IoTSSC Project – CW3

Cloud-powered Wearable Fitness Tracking

Joao Maio
s1621503

ABSTRACT

This project implements a robust, scalable data processing pipeline for use with accelerometer data originating from an IoT device. The device is set up to be managed using the Pelion IoT platform, and relay its sensor readings at a frequency of 5 Hz using the LwM2M protocol. Sensor data is then automatically forwarded to Google Cloud Platform, where it is ingested, aggregated, and analyzed using Cloud Functions and a streaming Dataflow pipeline to produce live activity predictions for fitness tracking, which are displayed on a simple React frontend.

1 BACKGROUND

The fairly recent rise in popularity of Internet of Things devices has seen them being used in new and innovative mainstream consumer applications. Within the realm of IoT, wearable technology faces a unique set of challenges, such as needing to be small and light enough to be worn for prolonged periods of time while constantly having its radios on, recording high-frequency data from multiple sensors, and simultaneously needing to be power-efficient to remain powered for multiple hours per day (and potentially even for multiple days at a time).

For human activity recognition, a successful wearable device will need to strike a balance between the characteristics above, with the most relevant sensors being inertial sensors like an accelerometer and gyroscope for encoding a user's perceived movement into discrete data points. Since most of the energy in these acceleration signals is below 15 Hz, [15] it is possible to achieve reasonable prediction results without sampling at excessively high frequencies.

As a starting point, the recommended "Run or Walk" dataset [8] was used to pre-train a machine learning model for activity recognition. Since this dataset uses a sampling rate of 5 Hz, the system was initially set up around it to guarantee a minimum viable product. The available activity classes would initially only be *Walking* and *Running*, but would eventually be extended with a third class for the *Resting* activity collected using a similar method to that of the original author.

From there, the board and the rest of the system were set up to collect data at 5 Hz, with good results. Different data representations were tested as the primary output from the board, including all three accelerometer axes, all three gyro axes, and a single scalar value representing the (squared) magnitude of the acceleration vector. After considering these representations, and given the limited number of activity classes and their relative orientation invariance, the magnitude-only representation was chosen as the data source, with the added benefit of being a more compact scalar representation instead of a three-dimensional vector for data transmission.

2 END-TO-END PROTOTYPE

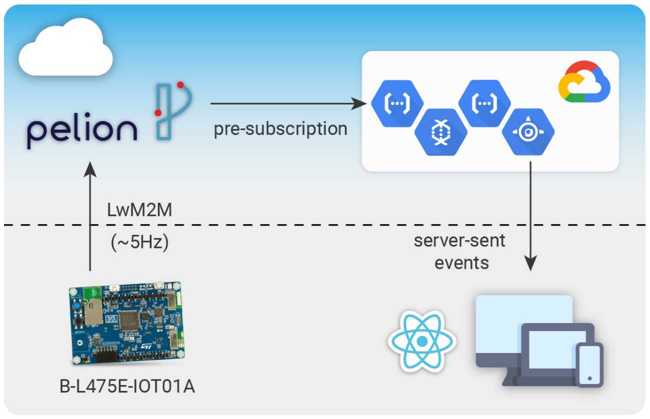


Figure 1: An overview of the project's end-to-end pipeline.

2.1 Embedded

The hardware used is an *STMicroelectronics B-L475E-IOT01A Discovery kit for IoT node.* This board contains multiple sensors, most importantly a 3D accelerometer and 3D gyroscope, as well as fast connectivity in the form of 802.11n Wi-Fi (b/g also supported).

Setting up the embedded firmware involved gathering snippets from and modifying publicly available example projects to bring together the desired functionality. Despite some of these projects being deprecated, [14] they mostly seemed to point to an active project (`mbed-os-example-pelion` [2]) which is compatible with current software versions, including Mbed Studio. This example project offers a basic management system for connecting to the Pelion IoT platform over Wi-Fi, and creates a bi-directional communication channel for data transfer between the board and the cloud.

