TinyML and LoRaWAN: two practical use cases

Final report for project "CoADDVA"

Mikhail Zolotukhin, Pyry Kotilainen and Timo Hämäläinen
University of Jyväskylä, Jyväskylä, Finland, e-mail: {mizolotu,pyjopeko,timoh}@jyu.fi

Abstract

The rise of low-budget single-board microcontrollers allows service providers to meet privacy, low latency and energy efficiency requirements by deploying artificial intelligence on the edge. In our research, we implement, train and deploy various supervised and unsupervised machine learning models on tiny boards for several data analysis applications. In this report, we describe two such applications. First, we use the electric current and voltage time series observed on a wind turbine to estimate the average wind speed. For this purpose, we use electric current and voltage sensors to perform the measurements and employ a LoRaWAN-enabled microprocessor development board to transmit the measurements obtained. Second, we develop a solution for training unsupervised machine learning models on tiny boards that can be used for anomalous vibration detection. For this purpose, an inertial measurement unit is used to collect necessary statistics in order to build the model of normal behavior and detect outliers. Similarly to the previous use case, the timestamps and scores for the anomalies detected are transmitted over LoRaWAN. The measurement values from the messages sent are then extracted, stored and visualised with the help of various Microsoft Azure cloud services.

I. Introduction

Increasing computing and connectivity capabilities of smart devices allow for their usage on the edge in order to decrease latency and increase availability of various services and applications [8]. Furthermore, the recent progress in development of low-budget sensors and single-board computers has made it possible to deploy tiny machine learning (tinyML) models not only for inference but also for training [16]. In this report, we overview two tinyML applications each of which uses a microprocessor development board combined with several sensors to collect and preprocess necessary data features, train an ML model and make predictions during the inference stage. Some of the data values collected as well as predictions made are then sent to a database over LoRaWAN connection for storing and further processing.

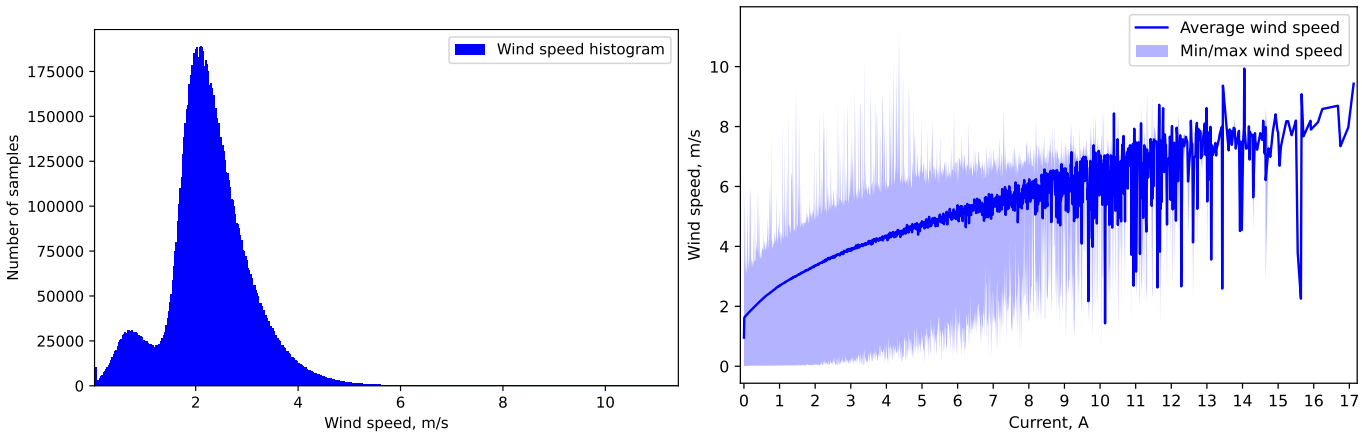
As mentioned in the abstract, we focus on tinyML solutions for wind speed estimation and anomalous vibration detection. In the former case, the estimations are carried out using the electric current time series observed during the recent time interval on a wind turbine developed by Windside. The rationale behind this task is that wind speed estimations obtained can be monitored by the turbine operators in real time for more efficient power extraction and detection of anomalous patterns which may be indicative of a fault.

