AdaGrad and RMS Prop optimizers to handle sparse gradients and nonstationary objectives.

3.2.1 Hardware details

We deployed algorithm on a computer equipped with an Intel i9-9900 processor featuring a 16 MB cache and a maximum frequency of up to 5 GHz. This processor has 8 cores and 16 threads, paired with DDR4 2666 MHz RAM, with substantial 64 GB of RAM memory. To facilitate the deep neural network training, we used an RTX 3060 GPU with 12 GB of GDDR6 RAM. Storage was handled by a Samsung 980 1 TB NVMe solid state drive, delivering impressive internal read speeds of 3500 MB/sec.

The mobile phone used for testing is a device that supports GSM, HSPA, and LTE technologies. It features a body dimension of 162 x 76.5 x 9.1 mm and weighs 197 grams, with a glass front, plastic back, and plastic frame, and supports dual Nano SIM cards. The display is a 6.53 inch IPS LCD with a resolution of 1080 x 2400 pixels, providing a screen to body ratio of approximately 83.1%. Running on Android 10 with Funtouch 10.0, it is powered by the Qualcomm SM6125 Snap dragon 665 chipset, an octacore CPU, and Adreno 610 GPU. The device offers substantial memory options with 128GB of internal storage paired with either 8GB or 6GB of RAM and supports microSDXC cards. Its quad rear camera setup includes a 13 MP wide lens, an 8 MP ultrawide lens, a 2 MP macro lens, and a 2 MP depth sensor, while the 16 MP front camera handles selfies. Additional features include Wi-Fi, Bluetooth 5.0, GPS, and USB Type C connectivity. The phone is equipped with a non removable 5000 mAh battery supporting 15W wired charging.

3.2.2 Data collection and training details

All values from the training data were read as arrays and concatenated into a single line. The feature vector organized the data into sets of three float values from the linear array of the original data. Once a window size of 192 values was reached, the X feature vector was saved, and Y was marked as 1,0,0, indicating that the data belonged to the first channel. This process was similarly repeated for sensor readings.

The next activity was recorded and read as a separate linear array. All values from all sensors were likewise divided into a window size of 192, and the vector was marked this time as 0,1,0. Similarly, for the third activity, the vector was marked as 0,0,1. Initially, we focused solely on three criminal activities for consideration, each of which was dramatically reproduced:

Mobile data was recorded when a mobile phone was in the pocket of a woman struggling against an attack (WSAA). Mobile data was recorded when a mobile phone was in the pocket of a man who was forcibly being dragged by kidnappers to enter a car (MKFE). Mobile data was collected from the pocket of a woman while she was running (WRUN). These three activities were dramatic recreation of incidents commonly associated with criminal activity. Trained professionals and an expert cameraman were involved in performing these activities.

For training, all time series data was concatenated along with their corresponding activity labels. A total of 947 readings were used in LSTM, each with a window size of 192. For testing, a similar LSTM approach was employed, where the data was windowed and concatenated into a single float vector, resulting in labels represented as 1,0,0: 0,1,0: 0,0,1. In total, 736 different files were used for subsequent denseNET-based testing.

3.2.3 Long Short Term Memory

For feature selection purposes, we used Long Short Term Memory (LSTM), which is a type of recurrent neural network (RNN) architecture that is well suited for processing and making predictions based on sequences of data. It was introduced by Hochreiter and Schmidhuber in 1997 and has since become a fundamental component in various applications, including 1D signal processing, like in our case. Speech recognition, time series forecasting, and NLP are some more examples where LSTM is used. Here are the key details and components of LSTM networks:

Memory Cells: LSTM networks are designed to capture and remember patterns over long sequences of data. They achieve this through the use of memory cells. These memory cells have the ability to store information for long durations, making them effective at capturing dependencies in sequential data.

Gates: LSTMs employ three different types of gates to control the flow of information within the memory cell and decide what to store, forget, and output.

Forget Gate: The forget gate determines what information from the previous time step should be discarded from the cell state. It takes both the previous hidden state and the current input as inputs and outputs a value between 0 and 1 for each component of the cell state. A value of 1 means ”keep this,” while a value of 0 means ”discard this.” Input Gate: The input gate decides what new information should be stored in the cell state. It also takes the previous hidden state and the current input, and it produces a new candidate value for the cell state.

Output Gate: The output gate decides what the next hidden state should be based on the cell state. It takes the current input and the previous hidden state and produces the new hidden state.

Hidden State: In addition to the cell state, LSTM networks have a hidden state at each time step. The hidden state is a filtered version of the cell state and carries information relevant to the predictions being made.

Backpropagation Through Time (BPTT): LSTMs are trained using back

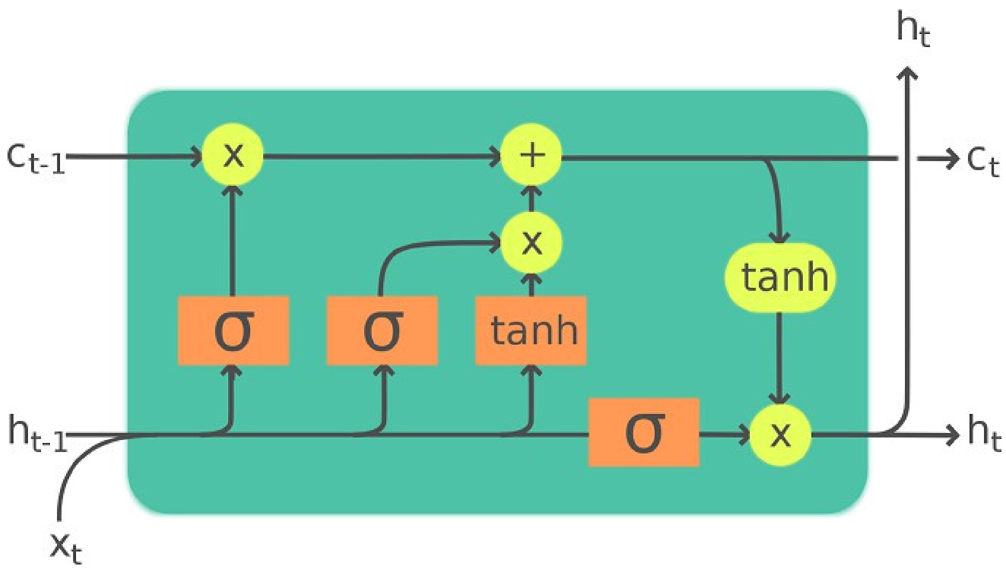


Fig. 3.2: LSTM details

Propagation, similar to feedforward neural networks. However, due to their recurrent nature, training involves back propagating errors through time, which can be computationally intensive. This is known as BPTT.

Vanishing Gradient Problem: LSTMs were introduced to mitigate the vanishing gradient problem, which affects traditional RNNs. The forget, input, and output gates in LSTM networks enable them to capture long range dependencies in data without suffering from the vanishing gradient problem.

Variants: Over time, several variants of LSTM have been developed to address specific challenges. One notable variant is the Gated Recurrent Unit (GRU), which simplifies the LSTM architecture while retaining much of its effectiveness.

LSTM networks as shown in figure 3.2 have proven to be highly effective in a wide range of applications where sequential data plays a crucial role. They are particularly valuable when dealing with time series data, text data, speech data, and any data where understanding dependencies over time is essential.

3.2.4 DenseNET details

A sequential model was employed with the initial layer being the dense net. The activation function used was Rectified Linear Unit (ReLU). The matrix was re-shaped to fit a dense net with a size of 128. Three additional dense layers followed, and softmax activation was used instead of ReLU. Loss was computed using mean squared error, and the optimizer utilized was Adam, with a learning rate of 0.01. The primary metric of interest was accuracy. Over the course of 16 epochs, the time per step decreased from 636 microseconds for the first epoch to 43 microseconds for subsequent epochs.

The results were evaluated based on 736 different samples, and accuracy scores, classification parameters, and a confusion matrix were employed for validation. These classification parameters included the F1 score, precision, and recall (sensitivity). To provide a clearer understanding of the three classes, the confusion matrix was normalized.

3.3 Mobile app

After delving into the intricacies of the proposed Dense Net based architecture, let’s explore the details of the mobile application developed for real time data collection. This application represents another significant contribution to the thesis, serving as a versatile tool for gathering data related to various activities. It’s worth noting that these activities aren’t limited to criminal scenarios; instead, we’ve designed a general-purpose data collection app that can be beneficial for researchers across different domains.

To initiate the collection of mobile data, users are required to connect sensors to their devices and create an object called ”MobileDEV” to facilitate data storage.

This app ensures the collection of sensor data even when the device lacks a network connection. It achieves this by utilizing a sensor data log, which is stored locally. The app periodically accesses the camera to acquire images at predefined resolutions, autofocus settings, and flash modes. In addition to images, the app logs data related to acceleration, angular velocity, magnetic field strength, orientation, and position.

Here’s a breakdown of the sensor data measurements:

- Acceleration is measured in meters per second squared (m/s^2).
- Angular velocity is measured in radians per second (rad/s).
- Magnetic field strength is measured in microtesla (μT).

- Orientation is determined considering elevation, XYZ coordinates, yaw, roll, and pitch.
- Position data includes latitude, longitude, speed, altitude, and course.
- Latitude is recorded in degrees, with positive values indicating north and negative values indicating south.
- Longitude is measured in degrees, with positive values indicating east and negative values indicating west.
- Speed is measured in meters per second (m/s).
- Altitude is measured in meters above sea level.
- Course is recorded in degrees with respect to true north.

To enable data transmission, the user must select the ”stream to cloud or log” option. Additionally, the user should choose ”send position data” in the background and activate ”auto upload.” For the proof of concept, we manually collected the data, classifying it as either normal or indicative of a fight scenario. To facilitate data collection, users installed the data logger app on their mobile devices.

We captured three similar fight sequences, each separated by an approximate time interval of 15 to 20 seconds. These sequences were recorded using the data logger app, and we configured the sampling rate to 300 milliseconds.

From these fight sequences, we extracted various parameters, including acceleration, angular velocity, magnetic field data, orientation, and position. This process was consistently applied to all three sequences. The resulting dataset comprised

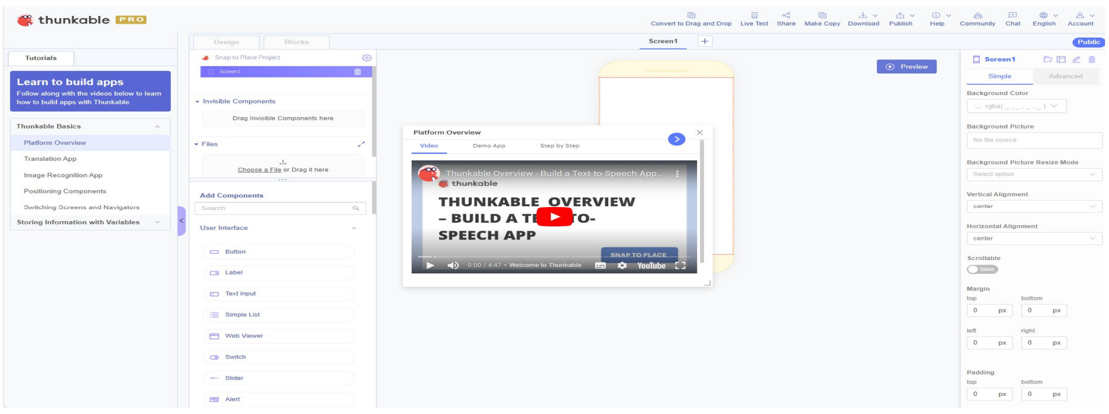


Fig. 3.3: Thunkable software front user design interface

all these collected insights, which we prepared for subsequent analysis and processing. We then compared this dataset to the one containing normal sequences for further evaluation.

After completing the multiclass classification using the LSTM with Dense NET approach for three classes, we opted to return to binary classification using real time data logged through our mobile app. For this task, we chose to employ the simplest machine learning algorithms, K-Nearest Neighbors (KNN) and Gaussian Support Vector Machine (SVM).

The reason for this choice is that we wanted to avoid introducing the complexity of DenseNET into this binary classification task. Additionally, using standard algorithms such as KNN and Gaussian SVM allows us to test the performance of the app against established methods rather than our proposed ones.

To train a model capable of distinguishing between normal and abnormal sequences, we employed two classification algorithms: k-nearest neighbors (KNN) and Gaussian Support Vector Machine (SVM). To generate a training dataset, we utilized a mobile application connected to various sensors deployed in different locations to collect real time data. This dataset was stored in an object referred to as ”MobileDEV.”

Thunkable Front User Design Interface: Thunkable’s interface as shown in figure 4.11 is divided into two main sections:

1. Design View: This visual workspace displays app’s layout, where one can drag and drop components like buttons, lists, and maps to build user screens. It resembles a typical visual development environment with a live preview of our app as we design.
2. Blocks Editor: This section houses the logic behind our app’s functionality. One can use drag-and-drop blocks to define how components interact and respond to user actions. Think of it as building the app’s behavior with visual puzzle pieces.

General Layout:

- Components Panel: Located on the left side, this panel houses all available UI components like buttons, text labels, images, maps, and sensors.

- Properties Panel: This panel on the right displays properties specific to the selected component, allowing one to customize its appearance and behavior.
- Screen Canvas: The central area displays the app's current screen layout in real-time, reflecting the components and their arrangement.
- Navigation Toolbar: Provides options for managing screens, previewing the app, and accessing project settings.

To build the app using thunkable , One needs to login the free thunkable account to login and then get started. The login page is as shown in figure 3.4

Once logged in, one can install thunkable and a menu will pop up as shown in figure3.5

There are many menu buttons, from simple screen sharing to calculating text to speech, from which users can select one and make the app.

Creating a mobile forensics app like the proposed one in Thunkable was definitely ambitious and technically challenging, but with careful planning and the right approach, it's achievable. Now we explain some of the basic steps involved in building the proposed app, with a breakdown of the main steps involved:

1. Design and Planning:
    - Define core functionalities: Identify the specific sensor data one want to capture (GPS, accelerometer, etc.), how one will handle timestamps, and what level of data analysis one can envision.