With this, the board's sensors were first configured to be read periodically at a reduced frequency of 1 Hz and to publish their readings directly to Pelion by declaring them as observable resources. [12] The accelerometer had each of its axes published to the `3313/0/X` URI, `X` being the resource ID, where these follow the definition of the Open Mobile Alliance (OMA LightweightM2M). [1] Similarly, the gyroscope was also set up to publish each of its axes to the `3334/0/X` URI in a similar way. To transmit the calculated acceleration vector magnitude, a similar resource was set up to effectively "simulate" a single-axis accelerometer. This virtual device was set up with the `3313/1/5702` URI, specifying that it is an accelerometer with ID 1 – the real accelerometer has ID 0 – and that it outputs data to a single axis – x – which is a composite axis resulting from calculating the magnitude of the acceleration vector provided by the real accelerometer.

During testing, an issue was encountered with publishing floating-point values to Pelion directly. This was traced to older versions of Mbed OS (<6.0) having floating-point printing disabled by default, [3] which made debugging more challenging. A workaround for this was to simply multiply the resulting values by a factor of 1000 (which is exactly how the accelerometer sensor provides its data without using floating-point numbers) and then dividing by 1000 later on in the pipeline.

Although reporting of data worked correctly at 1 Hz, the Pelion docs state that when publishing data from observable resources, "calling this API rapidly (< 1s) can fill up the CoAP resending queue and notification sending fails." [13] Clearly, choosing to send raw data at 5 Hz would not be the best approach, but despite high-frequency data transfer being discouraged, it was tested successfully in mostly ideal conditions (strong Wi-Fi signal, reliable internet connection). Sending aggregated data, such as in groups of five to ten data points every few seconds, was considered the most sensible alternative, but was not implemented as it would require additional firmware logic, creating encoding and decoding schemes, and would be harder to debug on Pelion as it would need to be transmitted as an "opaque" value.

2.2 Pelion device management

After the raw sensor data is successfully published to Pelion, it must be forwarded for further processing. Using the Pelion Connect and Notifications APIs, [11] a webhook-based data path was set up to transfer incoming data to another web resource capable of ingesting and processing that data. Preliminary test runs were carried out using webhook.site to ensure that the data path had been configured correctly and both systems could handle the proposed data frequency of 5 Hz sustainably. These test runs validated that both systems could indeed cope with the high data frequency.

The final configuration makes use of pre-subscriptions, which instructs Pelion to dynamically manage subscriptions according to predefined rules. To achieve this, the relevant resources must be specified for pre-subscription using the Connect API – in this case, Pelion has been set up to automatically subscribe to the `/3313/1/5702` URI for all devices. For transmitting data from all accelerometer axes, this could be set up using a wildcard (`/3313/0/*`), which Pelion then aggregates into a single response containing all the axis values. When a new device is registered, or when a registered device is powered on, Pelion begins to forward incoming data from that resource to all the available callback URLs – these URLs must be registered beforehand using the Notifications API. With this architecture in place, no additional steps are required by the developer to activate data forwarding, as it will be fully automated and managed by Pelion.

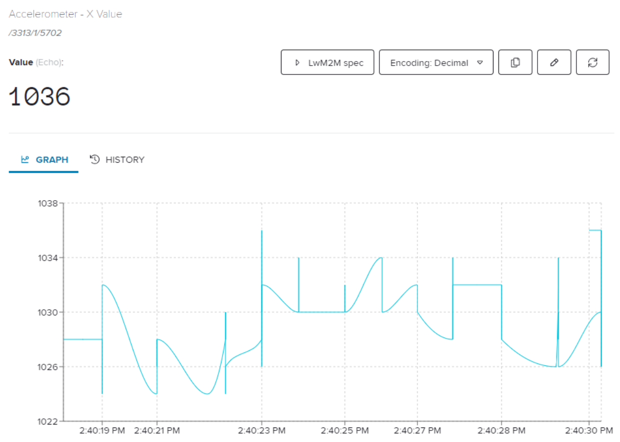


Figure 2: Pelion device management: live resource observation.

2.3 Google Cloud Platform ↔ Pelion interface

Having automated data forwarding on Pelion, the process of ingesting that data needed to be mirrored on Google Cloud Platform. Google Cloud Platform (GCP) provides hundreds of cloud services for specific use cases, encouraging the division of what would normally be a single, monolithic application into smaller, self-contained *services*.

The full data pipeline is explained in section 2.4, but the first step requires creating a link between Pelion and GCP. As before, Pelion can be set up with callback URLs, and a quick search revealed that GCP's Cloud Functions service provides serverless "functions as a service (FaaS)" which can be triggered by “third-party services that offer webhook integrations” [4] such as Pelion.