In the second use case, we study the problem of outlier detection in the data recorded by an inertial measurement unit (IMU). The task is to train a model of normal behavior which can then be used to classify anomalous vibrations that significantly deviate from the norms described by the model. In distinction from the previous case, the training is expected to be carried out on the device itself which can be useful in many use case scenarios when no training data is available in advance. The device is tested using a real pump installed in one of our university buildings and operated by Are.

The rest of the document is organized as follows. The model development process and some of the results obtained are presented in Section II. Hardware and software components of the device prototypes implemented during the project are described in Sections III and IV respectively. Sections V and VI describe how to collect the measurements obtained to store them in a database and visualise the data gathered using serverless computing. Section VII concludes the report and outlines future work.

II. Model development

In this section, we provide the results obtained for each of the use cases overviewed above. First, we describe the data used for training ML models. Next, we briefly summarise the algorithms that we have implemented and tested to solve the problems posed. Finally, some numerical evaluation and algorithm comparison results are provided based on which the best model is selected for each use case. The section is divided into two subsections based on the name of the company to which the corresponding use case is related: Windside - wind speed estimation and Are - anomalous vibration detection.



(a) Wind speed histogram.

(b) Dependence of wind speed on electric current.

Fig. 1: Visualisation of the dataset used in the Windside use case.

Data subset	Start date and time	End date and time
Training	11/05/2022 00:00:00	19/01/2023 09:25:35
Validation	19/01/2023 09:25:36	13/04/2023 07:39:42
Inference	13/04/2023 07:39:43	14/08/2023 08:43:21

TABLE I: Data subsets in the Windside use case.

A. Windside

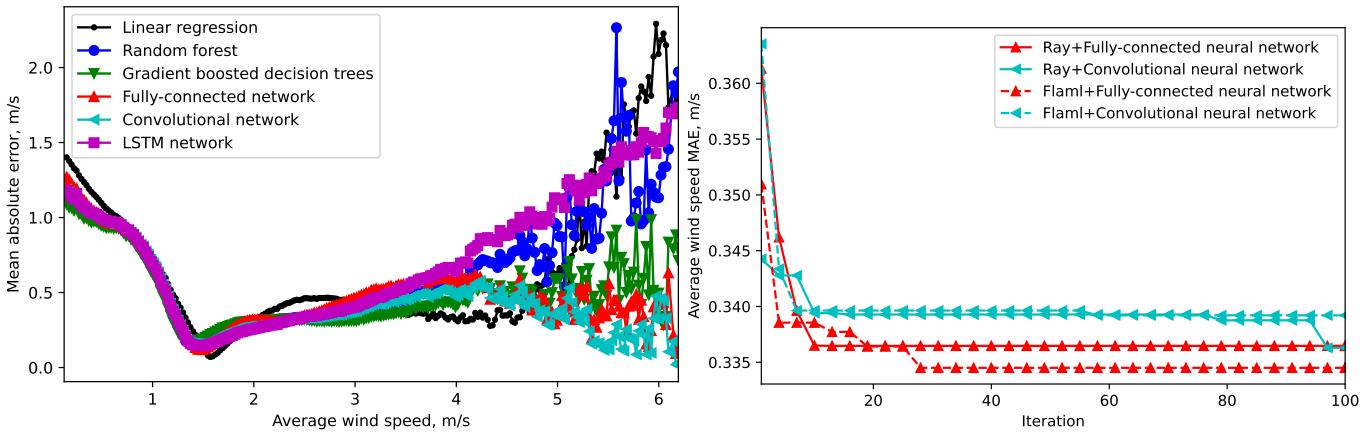
The data used in the first use case was obtained on a real wind turbine in period from 11/05/2022 to 14/08/2023 with one second interval between two adjacent measurements. Each data sample includes the following information: date and time, current, voltage, wind speed and several other metrics. In order to calculate the wind speed the current measurements observed during the last minute are used. It is also worth to mention that we have also tried using the voltage values observed for wind estimation but they do not provide any accuracy improvements. The wind speed value is predicted every second. Thus, input dimension is 60, whereas output dimension is 1. It is worth noticing that a big portion of the resulting input samples (namely around 48.2%) are zero vectors, i.e. the current value is zero during the entire minute. For these vectors, it does not make much sense to try to estimate the wind speed value using machine learning. We instead focus on the remaining data which contain at least one non-zero value. The resulting dataset visualisation is shown in Figure 1. In particular, Figure 1a shows histogram of wind speed values, while the dependence of the wind speed on the electric current in the dataset analysed can be found in Figure 1b.