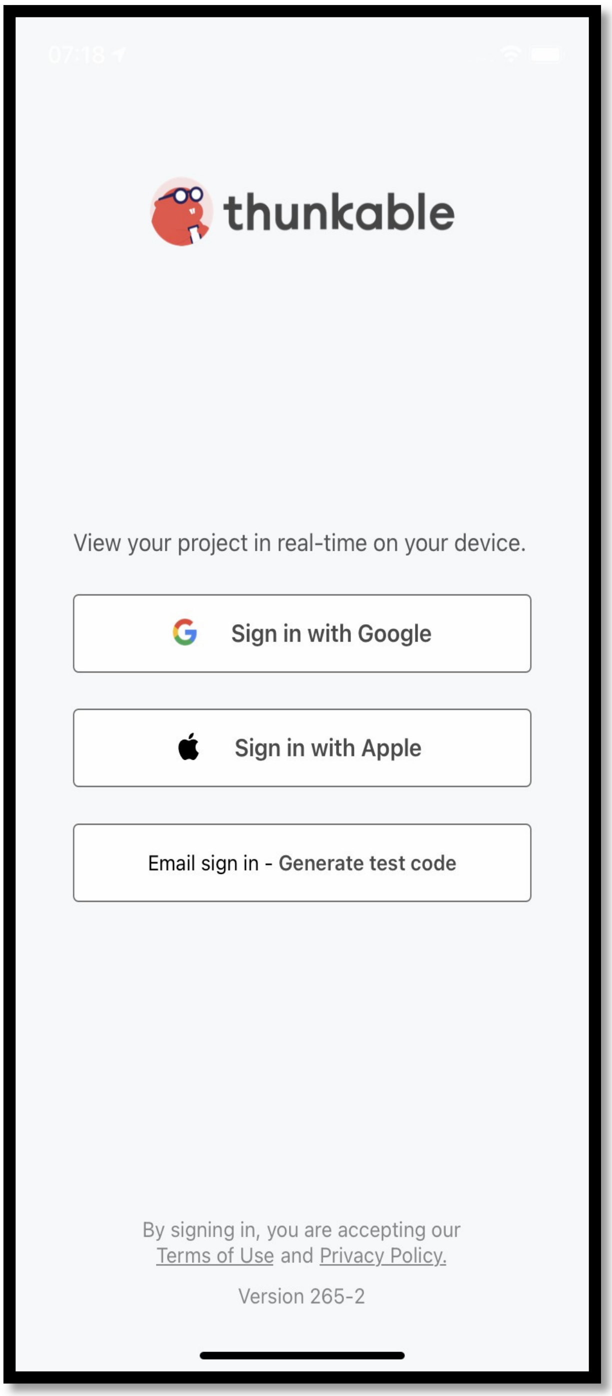

thunkable
View your project in real-time on your device.
Sign in with Google
Sign in with Apple
Email sign in - Generate test code
By signing in, you are accepting our
Terms of Use and Privacy Policy.
Version 265-2

Fig. 3.4 : Sign-in window at Thunkable

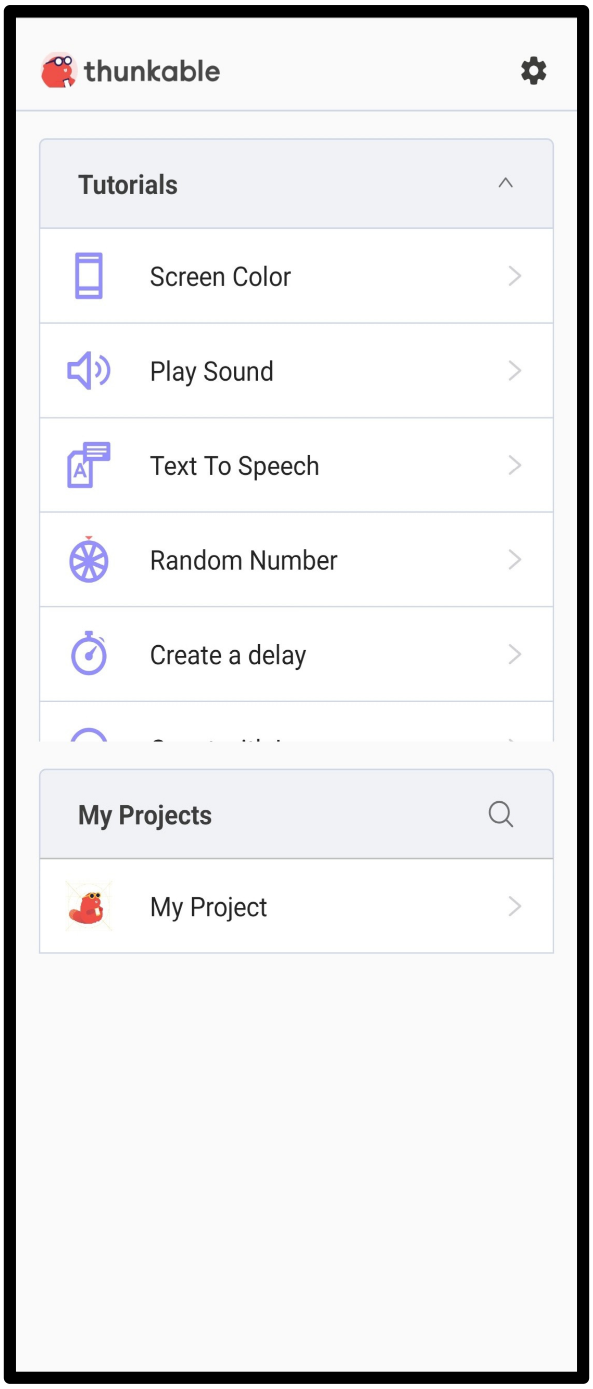


Fig. 3.5: Menu on thunkable app

- User interface: Plan the app's screens and layout, considering user consent mechanisms, data visualization, and secure storage options.
- Data security: Prioritize data encryption, secure storage techniques (local or cloud), and user control over data access and deletion.

2. Building the App in Thunkable:

- Data collection: Use Thunkable's built in sensor components to capture data from GPS, accelerometer, gyroscope, etc. Use the ”Clock” component for timestamps.
- Data storage: Utilize Thunkable's ”File” or ”Airtable” components to store data securely on the device or in the cloud. For advanced analysis, consider integrating with external databases.
- Data visualization: Use Thunkable's ”Charts” and ”Lists” components to visualize sensor data over time, like GPS tracks or accelerometer readings.
- User interface: Drag and drop UI components to create screens for data collection, visualization, and settings. Pay attention to user friendliness and clarity.

3. Advanced Features (Optional):
    - Data analysis: Consider integrating machine learning models for advanced analysis, like anomaly detection or pattern recognition in sensor data. This requires coding knowledge beyond Thunkable's block based interface.
    - Reporting: Implement features for generating reports or exporting data in formats useful for further analysis in forensic software.

3.4 KNN and Gaussian SVM

KNN is a classification technique that groups similar looking data into clusters, treating each cluster as a class. Conversely, Gaussian SVM is a classification technique that distinguishes datasets based on extracted features. Data is collected

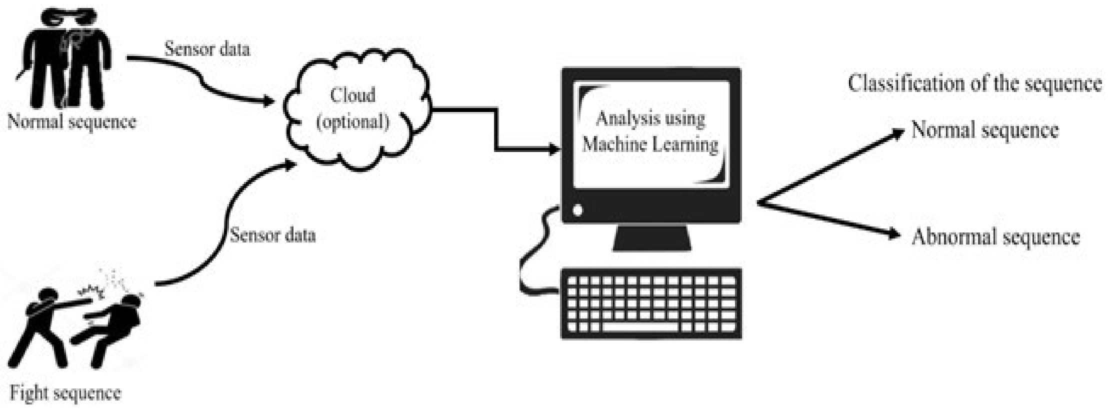


Figure 3.6: The flow of the proposed work with KNN and Gaussian SVM.

The datasets of normal and fight sequences are created by accessing multiple videos. The datasets are uploaded to the cloud. Analysis of these datasets is done with the help of machine learning. Lastly, the sensor data collected is classified into a normal and abnormal sequence with maximum accuracy.

Using various mobile sensors, and the collected data is stored in an object named "MobileDEV." We trained a machine learning model using KNN and Gaussian SVM, with the results stored in a database for further analysis. The system performs sequence classification, categorizing sequences as either "normal" or "abnormal." If a sequence is c lassified as "normal," the sensors are instructed to continue data collection. Conversely, if the sequence is classified as "abnormal," an abnormality report is generated. Figure 3.6 shows the proposed workflow diagram. We used multiple videos for the creation of normal and fight sequence datasets. This data is uploaded to the server for training. Machine learning is being used to analyze these datasets. Finally, the data collected in realtime is accurately classified into a normal and abnormal sequences.

3.5 Data Collection

For our research, we employed the Samsung S22 Ultra mobile phone as the primary device for data collection. This particular model features a variety of sensors, including an under display ultrasonic fingerprint sensor, an accelerometer, a gyro-scope, a proximity sensor, a compass sensor, and a barometer sensor. The ultrasonic fingerprint sensor serves the purpose of biometric authentication, while the accelerometer and gyroscope detect changes in device orientation. The proximity sensor identifies when the device is close to the user's face, the compass sensor determines the device's orientation relative to magnetic north, and the barometer sensor measures atmospheric pressure. Additionally, the Samsung S22 Ultra offers various features such as Samsung DeX, Samsung Wireless DeX for desktop experience support, Bixby natural language commands and dictation, and Ultra Wideband (UWB) support.

The objective of our research was to develop a mobile application capable of collecting data from mobile phones to assist in investigations or legal cases. To achieve this, we initiated the data collection process by gathering information for the training dataset. Trained actors were engaged to simulate ten real-life scenarios, encompassing situations such as a teenager running for survival, a man being forcibly dragged into a car during a kidnapping, and a woman calling for help while being kidnapped. The ten output classes for classification include descriptive labels such as ”Teenager Running for Survival,” ”Male Sitting in Car Just Before Accident,” ”Kids Fighting,” ”Woman Sitting in Moving Vehicle Just Before Car Crash,” ”Women Calling for Help During Kidnapping,” ”Woman Struggling Against Attackers,” ”Throwing of Mobile Phones to Protect Evidence,” ”Women Running to Save Herself,” ”Man Getting Dragged into Car During Kidnapping,” and ”Man Enters Vehicle by Forcibly Pushing Woman onto Back Seat.”

3.6 LSTM feature extraction

LSTMs (Long Short Term Memory networks) prove to be formidable tools in extracting features from 1D sensor data, particularly when temporal dependencies or patterns exist, as seen in our case where mobile data with discernible patterns was collected from sensors. The collaboration of LSTMs with Dense Net is instrumental in extracting features from this data.

There are notable strengths in employing LSTMs for feature extraction. Notably, they excel in capturing long term dependencies, eliminating the need for manually defining features as in traditional methods. Their ability to discern subtle relationships between data points at distant time steps is crucial for capturing trends, cycles, and concealed patterns within the data.

LSTMs are well suited for handling sequential data, a characteristic common in sensor data. They process sequences effectively, considering the context of each data point within its temporal context. This enables them to extract features influenced by both past and future values, resulting in richer representations.

Moreover, LSTMs showcase adaptability to diverse sensor data types, including accelerometer readings, temperature measurements, and vibration signals. By adjusting the model architecture and learning parameters, they can be tailored for specific sensor characteristics, extracting relevant features for different applications.

The feature extraction process with LSTMs follows a systematic approach: Preprocessing: Sensor data acquired through a mobile app undergoes preprocessing before feeding it to the LSTM. This may involve scaling, normalization, and segmentation into appropriate time windows based on the expected feature timescale.

LSTM Architecture: The LSTM network architecture is designed based on data complexity and desired features. Optimization involves factors like the number of hidden layers (2), activation functions (tanh), and learning rate (0.001) for optimal performance.

Feature Extraction: As the LSTM processes sequential data, it learns internal representations of information. Extracted features include dominant frequencies, trends, amplitudes, and relationships between past and future values.

Output Interpretation: Features extracted by the LSTM are accessed through various methods, depending on the model architecture. These can be directly accessed from hidden states, passed through additional layers for further processing, or utilized for downstream tasks such as classification or anomaly detection.

In the context of human activity recognition, LSTMs prove effective in analyzing accelerometer data from mobile devices. They extract features like movement patterns, steps, and gestures, facilitating activity recognition for forensics.

However, our study reveals that LSTMs can be computationally expensive and require larger datasets for effective training compared to simpler models. Hence, selecting the right hyperparameters and architecture is crucial for optimal feature extraction performance.

Combining LSTMs with other neural network types, such as DenseNet, can leverage both spatial and temporal dependencies for even more informative feature extraction. In summary, LSTMs present a potent approach for extracting meaningful features from 1D sensor data, enabling diverse applications, especially in our forensic context.

3.7 IFDenseNET

IFDenseNET-138 Conceptual Diagram for Classifying Mobile Phone Sensor Data 138 Conceptual Diagram for Classifying Mobile Phone Sensor Data.