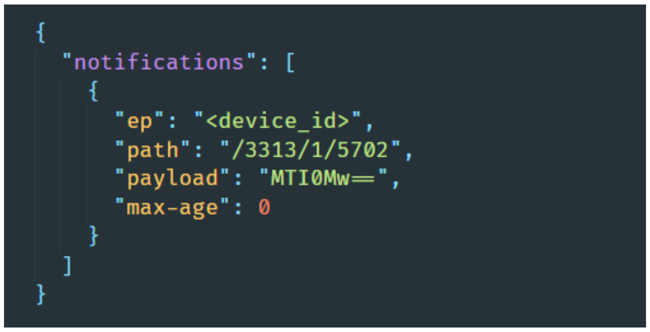
As an interesting aside, Pelion itself is described as being cloud native: it is “deployable on any public cloud platform such as AWS, Azure and Google” [10], so is likely to be using a service simialar to GCP Cloud Functions or AWS Lambda for generating webhooks behind the scenes.

2.4 Google Cloud Platform data pipeline

A GCP pipeline has been set up with four main services to support the required functionality (see section 2.4). Between each of these services resides a Pub/Sub topic, meaning that any service could hook into the data generated at any stage of the pipeline to extend the project and create additional functionality in a straightforward manner.

2.4.1 *Ingest.*

The first step requires ingesting the data forwarded by Pelion. This is implemented as a Python-based Cloud Function that simply reads the incoming Pelion webhook, parses its contents from a JSON string to a Python object, then indexes into the relevant base64-encoded notification payload (see fig. 3).



```
{
  "notifications": [
    {
      "ep": "<device_id>",
      "path": "/3313/1/5702",
      "payload": "MTI0Mw==",
      "max-age": 0
    }
  ]
}
```

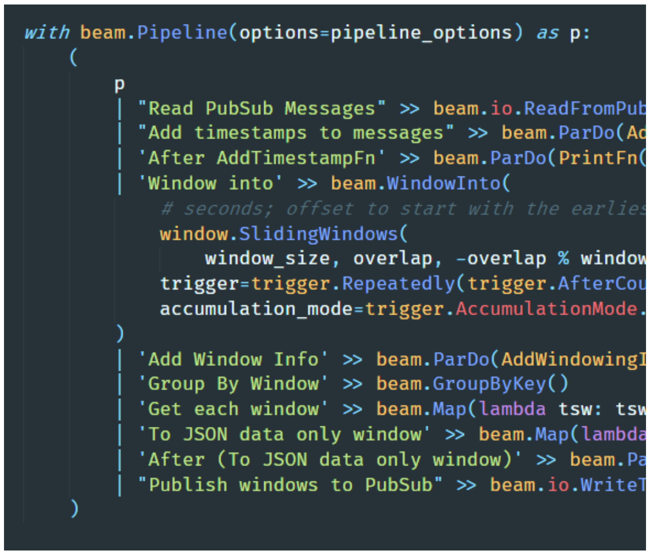
Figure 3: Pelion Notification API webhook contents.

After the data is retrieved and decoded from base64, it is published to the `accel-data` Pub/Sub topic. Any error causes the request to be discarded.

2.4.2 *Windowing with Dataflow.*

Dataflow provides “unified stream and batch data processing that’s serverless, fast, and cost-effective.” [5] It uses Apache Beam to enable advanced real-time stream processing that is uniquely suited to this use-case. Beam makes it simple to create data pipelines for aggregating the incoming data stream into sliding windows to be forwarded to the later stages of the pipeline.