In order to standardize the data, each sample is converted to z-scores which show how many standard deviations a data point is from the mean. After that, the dataset is divided into three parts: training, validation and inference. The training parts are used to train the corresponding ML models, whereas the main function of the validation parts is to control the models' overfitting. The inference parts are then used to evaluate the models. The size of the inference subset is 20%. The remaining 80% are divided into the training and validation parts with ratio 4:1. The time intervals of the training, validation and inference subsets are shown in Table I.

The processed data can be used to train an ML model. Probably the simplest possible model that can be applied for solving this problem is linear regression which is a simple regression model that generates a sloped straight line or a hyperplane that connects the input variables to the target variable. The coefficients of the linear regression model are found via an optimization process that seeks to minimize the sum squared error between the predictions and the expected target values. We use this model as the baseline.

The next category of algorithms is based on decision trees. A decision tree classifier can be trained using a variety of algorithms. Probably the most popular one, called ID3, is based on measuring entropy values of the features and selecting the one with the smallest value at each iteration to build the tree. Decision tree classifiers are often combined into ensembles via bagging (random forest) or boosting (gradient boosting tree). In this category, the methods tested include random forest regression and gradient boosting decision tree regression [5].

The last but not the least category of methods is based on deep learning approach. This approach relies on training multi-layered neural networks the first layers of which look for extraction of features from raw data samples whereas



(a) Dependence of MAE from wind speed for different tinyML models.

(b) Wind speed estimation accuracy improvement with AutoML.

Fig. 2: Results of our experiments with tinyML in the Windisde use case.

the achievement of the task given is the responsibility of the later layers. The neurons are activated through weighted connections. As a rule, the neuron output is calculated by applying a nonlinear activation function, which allows the network to approximate nonlinear transformations. The learning process is carried out by back-propagating the loss calculated in the output layer towards the input layer. The most popular deep learning models are fully-connected (dense), convolutional [12] and recurrent neural networks [7].

To evaluate the models mentioned, we calculate the mean absolute error (MAE) which is the absolute difference between the true wind speed value and the one predicted by the model. Some results can be found in Figure 2a. The figure shows average MAE values calculated using the inference portion of the dataset for the following six models: linear regression, random forest regression, gradient boosting decision tree regression using XGBoost [3], fully-connected neural network, convolutional neural network and long short-term memory (LSTM) which is a type of recurrent neural network. In our experiments, the random forest and gradient boosting decision tree models consist of 64 decision trees each of which has 64 leaves at maximum. The fully-connected neural network includes two hidden layers of 64 neurons; in the convolutional neural network, one-dimensional convolutional layer with 24 filters of size 20 that moves along the feature vector with stride of size 10 is followed by a dense layer of 64 neurons; LSTM network consists of 36 LSTM cells connected to a dense layer of 64 neurons. As one can see from the figure, the wind speed estimation error is significantly bigger for higher values of wind speed when using linear regression, random forest and LSTM models. On the contrary, the results obtained with fully-connected and convolutional neural networks are somewhat consistent with regards to the wind speed values. We therefore select these two models for further analysis.

In order to find optimal hyperparameters for the models selected, automated machine learning (AutoML) approach can be employed. AutoML is usually defined as automating the process of building and optimising ML models without human assistance [9]. AutoML can automate the process of training and deploying machine learning models and therefore advance ML research by saving time and resources required for finding optimal model parameters. In our research, we rely on the following two AutoML frameworks: Ray [14] and Flaml [19] as these lightweight frameworks are easy and they allow for implementing and optimising custom ML models. We use these frameworks to find optimal architectures for the full-connected and convolutional neural networks. In the case of fully-connected neural network, we search for optimal number of layers and optimal number of neurons in each layer. In the case of the convolutional neural network, we assume there are two layers: a convolutional and a dense one; we use AutoML to search for optimal parameters of the convolutional layer such as the number of filters as well as kernel and stride sizes and the number of neurons in the dense layer. In both cases, we limit the size of the model by the amount of flash memory available on Wio-E5 mini development board on which the model is planned to be deployed. The search is carried out for 100 iterations. The results calculated using the validation portion of the dataset can be found in Figure 2b. As one can notice, the fully-connected neural network allows for slightly lower MAE values. The optimal model architecture found consists of only one fully-connected layer with 246 neurons. The MAE in this case is the lowest and it is equal to 0.3325 m/s for the inference subset. This model is therefore selected to be implemented and deployed on the target LoRaWAN device.