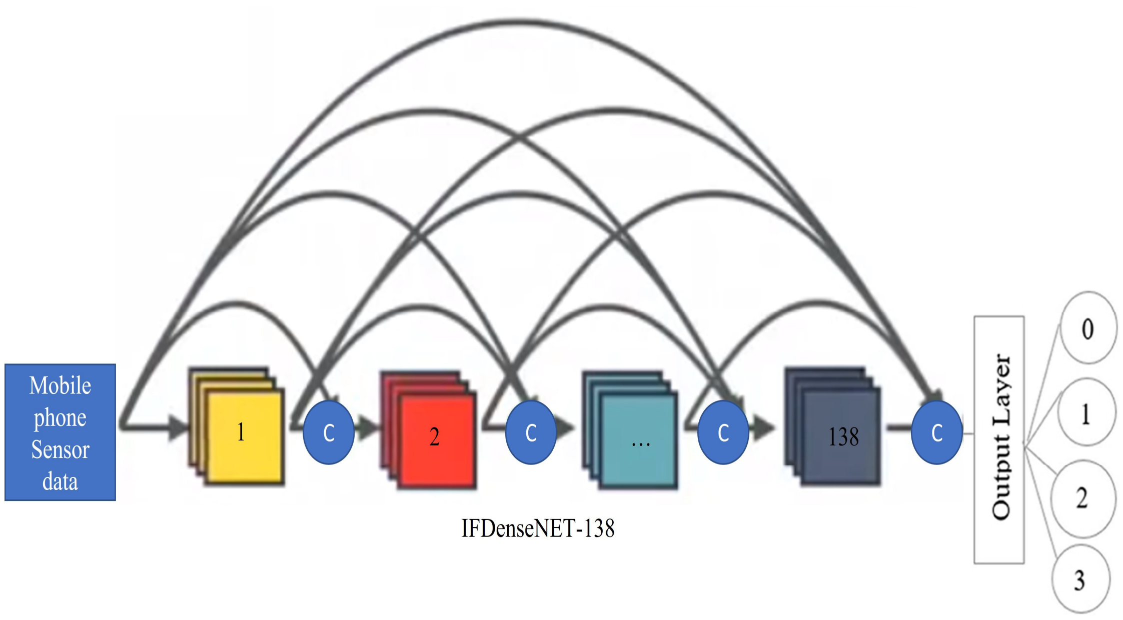
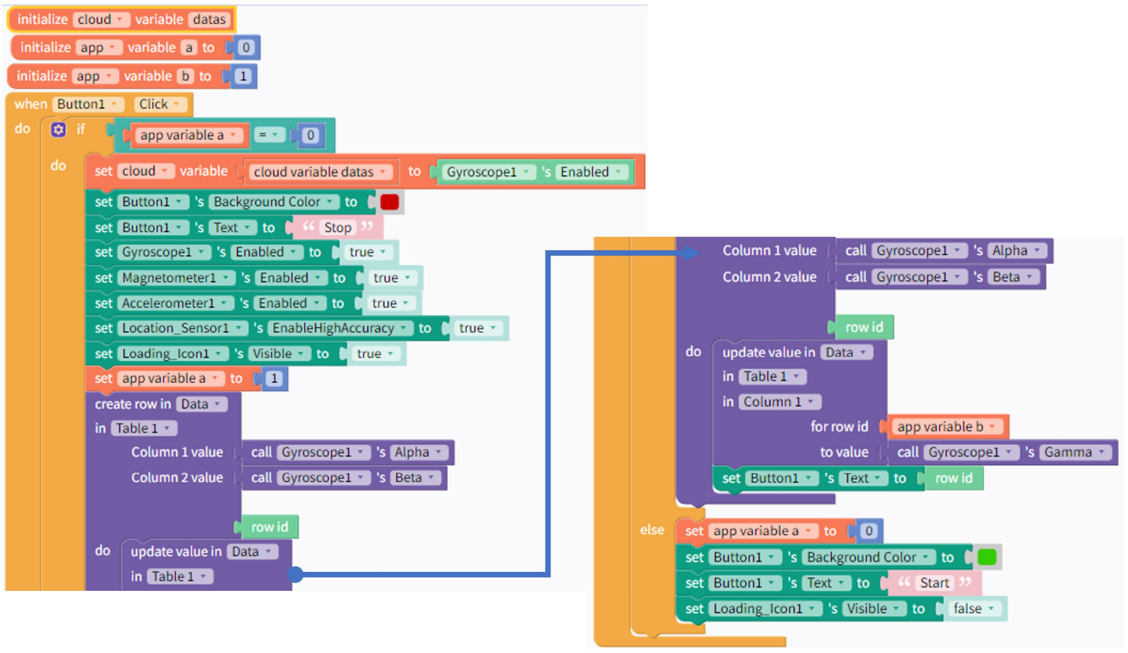


Fig. 3.7: IFDenseNET-138 Conceptual Diagram for Classifying Mobile Phone Sensor Data

The figure 3.7 shows a conceptual diagram of the IFDenseNET-138 deep learning architecture, which is proposed for classifying mobile phone sensor data. The model consists of 138 layers of dense connections, which allows for the efficient flow of information throughout the network. The input data to the model is a 2-D array, where each row represents individual sensors and each column represents a time series. The data is then processed through a 32x3 dense layer with ReLU activation for 138 times. To reduce the dimensions of the data, a max pooling layer and a flattened layer are used. Finally, the data is classified using the Softmax function, which outputs a probability distribution over the 4 possible classes.

The IFDenseNET-138 architecture has been shown to be accurate for classifying mobile phone sensor data, and it is also computationally efficient. This makes it a promising candidate for real time applications, such as activity recognition and health monitoring.

3.8 Mobile App details



```
initialize cloud variable datas
initialize app variable a to 0
initialize app variable b to 1
when Button1 Click
do  if  app variable a = 0
    do  set cloud variable cloud variable datas to Gyroscope1 's Enabled
        set Button1 's Background Color to
        set Button1 's Text to " Stop "
        set Gyroscope1 's Enabled to true
        set Magnetometer1 's Enabled to true
        set Accelerometer1 's Enabled to true
        set Location_Sensor1 's EnableHighAccuracy to true
        set Loading_Icon1 's Visible to true
        set app variable a to 1
        create row in Data
        in Table 1
            Column 1 value  call Gyroscope1 's Alpha
            Column 2 value  call Gyroscope1 's Beta
                            row id
        do  update value in Data
            in Table 1
            in Column 1
                for row id  app variable b
                to value    call Gyroscope1 's Gamma
            set Button1 's Text to row id
    else
        set app variable a to 0
        set Button1 's Background Color to
        set Button1 's Text to " Start "
        set Loading_Icon1 's Visible to false
```

Figure 3.8: The block code outlines a mobile application designed for evidence collection.

Button 1 serves as the trigger to initiate data collection from mobile sensors, which is then transmitted to a cloud variable. In situations where there is no internet connection, the app stores the data locally on the device. The interface, as depicted in Panel a, features both start and stop buttons, real time sensor readings, and a loading symbol during the data upload process. The block code includes the initialization of the cloud variable and a do-while loop that determines whether the app is in sleep mode or actively collecting sensor data. During the data collection phase, the app loads gyroscope readings into the cloud variable and stores additional data in table 1. The app variable A is set to 1 when data is actively being recorded and is set to 0 when either the stop button is pressed or data is already stored. The app continues to run in an infinite loop until the button is pressed again.

Figure 3.8 illustrates the block coding structure of a proposed mobile application designed specifically for evidence collection. Upon clicking button 1, the application initiates the process of gathering data from various sensors including the gyroscope, magnetometer, accelerometer, and location sensor. This collected data is then transmitted to a cloud variable that undergoes periodic updates. For instances, where an internet connection is unavailable, the data is stored locally on the mobile device's memory. In Figure 3.9, Panel a showcases the user interface display featuring a blue start button, indicating that the program has just been initiated and no data is currently present in either the cloud or local memory. As

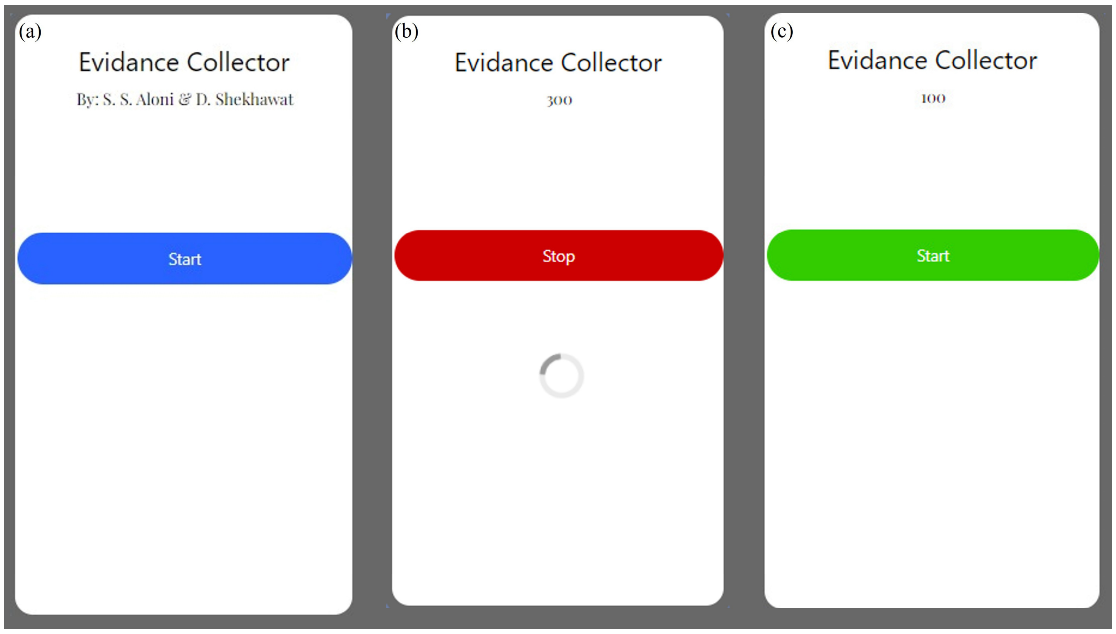


Figure 3.9: The EC mobile collection app exhibits different front-end states

(a) Initialization state: Marked by a blue button, this state indicates that no data has been collected yet. (b) Data collection state: Represented by a red button and the display of current sensor readings, this state is active when data is actively being collected. (c) Sleep mode state: Indicated by a green button, this state signifies that the data has already been stored in local memory and successfully uploaded to the cloud.

the app proceeds to collect data from all the mobile sensors, the interface switches to displaying a red stop button (Fig. 3.9 b). Additionally, the interface provides real time readings from each sensor, visible above the button. A rotating loading symbol is employed as a visual representation while the data is being uploaded to the cloud. Finally, the evidence collector app, denoted by a green start button, signifies that the data is not only stored in local memory but has also been successfully uploaded to the cloud.

Figure 3.8 depicts the block coding of the proposed evidence collector app. The red blocks represent the initialization of the cloud variable data S with two local app variables, A and B, where A is initialized to 0, and B is initialized to

1. The app has a single button called button 1 in the front end, which is initially blue (3.9a). When the user presses the button, the code goes into a do-while loop, where app variable A's value decides whether all sensor readings are being stored, or the app is in sleep mode. In sleep mode, the app variable A value is reset, and the button background color is set to green (Fig. 3.9c), the button text is set to Start, and the loading icon's visibility is set to false. If the app variable A is zero, then the 'if' loop is entered. In this loop, the cloud variable is loaded with the gyroscope's readings, and the button's background color is set to red (Fig. 3.9b), while the button's text is set to Stop. All four sensors (gyroscope, magnetometer, accelerometer, and location sensor) are enabled, and the loading icon's visibility is set to true. A new variable data is created in table 1, which holds the values of alpha, beta, and gamma.

3.9 App building using flutter

3.9.1 Creating an App in Flutter

Flutter is an open source, cross platform mobile application development frame work created by Google. It allows developers to build beautiful, high performance, and responsive applications for both iOS and Android platforms using a single codebase. In this thesis, we will explore the process of creating a simple app in Flutter.

Before diving into the app development process, it is essential to set up the Flutter development environment. This includes installing the Flutter SDK, an IDE (Integrated Development Environment) like Android Studio or Visual Studio Code, and configuring the necessary tools for the target platforms (Android and/or iOS).

3.9.2 Creating a New Flutter Project

The first step in building a Flutter app is to create a new project. This can be achieved either through the IDE's built in Flutter project creation wizard or by running the flutter create command in the terminal. During this process, one will be prompted to provide a project name, select the project type (Flutter Application), and specify the Flutter SDK path.

Once the project is created, it’s crucial to understand the project structure and the role of each file and directory. The main dart file is the entry point of the application, where the app’s execution begins. The lib directory contains the Dart source code files, including the user interface (UI) components and other application logic.

3.9.3 Building the User Interface

Flutter follows a widget based approach to building user interfaces. In the main dart file, one can find the MyApp and My Home Page widgets, which serve as the starting point for app’s UI. By modifying these widgets and adding additional widgets, one can create the desired layout and functionality for the app. In the provided code snippet, the My Home Page widget displays a Scaffold widget, which provides a basic structure for a screen in a Material Design app. It includes an AppBar at the top and a Center widget containing a Text widget that displays the message ”Welcome!”.

3.9.4 Running the App

After making the necessary modifications to the code, one can run the app on an emulator or a physical device. Flutter provides hot reload and hot restart features, which allow one to see the changes in our app instantly without having to restart the entire app.

3.9.5 Customizing the App

One can customize the app by adding more widgets, implementing navigation between different screens, integrating with APIs or databases, and incorporating various Flutter plugins and packages for additional functionality.

3.9.6 Deploying the App

Once the developer have completed the development and testing phases, one can prepare the Flutter app for release on the respective app stores (Google Play Store for Android and App Store for iOS). This process involves creating release builds, generating the necessary certificates and signing keys, and following the submission guidelines provided by the app stores. By following the guidelines outlined in this section one can gain a basic understanding of creating a simple yet functional app using the Flutter framework.

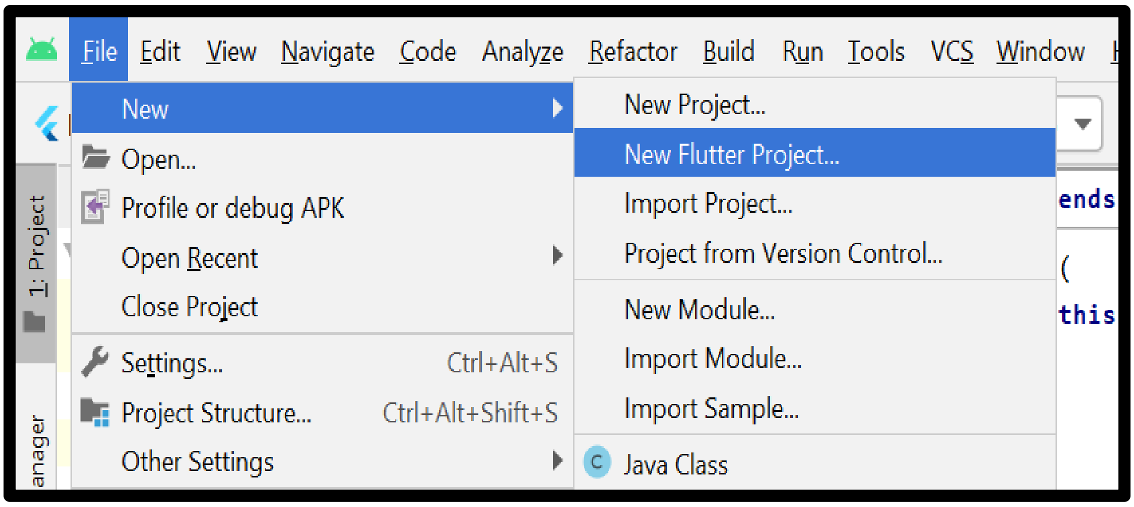
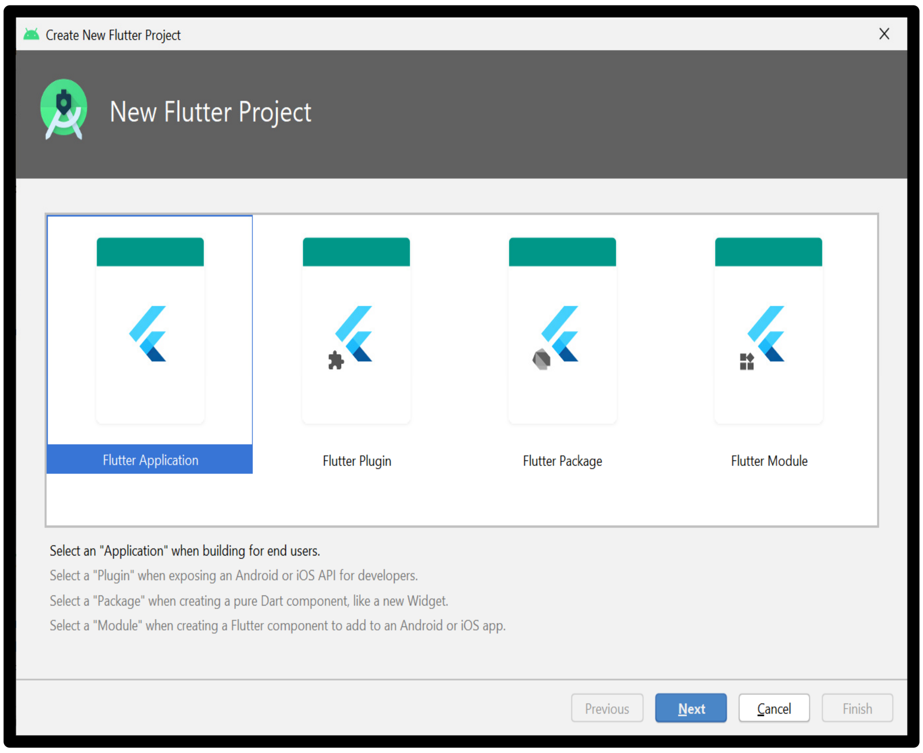


Fig. 3.10: File menu to flutter app

As shown in figure 3.10 Open the Android Studio IDE and select Start a new Flutter project.