The current Dataflow job is set up using Python and its subscription to the `accel-data` Pub/Sub topic acts as the pipeline’s primary data source. When the pipeline receives new discrete datapoints, it begins by adding timestamps to ensure correct ordering. The next step involves windowing these datapoints into sliding windows. A benefit of sliding windows is that they provide more frequent updates by eliminating the wait for an entirely new data window to be received to produce a new window. Creating sliding windows requires two main parameters: the window size (in seconds), and the period (which dictates how often the window is updated, in seconds). An issue faced with this convention is that our current system assumes perfect timing of data intervals, meaning that this Beam pipeline could produce windows with inconsistent numbers of datapoints. To work around this, an additional *trigger* has been specified which will instead create sliding windows of the required window size after enough points have been collected, and will also emit complete windows after a certain amount of overlap. The window size is currently set to 16 datapoints, and the overlap is set to 8 datapoints, which will produce new windows roughly every 1.6 seconds. Each window is then published to the `windowed-data` Pub/Sub topic.



```
with beam.Pipeline(options=pipeline_options) as p:
    (
        p
        | "Read PubSub Messages" >> beam.io.ReadFromPub
        | "Add timestamps to messages" >> beam.ParDo(Ad
        | 'After AddTimestampFn' >> beam.ParDo(PrintFn(
        | 'Window into' >> beam.WindowInto(
            # seconds; offset to start with the earlies
            window.SlidingWindows(
                window_size, overlap, -overlap % window
            trigger=trigger.Repeatedly(trigger.AfterCou
            accumulation_mode=trigger.AccumulationMode.
        )
        | 'Add Window Info' >> beam.ParDo(AddWindowingI
        | 'Group By Window' >> beam.GroupByKey()
        | 'Get each window' >> beam.Map(lambda tsw: tsw
        | 'To JSON data only window' >> beam.Map(lambda
        | 'After (To JSON data only window)' >> beam.Pa
        | "Publish windows to PubSub" >> beam.io.WriteT
    )
```

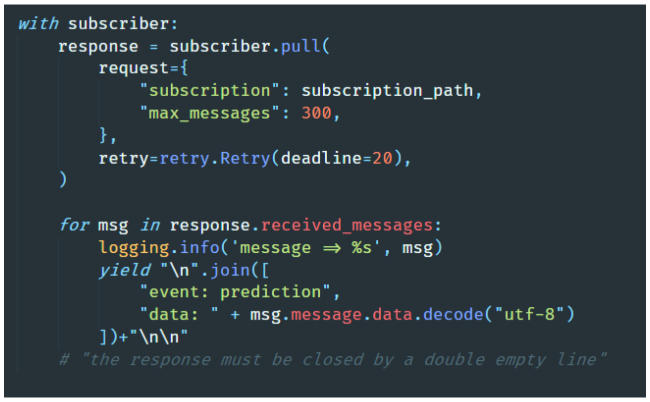
Figure 4: Dataflow: Apache Beam pipeline excerpt.

2.4.3 *Inference / prediction service.*

The activity detection service is implemented as a Python Cloud Function, and is subscribed to the `windowed-data` Pub/Sub topic. This function contains the machine learning model architecture, and uses TensorFlow to predict the activity of each window using an exported saved model available through a Cloud Storage Bucket.

If the function has a “cold start”, it loads the saved model from the Bucket to be cached for subsequent “warm” invocations. Caching the model in this way allows for faster function completion and thus reduced running costs with no adverse effects.

Warm execution of this function averaged about 60ms but varied between 40ms and 130ms, showing that the inference system is capable of keeping up with the incoming data, especially since the sliding windows produced by the previous step are only posted roughly once every 1.6 seconds. Activity predictions are then published to the `model-pred` Pub/Sub topic.



```
with subscriber:
    response = subscriber.pull(
        request={
            "subscription": subscription_path,
            "max_messages": 300,
        },
        retry=retry.Retry(deadline=20),
    )

    for msg in response.received_messages:
        logging.info('message => %s', msg)
        yield "\n".join([
            "event: prediction",
            "data: " + msg.message.data.decode("utf-8")
        ])+"\n\n"
    # "the response must be closed by a double empty line"
```

Figure 5: An excerpt of the Flask server which relays Pub/Sub messages.

2.4.4 Prediction relay service. Evidently, Pub/Sub is good for moving data between services, but this mostly only applies within the GCP ecosystem. After researching whether it was possible to directly subscribe to a Pub/Sub topic externally, it appeared that the system is not meant for this purpose, and the most common solution was instead to create an API endpoint to read from the relevant topic. Following the requirement that activity predictions should be produced periodically, a possible way to address this is using server-sent events. [9]

Server-sent events don't require a user or front-end application to check an API resource continuously, instead opening a connection and relying on the server to send updates in the form of *events*, similarly to websockets. This was achieved by creating a Flask server which is subscribed to the `model-pred` Pub/Sub topic, and provides a simple `/stream` endpoint for relaying published messages.