Finally, it is worth mentioning that the problem posed in this use case is quite straightforward as the dependence

Benchmark	Description	Number of datasets	Number of samples	Number of features
ODDS	Various domain data, one record per data point	20	129 - 20000	3 - 36
SKAB	Multivariate time series collected from multiple sensors	20	714 - 1296	8
Ours	Vibrations recorded in the bearing with IMU	1	23998	12

TABLE II: Datasets used in the Are use case.

Dataset	Algorithm accuracy, %											
	SKM	CS	WAP	WCM	SOM	GNG	UAE	SAE	DAE	VAE	DSOM	DSVDD
ODDS	72.1952	73.0376	73.7080	72.9894	71.7573	70.1102	59.6643	59.7354	59.8790	59.9608	65.6161	73.0521
SKAB	89.7119	90.9884	90.9982	90.4495	87.0394	85.8140	60.2228	63.5130	62.3331	58.6843	88.1397	90.5361

TABLE III: Accuracy (%) of various anomaly detection algorithms tested using ODDS and SKAB benchmark datasets in the Are use case.

of wind speed on electric current is somewhat linear. As we can notice when looking at the results obtained, using decision tree and deep learning models allow for quite marginal improvements of wind speed estimation accuracy compared to the baseline linear regression model. We have also tried to improve the results by adding voltage and timestamp to the input feature space. However, none of these new features allow us to increase the wind speed estimation accuracy: in both cases, the error slightly increases compared to the case when only the electric current values are used.

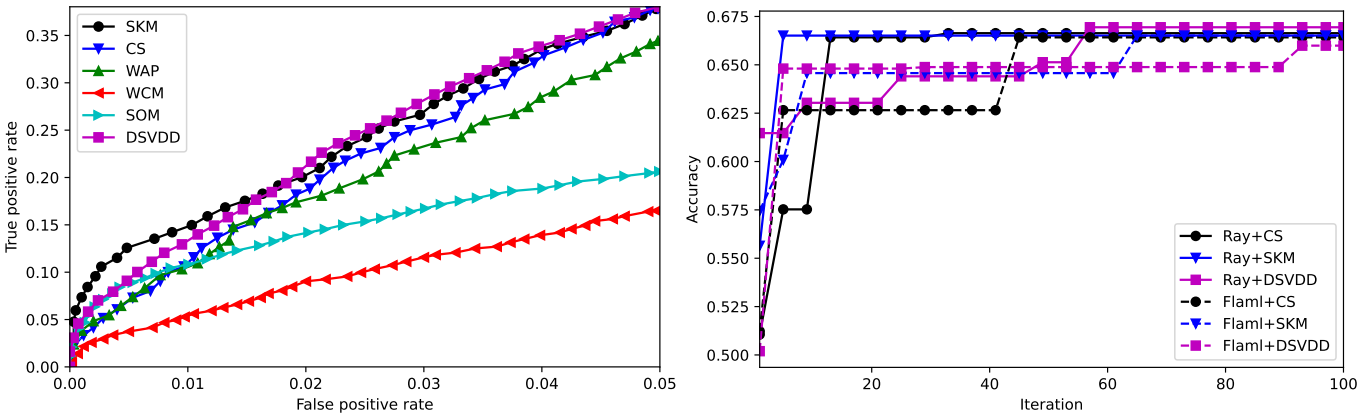
B. Are

As mentioned in the introduction, in the Are use case, the real training data is not available in advance. For this reason, we rely on several publicly available benchmark datasets. In particular, the following two are used: Outlier Detection DataSets (ODDS) [15] and Skoltech Anomaly Benchmark (SKAB) [10]. We have picked 20 datasets from each of these benchmarks and limited the maximum number of samples extracted from each dataset by 20000. We have also built a simple environment which includes an electric motor and two bearings: one is normal and another is with a crack inside. A device with an IMU can be installed on top of the bearing in order to classify vibrations recorded as either produced by the normal or the defective bearing. We have collected several minutes of such IMU data for each of the bearings mentioned. The resulting time series are then transformed into the frequencies using fast Fourier transformation (FFT). The datasets used in the study are summarised in Table II.