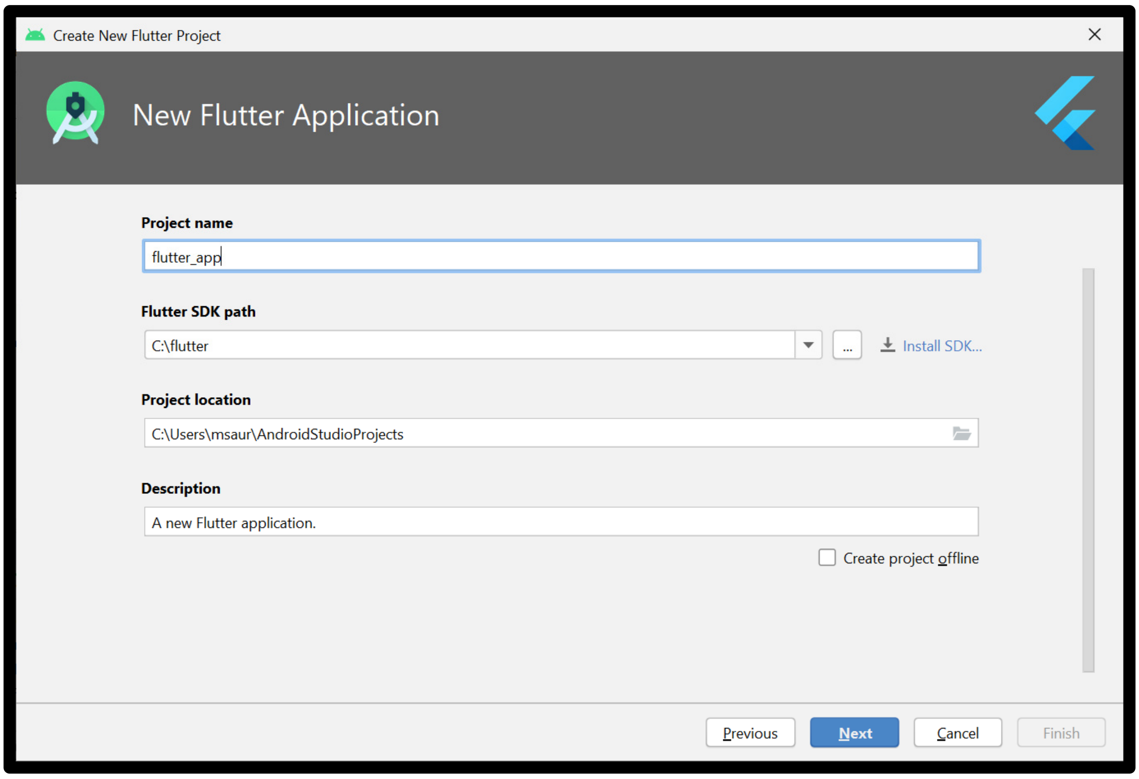


Select an "Application" when building for end users.
Select a "Plugin" when exposing an Android or iOS API for developers.
Select a "Package" when creating a pure Dart component, like a new Widget.
Select a "Module" when creating a Flutter component to add to an Android or iOS app.

Fig. 3.11: New flutter project

As shown in figure 3.11 Select the Flutter Application as the project type. Then click Next. Verify the Flutter SDK path specifies the SDK's location (select Install SDK. . . if the text field is blank).

As shown in figure 3.12 Enter a project name (for example, myapp). Then click Next.


Create New Flutter Project

New Flutter Application

Project name: flutter_app

Flutter SDK path: C:\flutter

Project location: C:\Users\msaur\AndroidStudioProjects

Description: A new Flutter application.

☐ Create project offline

Fig. 3.12: UI for new flutter application

The following code demonstrates a basic Flutter application. Flutter is an open source framework by Google for building natively compiled applications for mobile, web, and desktop from a single codebase. This example includes a simple app with a home page that displays a ”Welcome to GeeksForGeeks!” message.

In the code, the main function is the entry point of the application, calling runApp() to attach the given widget to the screen. The MyApp class extends Stateless Widget, making it a widget that does not maintain any internal state. The build method of MyApp returns a MaterialApp widget, which serves as the root of the application. It sets the title, theme, and home page of the app. The home page is defined by the My Home Page class, which also extends Stateless Widget and returns a Scaffold widget, providing a structure for the app with an app bar and centered body content.

```
// Importing important packages require to connect
//  Flutter  and  Dart
```

```
import ' package : flutter/ material. dart ';

// Main Function

void main () {

        // Giving  command  to  run App ()  to  run  the  app .

        /* The purpose of the run App () function is to attach the given

        widget to the screen .  */

        run App ( const MyApp ());

}

// Widget is used to create UI in flutter framework .

/* StatelessWidget is a widget , which does not maintain any

state of the widget. */


/* MyApp extends StatelessWidget and overrides its build

method .  */

class MyApp extends StatelessWidget {

        const MyApp ({ Key ? key }) : super( key : key );

        // This widget is the root of the application .  @

        Override

        Widget build ( Build Context context) { return

                MaterialApp (

                        //  title  of  the  application

                        title : ' Hello World Demo Application ',

                        // theme of the widget theme

                        : Theme Data (

                                primary Swatch :  Colors. lightGreen ,
                    ),
                    // Inner UI of the application
                    home : const My Home Page ( title : ' Home page '),
                );
        }
}
/* This class is similar to MyApp instead it returns
Scaffold Widget */
class My Home Page extends StatelessWidget {
        const My Home Page ({ Key ? key , required this. title }) :
                super( key : key );
        final String title ;
        @override
        Widget build ( Build Context context) {
                return Scaffold (
                    app Bar: App Bar(
                        title : Text( title ),
                ),
                // Sets the content to the
                // center of the application page body :
                const Center(
                    // Sets the content of the Application child :
                    Text(
                        ' Welcome to GeeksForGeeks!',
                    )
```

```
            ),
        );
    }
}
```

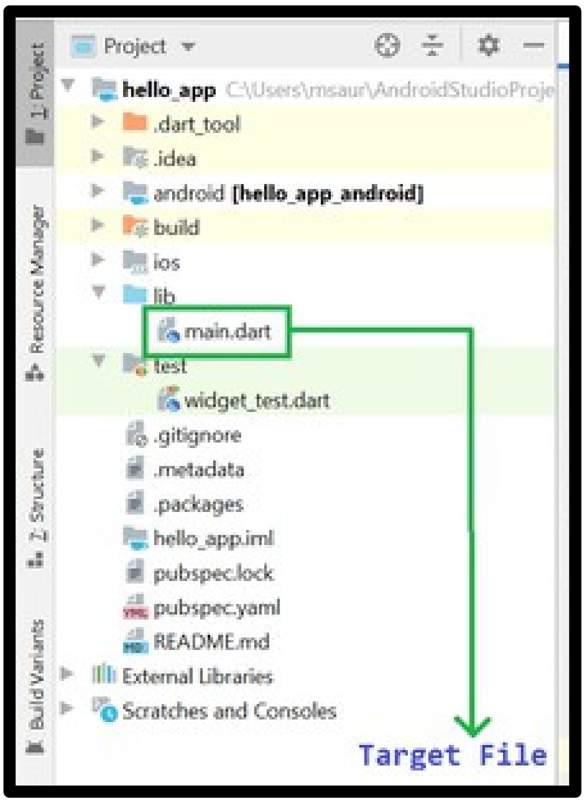


Fig. 3.13: Select main.dart as target file

Figure 3.13 shows that main.dart is the main target file to be chosen in which all the codes needs to be uploaded.

As shown in figure 3.14 Once the file is built apk file can be installed on mobile and one can see app like this. This is sample code executed from website GeeksForGeeks! that acted as hello world for us to develop fultter apps.

3.10 Classification of actual Sensor data using python code

The algorithm shown is a Python script that reads accelerometer, gyroscope, and magnetometer data from CSV files, combines the features into a single feature vector, and uses a deep learning model to classify the data into two classes (0 and 1). Here's an explanation of the code: The script starts by importing the necessary libraries: pandas for data manipulation, numpy for numerical operations, train testsplit from scikit learn for splitting the data into training and testing sets, and Sequential and Dense from TensorFlow's Keras library for building the deep learning model. Next, the accelerometer, gyroscope, and magnetometer data are read

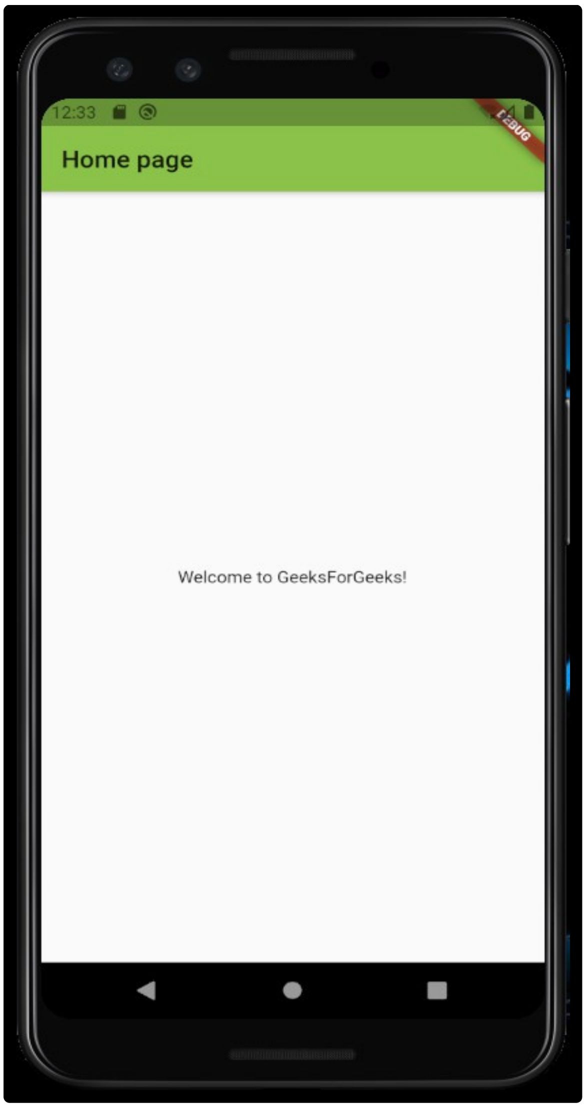


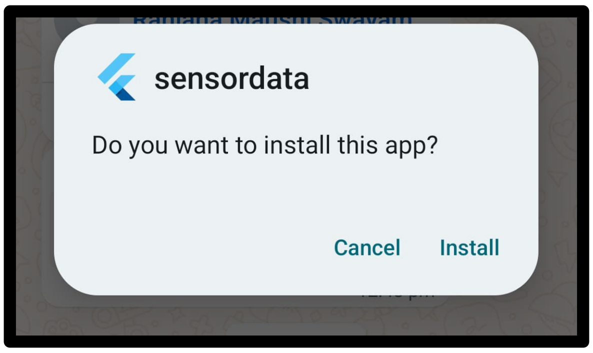
Fig. 3.14: The final app built

from their respective CSV files using the pd.read csv() function from pandas. The feature columns (columns 2 to 4) are extracted from each data frame using the iloc method and converted to NumPy arrays. To ensure that the feature vectors have the same length, the script finds the minimum length among the three feature vectors and truncates each vector to that minimum length using slicing [:minlength]. This step is necessary because NumPy's concatenate function requires arrays of the same shape along the non-concatenation axis. The truncated feature vectors are then combined into a single feature vector using np.concatenate() along axis=1, which means concatenating horizontally (column-wise). Random labels (0 or 1) are assigned to the data using np.random.randint(2, size=featurevectors.shape[0]).

In the real world scenario, one would need to provide the actual labels for the data. The combined feature vectors and labels are then split into training and testing sets using train-test-split from scikit learn. The testsize parameter is set to 0.2, which means 20% of the data will be used for testing, and the remaining 80% for training. The random state parameter ensures reproducibility of the split. A deep learning model is defined using the Sequential API from Keras. The model has four dense (fully connected) layers: the input layer with 9 units (corresponding to the 9 features), two hidden layers with 64 and 32 units, respectively, and an output layer with a single unit. The ReLU activation function is used for the hidden layers, and the sigmoid activation is used for the output layer, which is suitable for binary classification problems. The model is compiled with the binary cross-entropy loss function, the Adam optimizer, and the accuracy metric. The model is then trained on the training data using ”model.fit()” function. The epochs parameter specifies the number of times the model will be trained on the entire training dataset (50 in this case), and the batchsize parameter determines the number of samples propagated through the network at once (32 in this case). The validation data parameter is used to monitor the model's performance on the validation set (Xtest, Ytest) during training. Finally, the model's performance on the test data is evaluated using model evaluate (Xtest, Ytest), which returns the loss and accuracy values. The accuracy score is printed to the console. Note that in this example, random labels are assigned for demonstration purposes. In a real world scenario, one would need to provide the actual labels for the data. Additionally, one might want to adjust the architecture of the deep learning model and the hyperparameters (e.g., number of epochs, batch size) based on the specific use case and the performance of the model.