Using App Engine, the Flask server was deployed and assigned a public URL that can be accessed from any client. Navigating to this URL subscribes the client to server-sent events which are pushed whenever a new message is received by the Pub/Sub subscription. App Engine is essentially free for continuous use, and will enter a dormant state if no requests are detected after a certain amount of time. If a request is issued while in this state, the service is woken up and returns to normal, and the timeout is reset.

2.4.5 Auto-scaling. All the GCP services used in this pipeline are either stateless or capable of auto-scaling to zero workers. In simple terms, the pipeline is always ready to process incoming data while not actually using any billable resources. Not only that, it can also scale up the other way, spawning multiple copies of any one service if its host service detects that the function is falling behind on the input data. Naturally, this would put it at an advantage over traditional, server-based services.

2.5 Inference

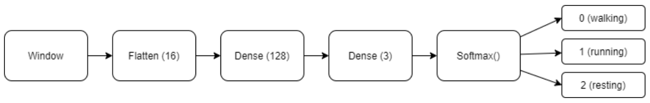


Figure 6: Simplified diagram of the "simple" neural network.

In order to generate activity predictions, preliminary tests were performed on two neural network architectures using Google Colab. Both consist of a one-dimensional input of length 16, which represents a window of accelerometer data of roughly three seconds. This window size was found to generate good predictions in this context. The first model is "simple": a neural network consisting of a single hidden layer of 128 units and a ReLU activation function. The second model is "LSTM": a model consisting of an LSTM layer [7] of four units. Both models output to a layer with two units that are passed through a softmax function to correspond to the two activity classes.

In these tests, the *simple* model was found to have better performance than the LSTM model while also being less complex and training more quickly. The *LSTM* model was abandoned for simplicity. Thanks to the modular cloud architecture presented previously, swapping out the machine learning model used for a more advanced solution would only require a simple code update.

Following testing, additional data was collected to enable the model to classify the *Resting* activity. An Android application was used to collect accelerometer data at the default sensor delay (`SENSOR_DELAY_NORMAL`) of 200,000 microseconds, equivalent to a rate of 5 Hz. [6] Due to the low amount of resting data compared to the existing data, the classes ended up being imbalanced and training was negatively impacted. Since the resting class represents relative lack of movement, it would likely not yield much benefit to collect more data until the classes became balanced. With this in mind, the limited resting data was simply resampled using scikit-learn to match the number of samples present in the existing classes. While this did improve training without needing to collect more data, the examples may still not have been varied enough, thus leading to "perfect classification" as shown in fig. 8. Another explanation is that the testing data was almost identical to the training data as a result of the resampling process being done prior to the data split, which had not been taken into account at the time.

The final "simple" model achieved a test accuracy of 98.95%, and a training loss of 0.5732.

2.6 Front-end application

A simple front-end application was created using React. It features a rounded card centred on the page which displays the server-sent data using a large, descriptive icon accompanied by the activity name and the prediction confidence percentage as reported by the model. All three fields are automatically updated whenever a new event is received. This frontend application could easily be extended with more advanced functionality, provided that the corresponding cloud services are created.

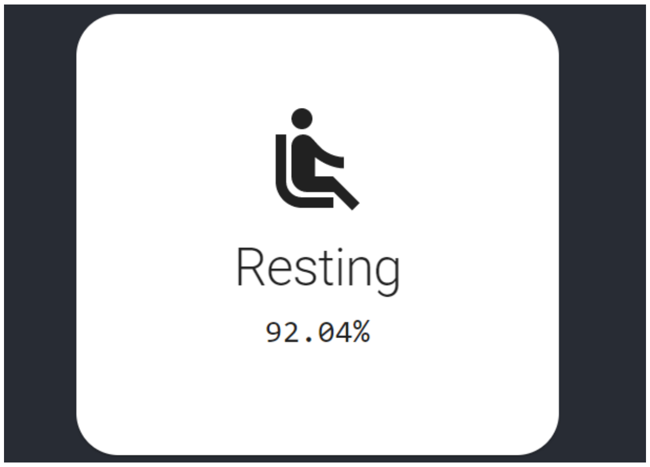
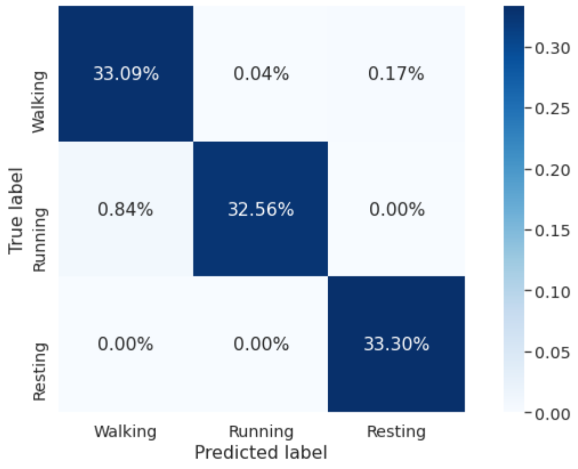


Figure 7: Simple React User Interface.