In order to standardize the data, each data sample is scaled into range from zero to one using min-max standardization. Similarly to the previous use case, we split all the datasets listed above into training-plus-validation (around 70%) and inference (around 30%) parts and use only normal samples from the former to train the algorithms. Each inference subset is adjusted in such a way that it contains an equal number of normal and abnormal samples which allows us to compare the algorithms in terms of accuracy.

The methods used to solve the problem in this use case can be divided into three categories: the ones based on stream clustering, traditional neural networks and deep learning. The first category of methods classifies a sample as an outlier by measuring its distance from the clusters obtained for the normal data. These clusters are found using one of the stream clustering techniques such as ScalableKMeans++ (SKM) [2], CluStream (CS) [1], weighted affinity propagation (WAP) [20], weighted c-means (WCM) [13] and many others [18]. The next category of algorithms is based on traditional neural networks. Two of the most famous algorithms in this category are self-organising maps (SOM) [11] and growing neural gas (GNG) [6]. These algorithms suit well for the task we aim to solve in this study by design: they compress the data into a set of representative units in an incremental manner by taking a small subset of data at a time. The last but not the least category of methods is based on deep learning approach. The deep learning based methods for anomaly detection tested in this study are several variations of autoencoder including undercomplete autoencoder (UAE), sparse autoencoder (SAE), denoising autoencoder (DAE) and variational autoencoder (VAE) as well as deep self-organising map [4] and deep support vector data description (DSVDD) [17].

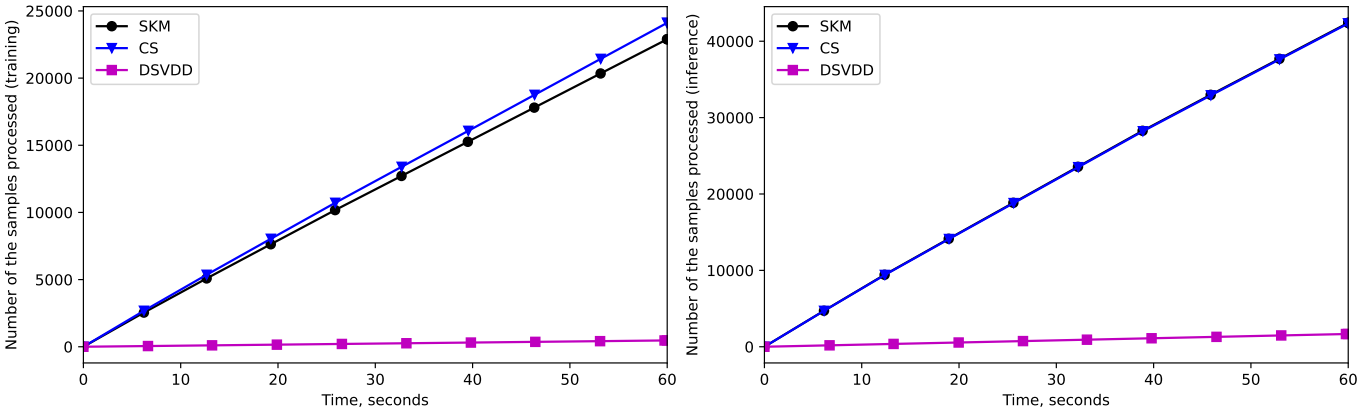
We first evaluate the algorithms listed above in terms of accuracy using the benchmark datasets ODDS and SKAB. For this purpose, the algorithms are implemented with several arbitrarily chosen combinations of hyperparameters from which the best combinations are selected for the comparison. The results obtained for each benchmark can be found in Table III. As one can see, all the algorithms based on stream clustering provide the best results for anomaly detection in each of the benchmark datasets used. Speaking of neural network based algorithms, all the auto-encoder based models tested have resulted in quite low accuracy values compared to other methods. Interestingly enough, even results of the deep self-organising map model in which auto-encoder is also used are worse compared to



(a) Dependence of TPR on FPR.