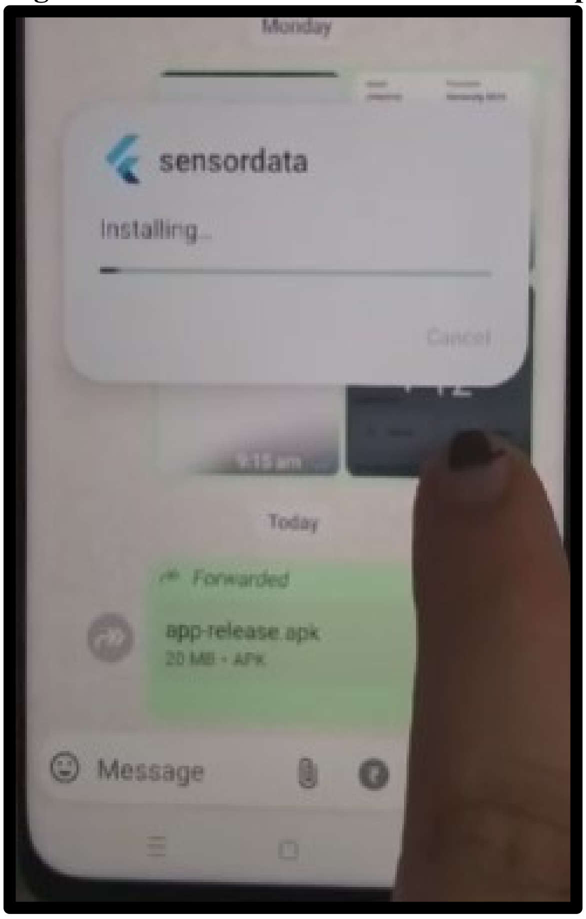
3.11 Extracting the information from the Sensor data app on android

This section of the thesis will explain in detail how to run the app and see the results of data logged. Figure 3.15 shows the basic app installation front UI where user will select install.


sensordata

Do you want to install this app?

Fig. 3.15: Installation UI for Sensordata app


sensordata

Installing...

Fig. 3.16: UI to show progress bar of installation

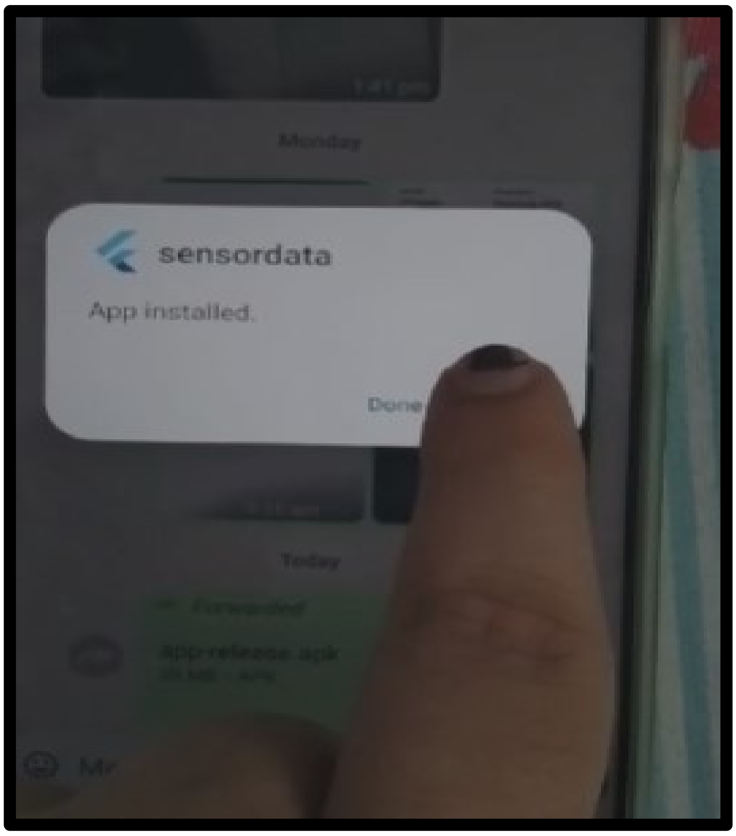


Fig. 3.17: UI at the end of installation

Figure 3.16 shows UI to show progress bar of installation. Total file size of APK is 22 MB and post install size is around 500 MB.

Figure 3.17 shows UI for the user to show completion of installation of the software. One can click on Done button to confirm installation completion.

Figure 3.18 shows sign in menu to the user, which appears only first time when user press done.

Figure 3.19 shows the UI to choose account during sign in. Here all the accounts are displayed where user has alrady signed-in through the device.

Figure 3.20 shows the App main front UI offering user with an option to save accelerometer data. Also there is option to save magnetometer and gyroscope data. All the basic setup is done and next steps will be post crime activities.

Figure 3.21 shows version button in settings sub menu about device.

Figure 3.22 shows the version number in this case 14.0.0.6 where user need to click 7 times to enter debug mode.

Figure 3.23 shows file menu to be clicked after pressing home button.

Figure 3.24 shows other storage option where user need to enter and turn on

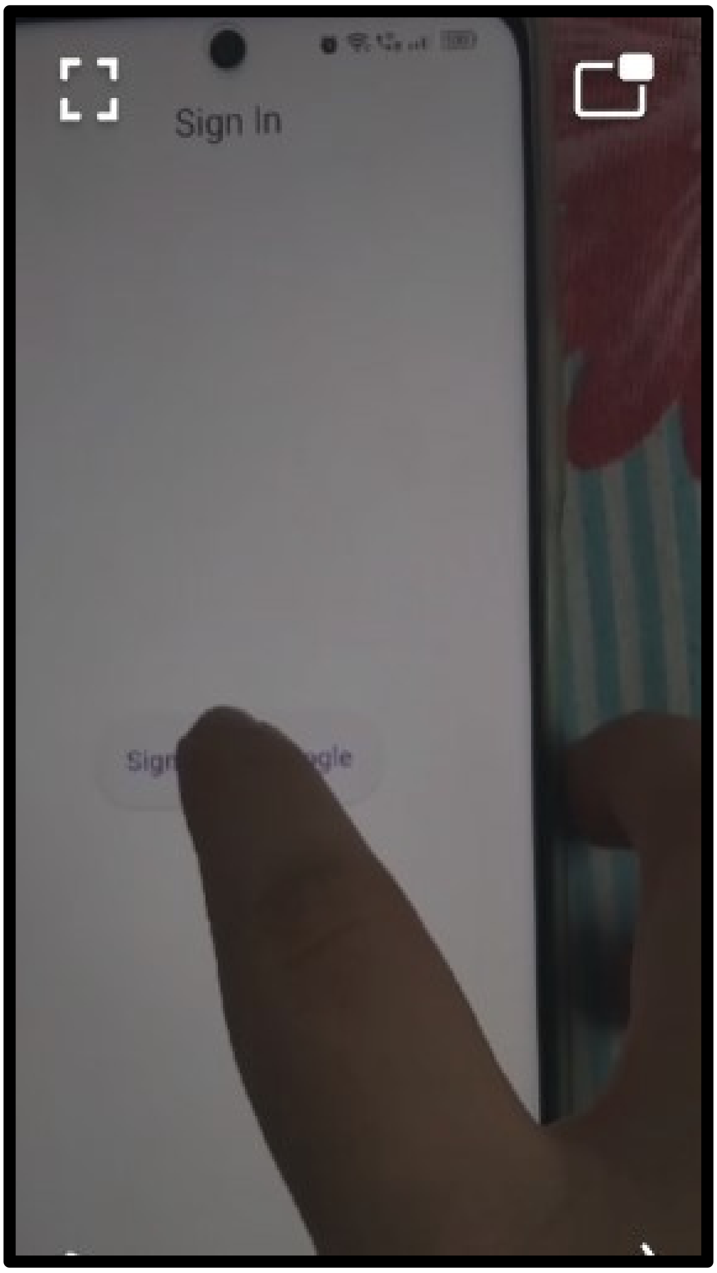


Fig. 3.18: Sign in window for app

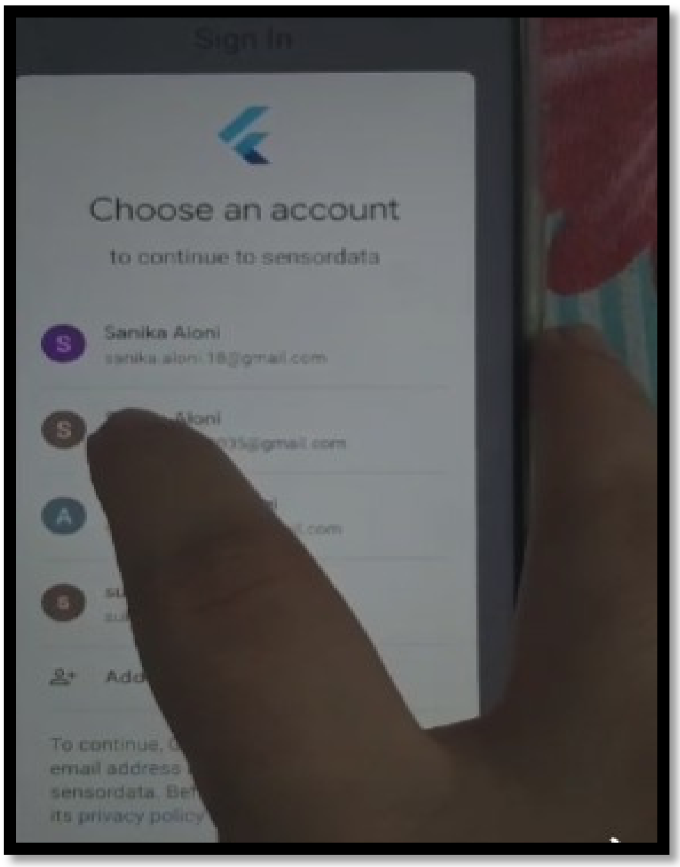


Fig. 3.19: UI to choose account during sign in

bug reports. Figure 3.25 shows menu appered in setting after turning on debug mode and one needs to click on bug report in order to see the data from sensordata app. Figure 3.26 shows new option that appears once bug report is enabled inside files of the phone.

Enter bug report by clicking newly arrived option of bug report as shown in figure 3.27

Figure 3.28 shows the files inside Oppo A79 5G phone where the testing of software was carried out. One needs to select Android from it. Figure 3.29 shows app data stored inside android folder. Along with app there are media and obb files avaiable which can store other forms of information.

Figure 3.30 shows folders inside data folder showing installed app folder ”com. example. sensordata”. this is the domain name by default for any application built on flutter. One can change this name if required.

Figure 3.31 shows files once we enter com.example.sensordata. Most of the logged data is stored inside this folder.

Figure 3.32 shows the UI when we enter Sensordata folder inside file. Only

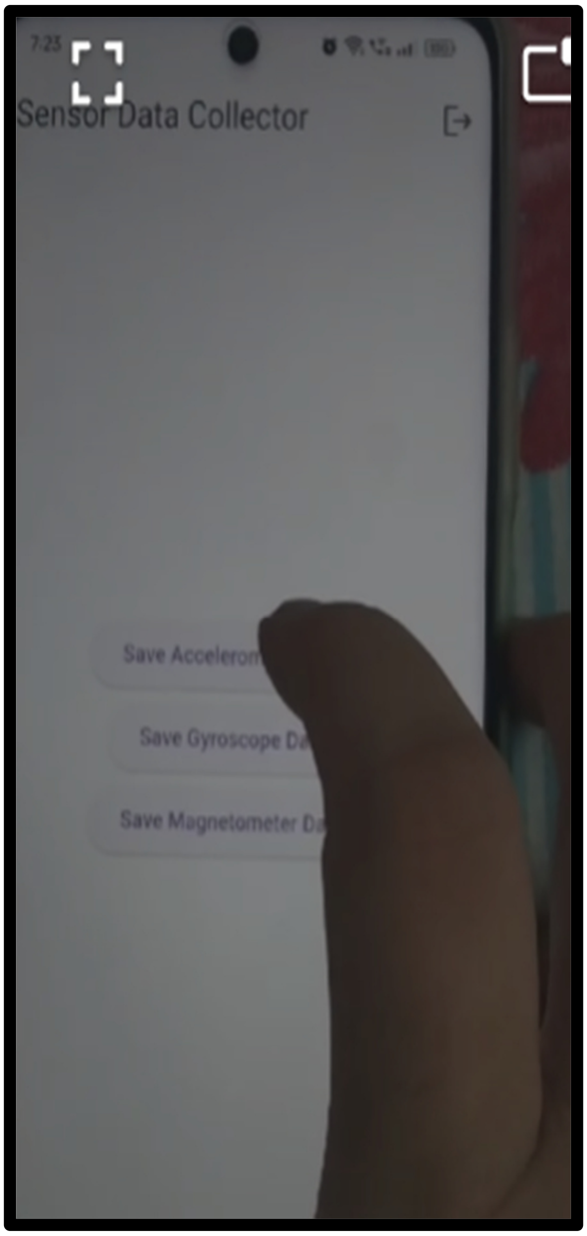
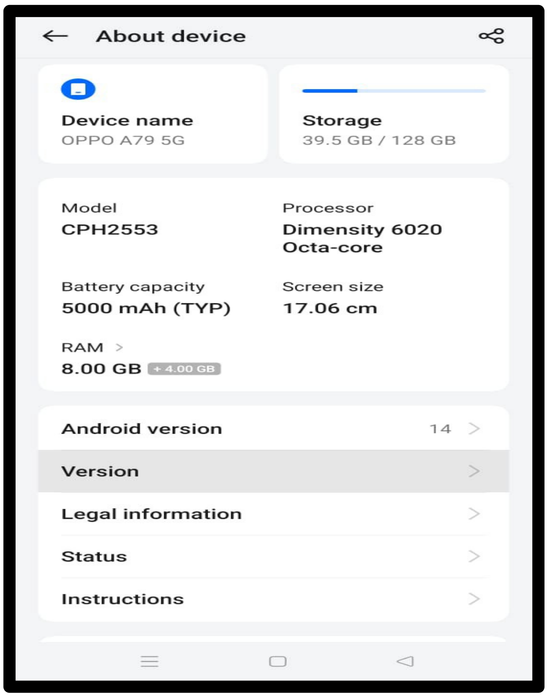


Fig. 3.20: App main front UI offering user with an option to save accelerometer data


About device

Device name
OPPO A79 5G

Storage
39.5 GB / 128 GB

Model
CPH2553

Processor
Dimensity 6020 Octa-core

Battery capacity
5000 mAh (TYP)