3 EVALUATION

The prototype system exhibits high test accuracy and low latency while making use of stateless cloud functions and services that support auto-scaling for minimal resource usage. Over the entire project, the GCP budget of $50.00 has only decreased to $49.87 (-$0.13), clearly highlighting the benefits of cloud computing.

Currently, the GCP pipeline only produces live activity predictions, and is not capable of storing those predictions in a database for future reference. Despite not being fully-featured, this modular approach provides a robust starting point for extending the available functionality, with the ability to easily hook into any point on the pipeline – for example, a logical extension would be to introduce a Cloud Function to push timestamped activity predictions to a BigQuery database for persistent storage, thus enabling historical activity



True label \ Predicted label	Walking	Running	Resting
Walking	33.09%	0.04%	0.17%
Running	0.84%	32.56%	0.00%
Resting	0.00%	0.00%	33.30%

Figure 8: Simple model confusion matrix

records and analytics. Another extension could be to use Firebase Cloud Messaging to send push notifications to a user once they have reached a certain activity milestone.

As noted previously, sending raw accelerometer data over the internet at 5 Hz is not an ideal scenario, considering that a lot of data could be permanently lost and systems could easily be overloaded. Producing data windows locally on the firmware seems like a more optimal approach, which could reduce power consumption from radio communication and network congestion.

REFERENCES

[1] Open Mobile Alliance. 2021. *OMA LightweightM2M (LwM2M) Object and Resource Registry.* http://www.openmobilealliance.org/wp/OMNA/LwM2M/LwM2MRegistry.html
[2] ARMmbed. 2021. *ARMmbed/mbed-os-example-pelion.* https://github.com/ARMmbed/mbed-os-example-pelion
[3] ARMmbed. 2021. *Mbed OS – Minimal printf and snprintf.* https://github.com/ARMmbed/mbed-os/blob/master/platform/source/minimal-printf/README.md
[4] Google Cloud. 2021. *Google Cloud – Cloud Functions.* https://cloud.google.com/functions
[5] Google Cloud. 2021. *Google Cloud – Dataflow.* https://cloud.google.com/dataflow
[6] Google Developers. 2021. *Android Documentation: Sensors Overview – Monitoring Sensor Events.* https://developer.android.com/guide/topics/sensors/sensors_overview.html#sensors-monitor
[7] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
[8] Viktor Malyi. 2017. *Kaggle – Run or Walk.* https://www.kaggle.com/vmalyi/run-or-walk
[9] Mozilla. 2021. *Using server-sent events.* https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events
[10] Pelion. 2021. *Pelion – IoT Device Management.* https://pelion.com/product/iot-device-management/
[11] Pelion. 2021. *Pelion – Service API documentation.* https://developer.pelion.com/docs/device-management/current/service-api-references/service-api-documentation.html

[12] Pelion. 2021. *Pelion Documentation – Resources and resource instances*. https://developer.pelion.com/docs/device-management/current/connecting/collecting-resources.html

[13] Pelion. 2021. *Pelion M2MResourceBase*. https://developer.pelion.com/docs/device-management/current/mbed-client/class_m2_m_resource_base.html#ad8891935effd624ca4f66a97ebcd6002

[14] ST. 2021. *Arm Mbed: pelion-example-disco-iot01 (deprecated)*. https://os.mbed.com/teams/ST/code/pelion-example-disco-iot01/

[15] Niall Twomey, Tom Diethe, Xenofon Fafoutis, Atis Elsts, Ryan McConville, Peter Flach, and Ian Craddock. 2018. A Comprehensive Study of Activity Recognition Using Accelerometers. *Informatics* 5, 2 (2018). https://www.mdpi.com/2227-9709/5/2/27