(b) Accuracy improvement with AutoML.

Fig. 3: Anomaly detection algorithm evaluation using the data generated in the bearing test environment in the Are use case.



(a) Data processing rate (training).

(b) Data processing rate (inference).

Fig. 4: The number of samples that can be processed by each of three algorithms implemented on Wio-E5 mini during the training and inference stage in the Are use case; duration of each stage is one minute.

the standard SOM model. The only deep learning model that shows promising results is deep support vector data description. Thus, based on the accuracy values obtained, we select the following six algorithms for further evaluation: CS, SKM, WAP, WCM, SOM and DSVDD.

Next, we evaluate the algorithms selected using the data generated in the bearing test environment. The dependence of true positive rate (TPR) on false positive rate (FPR) can be found in Figure 3a. It is worth saying that we are interested the most in the methods that provide for higher detection rates in case of low FPR values. For this reason, the figure shows the dependence of TPR on FPR for the cases when the latter is below 5%. In this context, anomaly detection methods based on stream clustering algorithms, scalable k-means++ and CluStream as well as deep learning based support vector data description show the most promising results as one can notice from the figure. Based on the results obtained we select SKM, CS and DSVDD for implementation on the target device.

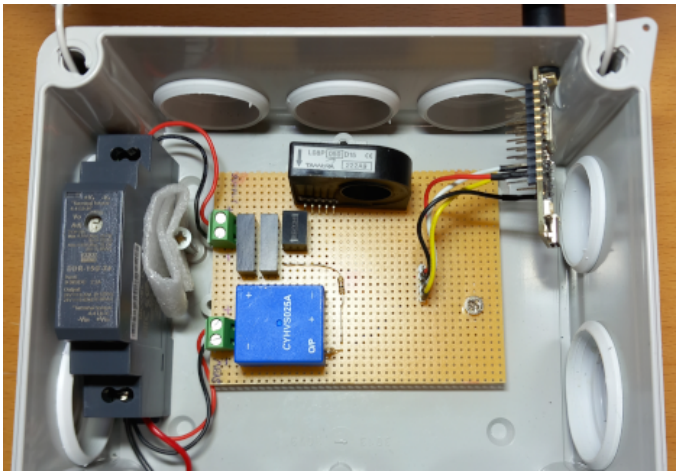
As previously, we can also use the AutoML approach to search for optimal hyperparameters. In the case of scalable k-means++, the hyperparamters are the number of clusters and the batch size whereas in the case of CluStream those are the number of clusters and the number of microclusters. For DSVDD, we search for an optimal two-layer fully-connected network architecture. The improvement of the anomaly detection accuracy over 100 search iterations conducted for the models selected is shown in Figure 3b. As one can notice, all three algorithms provide for somewhat similar accuracy values of around 66.8%.

Finally, we evaluate each selected algorithm's data processing rate during both the training and the inference stage. The hyperparameters used for the implementations are the ones that provide the best accuracy values among the combinations found with Ray and Flaml. For SKM, this combination is 3 clusters with 4 samples per batch;

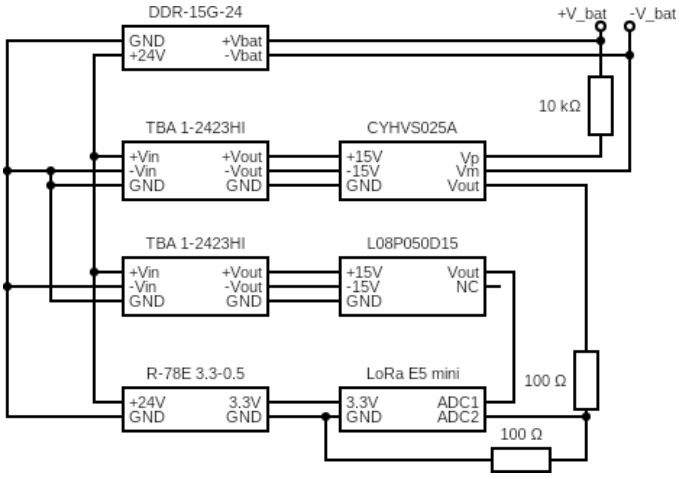
for CluStream - 3 clusters and 4 microclusters; for DSVDD - two layers of 110 and 85 neurons respectively. The models with these hyperparameters are implemented and tested on the target board, i.e. Wio-E5 mini. To calculate the training and the inference data processing rate for each algorithm implementation, we count the number of samples which can be processed by the algorithm during a one minute interval. As one can see from the results obtained (see Figure 4), stream clustering based anomaly detection methods provide for significantly faster data processing rate values during both the training and inference stage. With the hyperparameters obtained, the number of samples that can be processed in one second by SKM, CS and DSVDD models are 381.61, 401.92, 7.85 and 706.82, 705.46, 27.88 during the training and inference stage respectively.