Screen size
17.06 cm

RAM >
8.00 GB + 4.00 GB

Android version 14 >

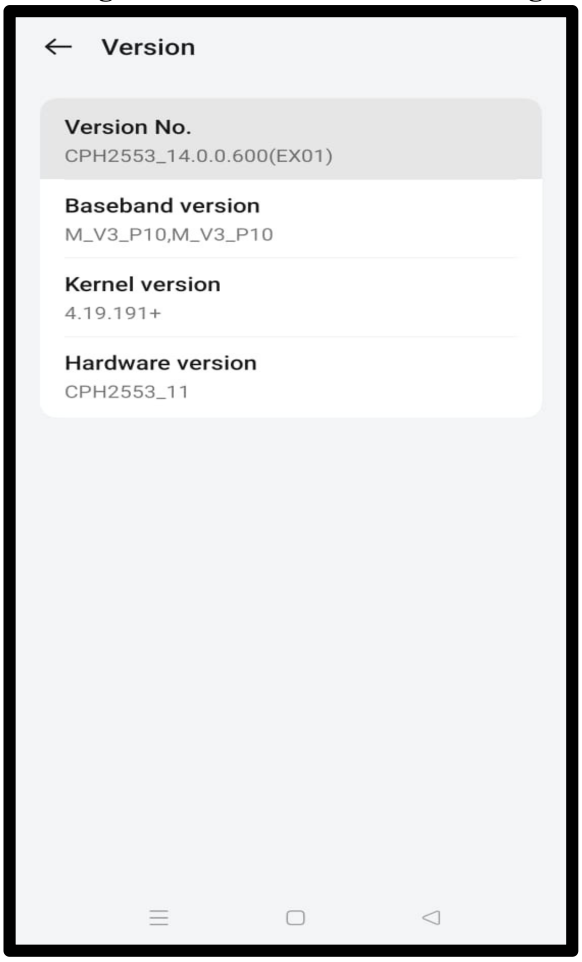
Version >

Legal information >

Status >

Instructions >

Fig. 3.21: Version button in setting


Version

Version No.
CPH2553_14.0.0.600(EX01)

Baseband version
M_V3_P10,M_V3_P10

Kernel version
4.19.191+

Hardware version
CPH2553_11

Fig. 3.22: Version button from android device

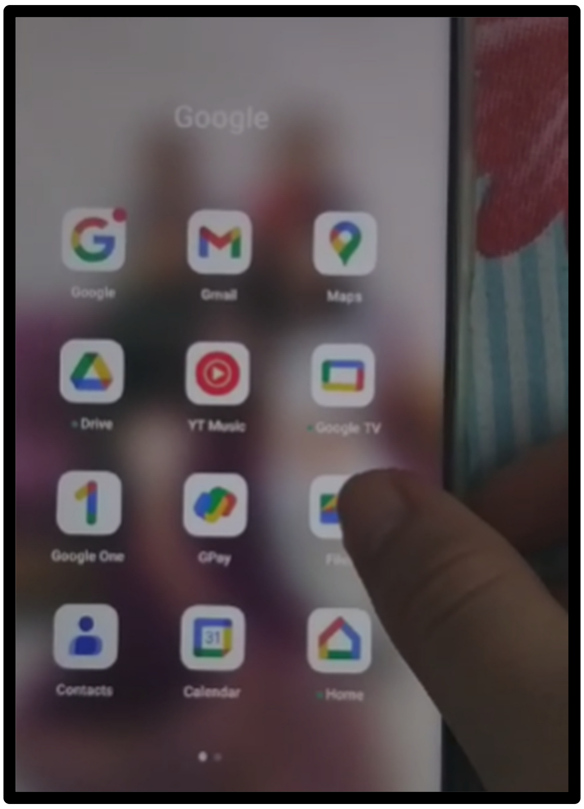


Fig. 3.23: File menu to recover the data

one folder named sensordata will be there. Figure shows different files under Sensordata app folder inside com.example.sensordata folder. Three .csv files are produced that holds all the data required. accerlometer.csv, gyroscope.csv and magnetometer.csv file.

All the values are simultaneously written inside the 3 files. One of the files selected called gyroscope.csv is as shown in figure 3.34. Figure 3.35 shows that the data stored in 4 columns one is sensor name and next 3 are the values from these sensors. For simple sensor like light, only two values per row will be present; and for complex sensors like accelrometer, atleast 3 values will be present.

3.12 Mobile Application for Sensor Data Collection

This Flutter application, named SensorData, is designed to collect and store data from various mobile device sensors. The app begins with a login interface, allowing users to authenticate via username and password or through Google OAuth