III. Hardware

As mentioned in the previous section, we implement the tinyML models found on LoRaWAN-enabled Wio-E5 mini with 48 MHz ARM Cortex-M4 CPU, 64 KB of RAM and 256 KB of flash memory. The device is a class A LoRa-device meaning that the communication is always initiated by the device, and downlink is only possible in two receive windows after the uplink message. The downlink capability however is not utilised in our prototypes. It is also worth mentioning that the energy consumption is not a problem in the use cases under consideration as there are sources of electricity in the areas where each of the device prototypes built is planned to be deployed and therefore it can be powered over a USB cable. Below we provide more details for each use case separately.

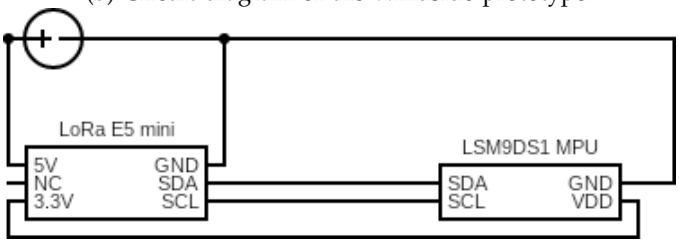


(a) Photo of the Windside prototype.



(b) Circuit diagram of the Windside prototype.

(c) Photo of the Are prototype.

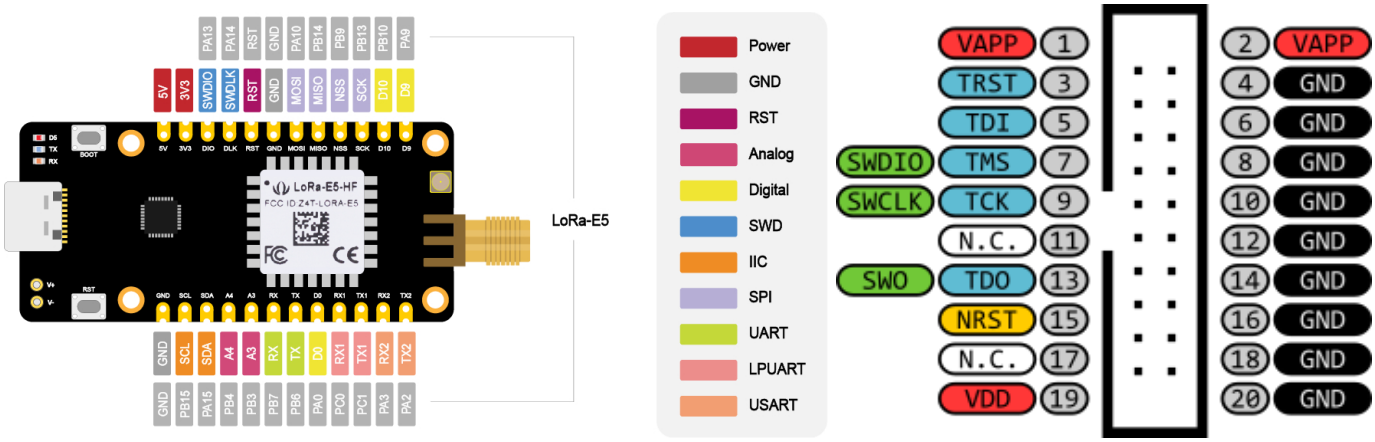


(d) Circuit diagram of the Are prototype.

Fig. 5: Device prototypes built during the project.