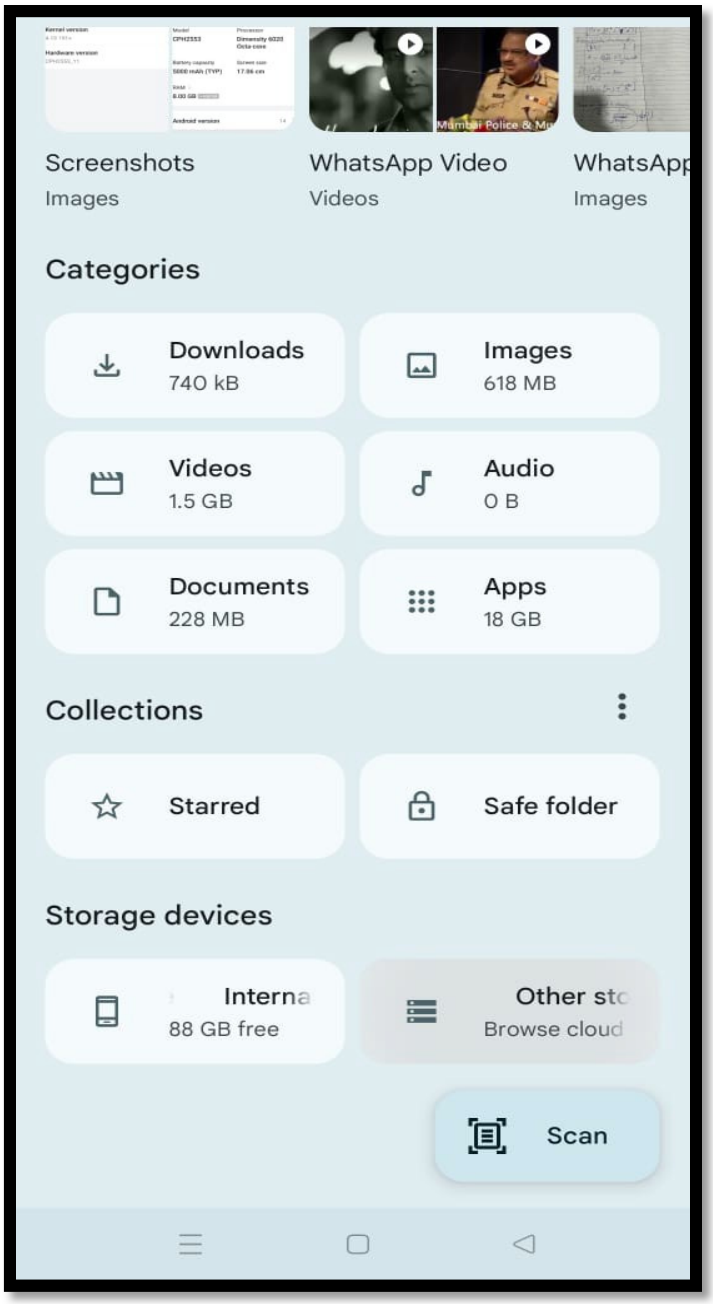


Fig. 3.24: Storage device setting to turn on bug reports

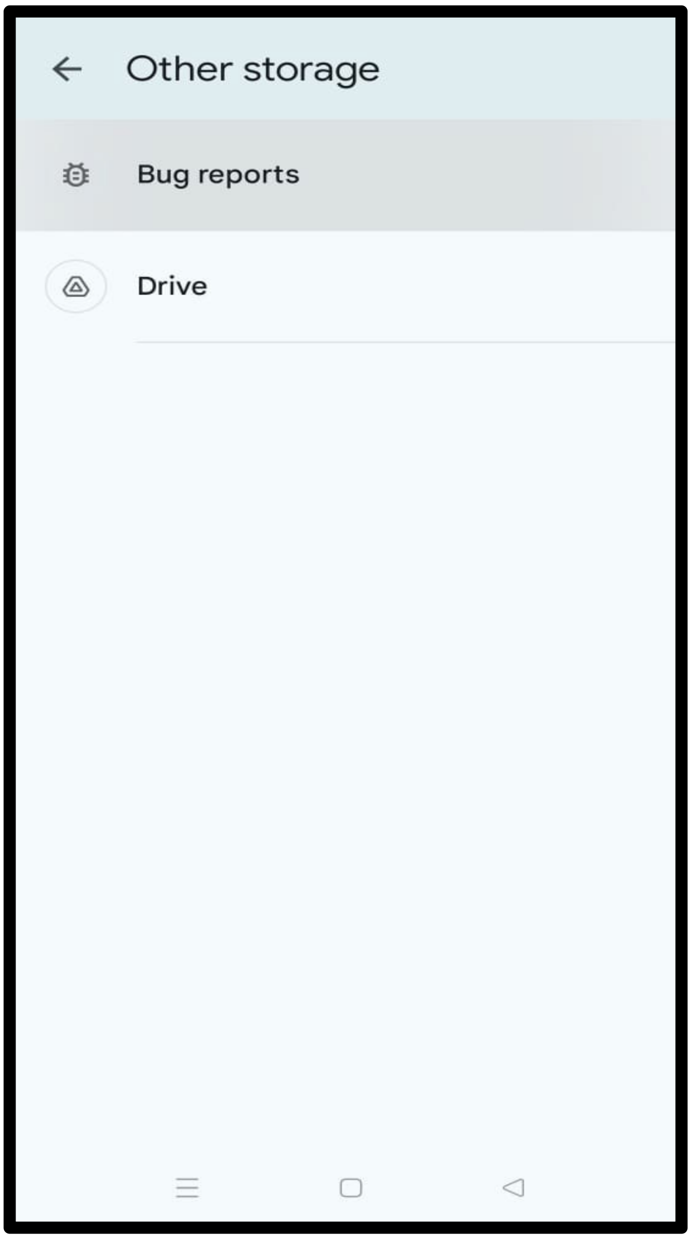


Figure 3.25: Setting to turn on Bug Report

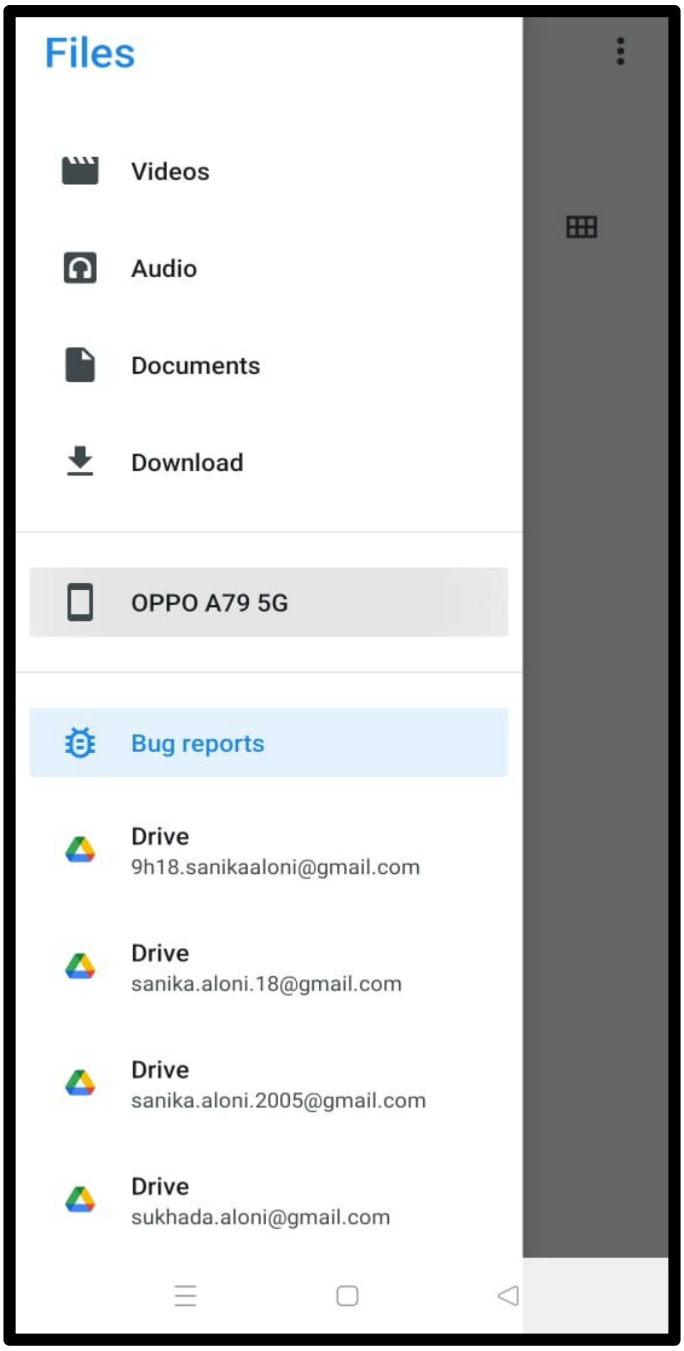


Fig. 3.26: Enabled Bug report in file

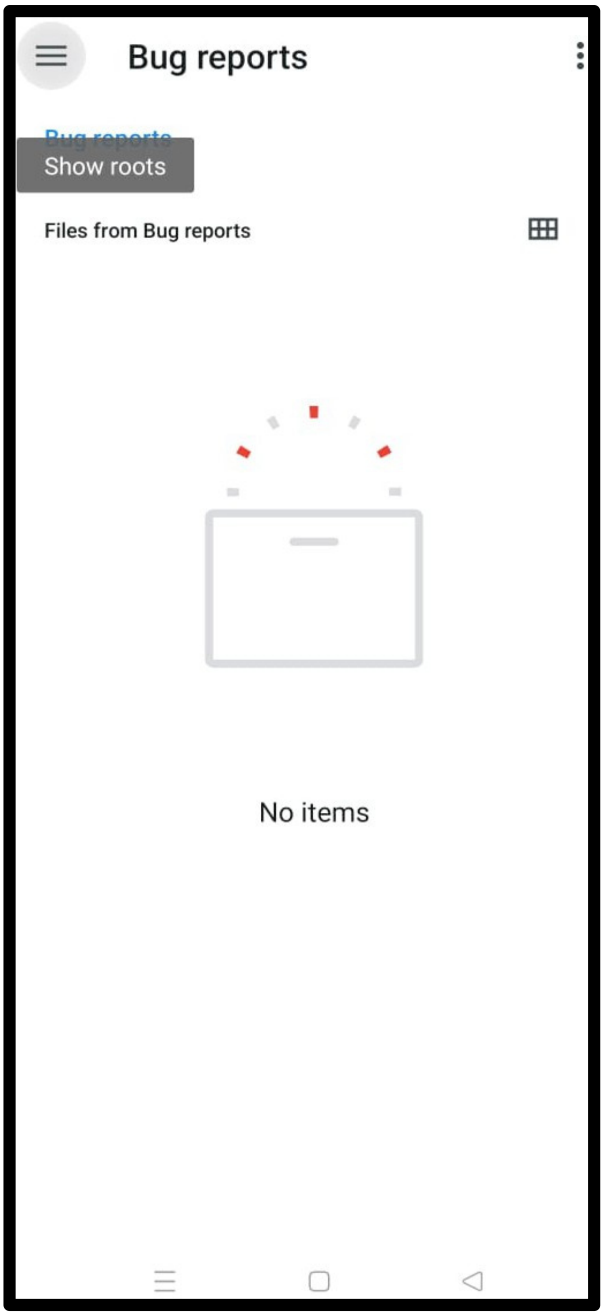


Fig. 3.27: Bug report folder contents

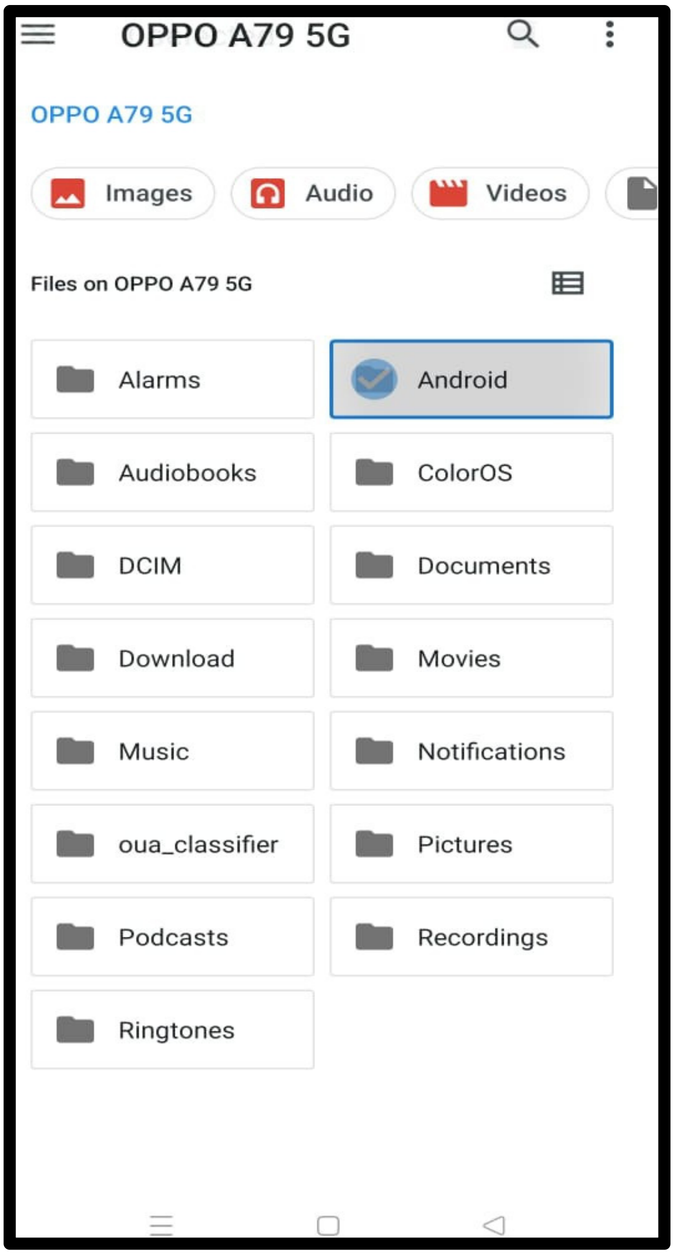


Fig. 3.28: Android folder to be opened inside the device Oppo A79 5G in this case

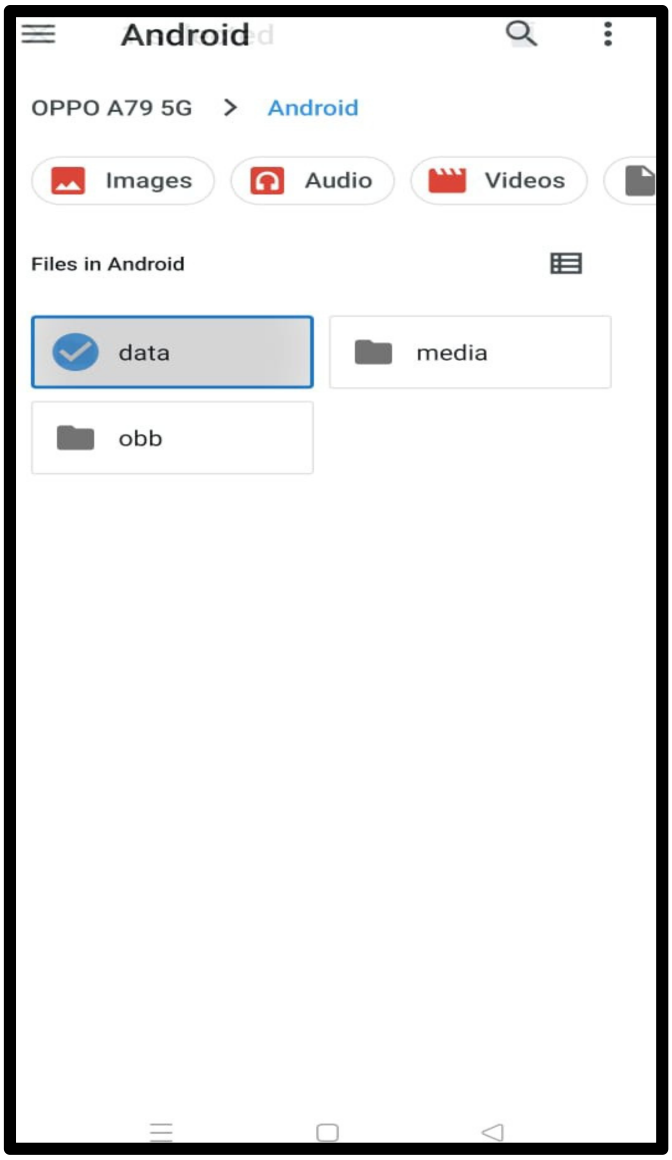
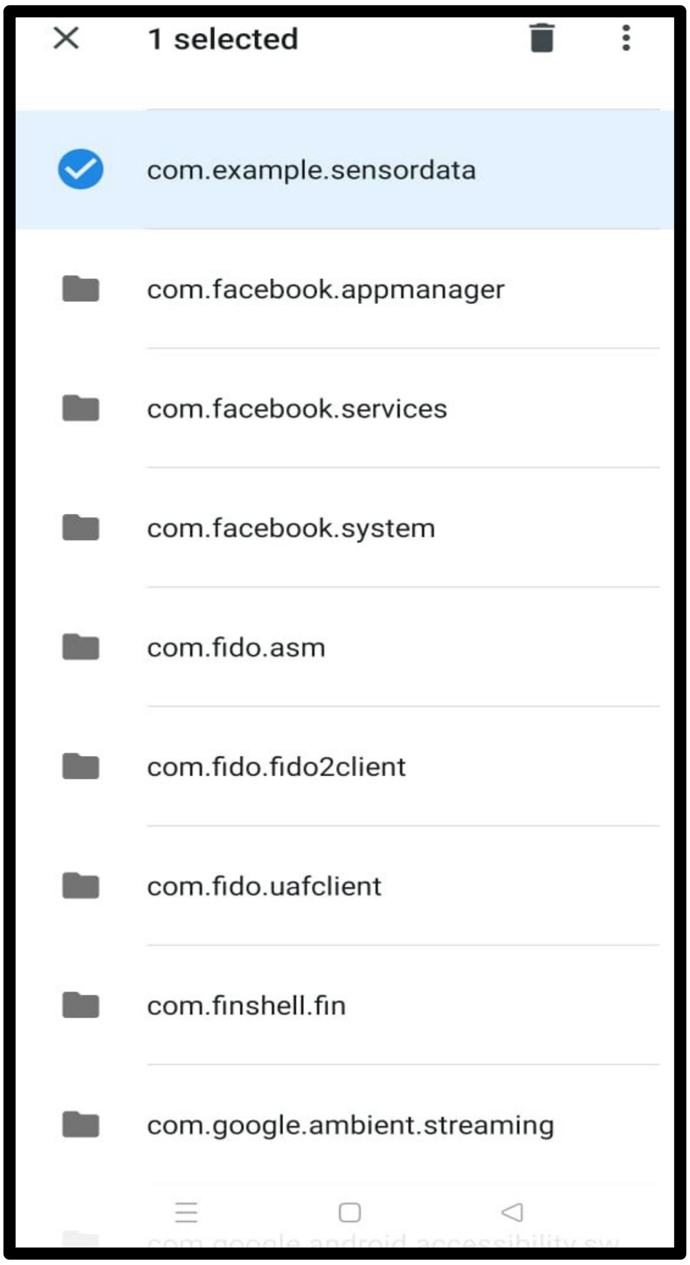


Fig. 3.29: Data folder inside the android folder where app data is stored.



1 selected

com.example.sensordata
com.facebook.appmanager
com.facebook.services
com.facebook.system
com.fido.asm
com.fido.fido2client
com.fido.uafclient
com.finshell.fin
com.google.ambient.streaming

Fig. 3.30 : Folders inside data folder showing installed app folder com. example. sensordata

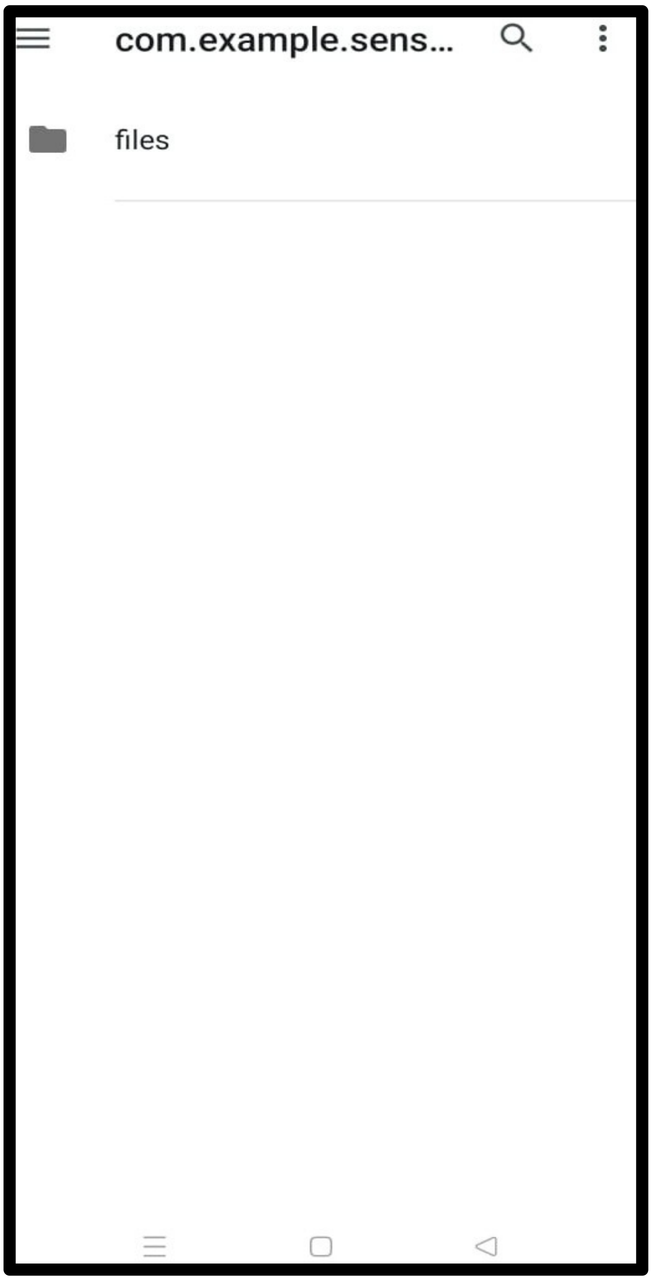


Fig. 3.31: Files inside com.example.sensordata

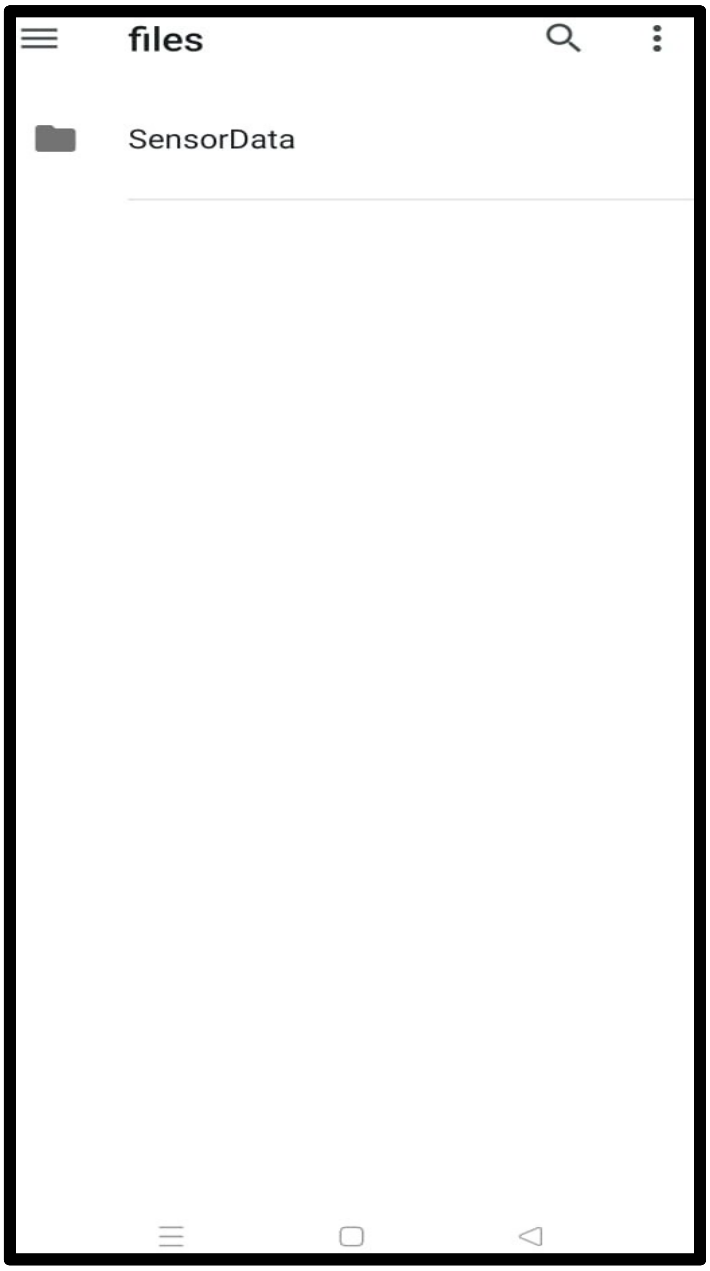
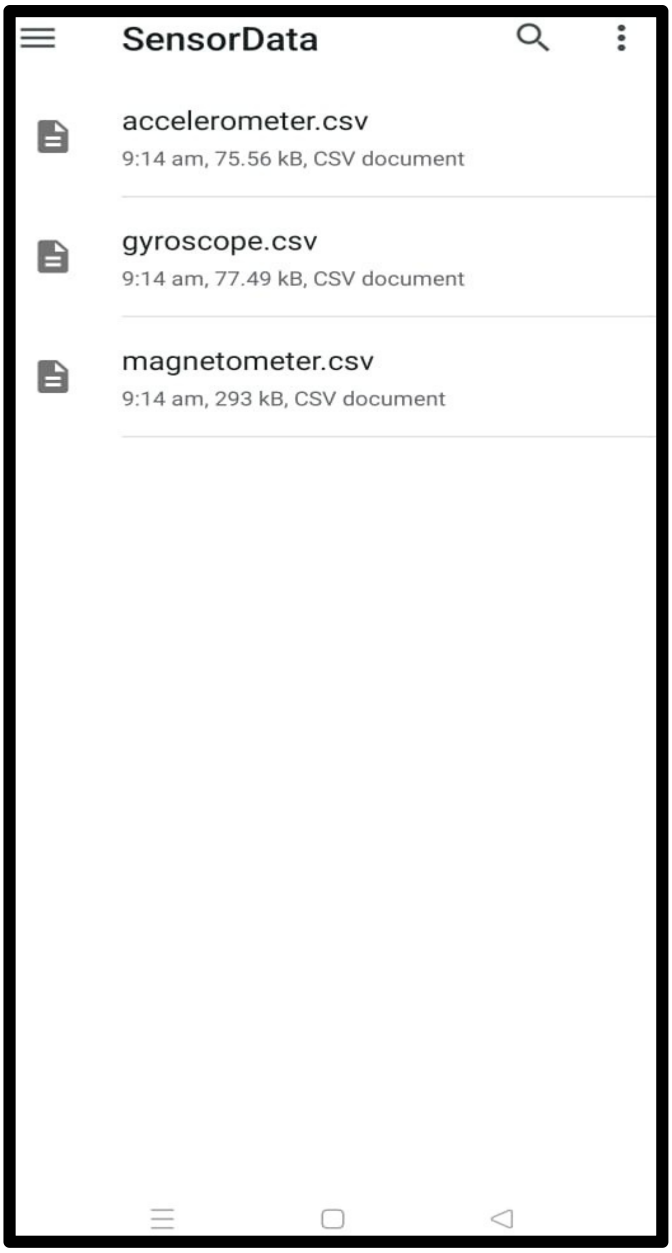


Fig. 3.32: Sensordata folder inside files

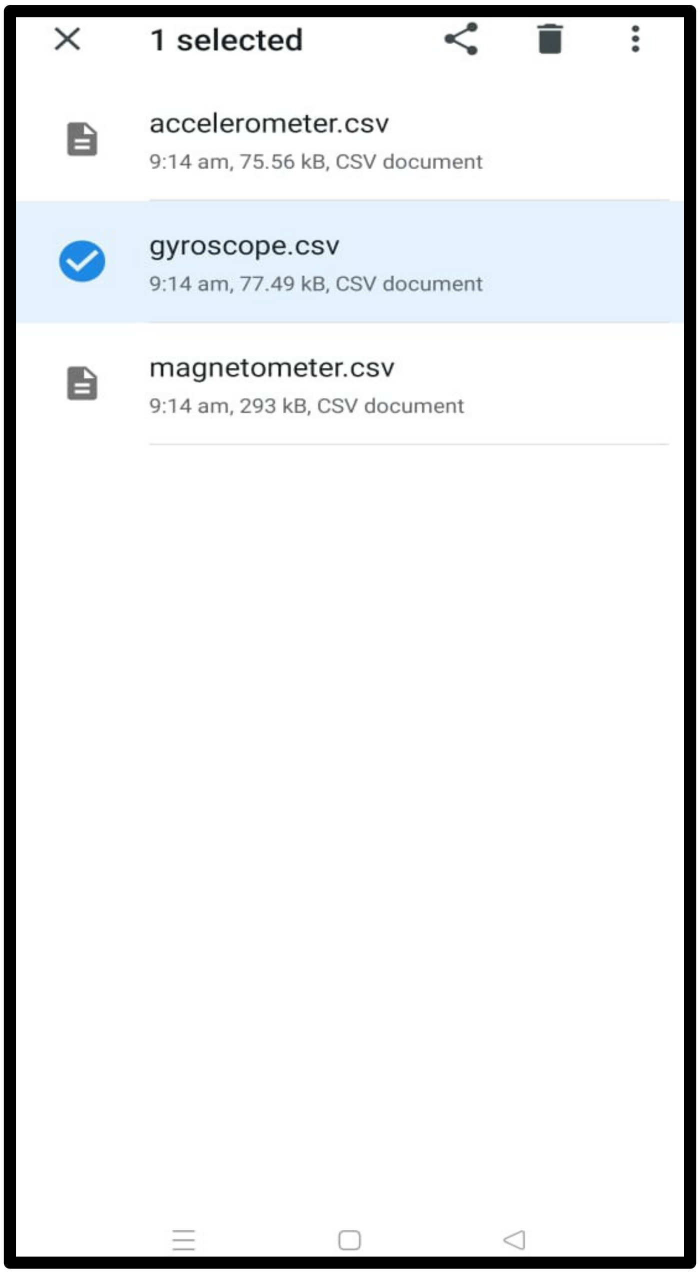

SensorData

accelerometer.csv
9:14 am, 75.56 kB, CSV document

gyroscope.csv
9:14 am, 77.49 kB, CSV document

magnetometer.csv
9:14 am, 293 kB, CSV document

Fig. 3.33: Files under sensor data

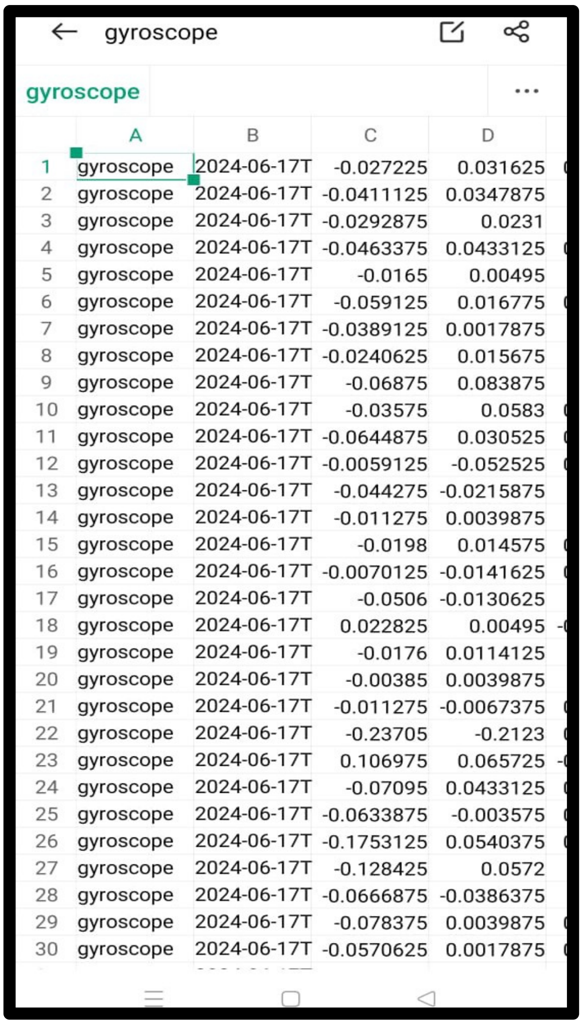


accelerometer.csv
9:14 am, 75.56 kB, CSV document

gyroscope.csv
9:14 am, 77.49 kB, CSV document

magnetometer.csv
9:14 am, 293 kB, CSV document

Fig. 3.34: Sample file selection : Gyroscope.csv



	A	B	C	D
1	gyroscope	2024-06-17T	-0.027225	0.031625
2	gyroscope	2024-06-17T	-0.0411125	0.0347875
3	gyroscope	2024-06-17T	-0.0292875	0.0231
4	gyroscope	2024-06-17T	-0.0463375	0.0433125
5	gyroscope	2024-06-17T	-0.0165	0.00495
6	gyroscope	2024-06-17T	-0.059125	0.016775
7	gyroscope	2024-06-17T	-0.0389125	0.0017875
8	gyroscope	2024-06-17T	-0.0240625	0.015675
9	gyroscope	2024-06-17T	-0.06875	0.083875
10	gyroscope	2024-06-17T	-0.03575	0.0583
11	gyroscope	2024-06-17T	-0.0644875	0.030525
12	gyroscope	2024-06-17T	-0.0059125	-0.052525
13	gyroscope	2024-06-17T	-0.044275	-0.0215875
14	gyroscope	2024-06-17T	-0.011275	0.0039875
15	gyroscope	2024-06-17T	-0.0198	0.014575
16	gyroscope	2024-06-17T	-0.0070125	-0.0141625
17	gyroscope	2024-06-17T	-0.0506	-0.0130625
18	gyroscope	2024-06-17T	0.022825	0.00495
19	gyroscope	2024-06-17T	-0.0176	0.0114125
20	gyroscope	2024-06-17T	-0.00385	0.0039875
21	gyroscope	2024-06-17T	-0.011275	-0.0067375
22	gyroscope	2024-06-17T	-0.23705	-0.2123
23	gyroscope	2024-06-17T	0.106975	0.065725
24	gyroscope	2024-06-17T	-0.07095	0.0433125
25	gyroscope	2024-06-17T	-0.0633875	-0.003575
26	gyroscope	2024-06-17T	-0.1753125	0.0540375
27	gyroscope	2024-06-17T	-0.128425	0.0572
28	gyroscope	2024-06-17T	-0.0666875	-0.0386375
29	gyroscope	2024-06-17T	-0.078375	0.0039875
30	gyroscope	2024-06-17T	-0.0570625	0.0017875

Fig. 3.35: Sample data stored for processing inside csv file

(Authentication platform of google). Once logged in, users are directed to the main functionality of the app: continuous sensor data collection. The core of the application lies in the HomePage widget, which initializes and manages the data collection process. It utilizes the sensors plus package to access the device’s accelerometer, gyroscope, and magnetometer. The app continuously listens to events from these sensors, capturing data points at regular intervals. Each data point includes the sensor type, timestamp, and X, Y, Z coordinates. This real time data is temporarily stored in memory as lists of strings. To ensure data persistence, the application implements a periodic save mechanism. Every second, the collected sensor data is written to CSV files, with separate files for each sensor type (accelerometer.csv, gyroscope.csv, magnetometer.csv). These files are stored in a dedicated ”Sensor Data” folder within the app’s external storage directory. The app’s interface provides buttons for manual data saving, while also featuring an automatic logout option for security. This structure allows for efficient data collection and storage, making it suitable for various research and analysis purposes in mobile sensing applications.

3.13 Firebase Configuration for Cross-Platform Development

This Dart code defines a crucial configuration class, Default Fire base Options, which manages Firebase setup across multiple platforms in a Flutter application. The class utilizes conditional logic to determine the appropriate Firebase configuration based on the target platform, supporting web, Android, iOS, macOS, and Windows. This approach ensures that the application can seamlessly integrate with Firebase services regardless of the deployment platform.

The current Platform getter method serves as the core of this configuration system. It checks whether the application is running on the web using the kIsWeb constant, and if not, it uses default Target Platform to identify the specific operating system. Based on this determination, it returns the corresponding platform specific Firebase Options object. This method provides a convenient way to initialize Firebase with the correct configuration by simply calling ”Default Firebase Options current Platform”.

Each supported platform has its own Firebase Options constant, containing essential parameters such as apiKey, appId, projectId, and storage Bucket. These constants also include platform specific details like androidClientId and iosClientId for mobile platforms, and measurementId for web and Windows. The code throws Unsupported Error for Linux and any unrecognized platforms, ensuring that developers are aware of unsupported environments. This structured approach to Firebase configuration facilitates efficient cross platform development and deployment of Flutter applications with Firebase integration.

3.14 Supabase Configuration

The SupabaseCread class encapsulates the essential credentials required for connecting to a Supabase backend. It contains two final string properties: url and key. The url property stores the unique URL of the Supabase project, which in this case is ’https://xnlvzblwtolrtfyrvgta.supabase.co’. The key property holds the project’s API key, a long JWT token that provides authentication and authorization for accessing the Supabase services. This configuration allows the application to establish a secure connection with the Supabase backend, enabling data operations and other backend functionalities. By encapsulating these credentials in a separate class, the code promotes better organization and easier management of backend configuration details.

3.15 Deep learning code for data classification

The code begins by importing necessary libraries and defining paths to CSV files containing sensor data (accelerometer, gyroscope, and magnetometer). It then loads these CSV files into Pandas DataFrames and combines them into a single DataFrame. The ’sensorType’ column is encoded into numerical classes using Label Encoder. This preprocessing step is crucial for preparing the data for machine learning algorithms, as it converts categorical data into a format that can be easily processed by neural networks.

The data is then split into features (X) and labels (y), where X contains the sensor type and x, y, z coordinates, and y contains the encoded sensor types. The dataset is further divided into training and testing sets using train test split from scikit learn. A Sequential model is defined using Keras, consisting of two Dense layers with 64 neurons each, using ReLU activation functions, and Dropout layers to prevent overfitting. The output layer has 10 neurons with a softmax activation function, suitable for multiclass classification.

The model is compiled using the Adam optimizer and sparse categorical cross-entropy loss function. An Early Stopping callback is implemented to prevent over-fitting by monitoring the validation loss. The model is then trained on the training data for 20 epochs with a batch size of 32, using the test data for validation. After training, the model’s performance is evaluated on the test set, and the accuracy is printed. Finally, the code demonstrates how to use the trained model for making predictions on new data by providing an example data point and printing the predicted class.