1) Windside: In the Windside use case, the device prototype consists of the LoRaWAN-enabled development board Wio-E5 mini and two sensors: a hall-effect voltage sensor CYHVS025A and a hall-effect current sensor L08P050D15. The device gets its power from the wind generator battery system through a DDR-15G-24 24V DC-DC converter to protect the device from voltage spikes. Both sensors require +15V and -15V supplies to operate, so these are generated by two TBA 1-2423HI DC-DC converters. The 3.3V supply for the Wio-E5 mini is generated by a R-78E3.3-05 DC-DC converter. The output of the sensors is connected to the ADC pins of the Wio-E5 mini to be read by the microcontroller, but the voltage sensor output is halved by a voltage divider to make sure the output stays in acceptable range. The photo of the device prototype as well as its circuit diagram can be found in Figures 5a and 5b respectively.

When installing the device, one should connect the input of the DDR-15G-24 DC-DC-converter to the battery or the generator. The voltage sensor also needs to be connected to the battery voltage to measure the voltage. In



(a) Wio-E5 mini pinout.

(b) ST-LINK pinout.

Fig. 6: Pinouts of Wio-E5 mini development board and ST-LINK programmer.

the prototype, the voltage sensor input has a screw terminal which is already connected to the input of the DC-DC-converter, so it does not need to be connected separately. The current sensor requires the wire from which the current is measured being passed through the loop of the sensor, with the direction of the current going according to the arrow printed on the current sensor.

2) Are: In the Are use case, the device prototype consists of only two main components: Wio-E5 mini board and LSM9DS1 IMU to measure a changing acceleration on the sensor. Wio-E5 mini is connected to LSM9DS1 IMU via inter-integrated circuit (I2C) connection that uses two bidirectional open-collector/open-drain lines: serial data line (SDA) and serial clock line (SCL). Power for the built prototype device is fed through a USB connector connected to the nominal 5V input, which can handle voltages in the range of 3.7V – 5V. Both the board and the IMU use +3.3V voltage. The chassis of the prototype has a powerful magnet to attach the device to the mechanism on which model training and/or inference is going to take place. The photo of the resulting device prototype as well as its circuit diagram can be found in Figures 5c and 5d respectively.

IV. Software

In both use cases, we run mbedOS on the target device. The code used can be found in our Github repository. To upload the programming code to a Wio-E5 mini board, ST-LINK can be used. ST-LINK is an in-circuit debugger and programmer for the STM8 and STM32 microcontrollers. Pinout for both Wio-E5 mini and ST-LINK can be found in Figure 6. Wio-E5 mini board can be connected to the ST-LINK as follows:

- ST-LINK pin 1 (VAPP) → Wio-E5 mini 3V3
- ST-LINK pin 7 (SWDIO) → Wio-E5 mini DIO
- ST-LINK pin 9 (SWCLK) → Wio-E5 mini CLK
- ST-LINK pin 19 (VDD) → Wio-E5 mini SCK
- ST-LINK pin 20 (GND) → Wio-E5 mini GND

To be able to upload custom code to a new Wio-E5 mini board, read protection is supposed to be turned off. For this purpose, one should connect the board to the ST-LINK programmer as described above. After that, one is supposed to download, install and start STM32CubeProgrammer software. Next, it is required to connect ST-LINK to the PC where STM32CubeProgrammer is running via a USB port followed by connecting the board itself via another USB port. Next, one should first press reset button on the board, then press ”Connect” in the programmer software and only after that release the reset button. Finally, one have to change ”Read Out Protection” value in OB menu from ”BB” to ”AA”. The changes can be then saved by pressing ”Apply” and then ”Disconnect” in STM32CubeProgrammer. The programming software can after that be closed.

Once the read protection on the board has been disabled, code editing and compiling can be carried out in Mbed Studio. For this purpose, the board is supposed to be connected to ST-LINK in the same way as described above. After that, the directory containing project files downloaded from the Github repository can be opened in Mbed Studio. When opening the project for the first time, Mbed Studio will automatically ask to update the libraries used: ”mbed-os” and ”stm32customtargets”. The update can be carried out by pressing round arrow icon in the bottom right corner. Next, one should select the target device, i.e. LORA_E5_MINI, and the build profile

in the top left corner. After that, the program can be compiled and uploaded to the target device by pressing "Build program" and "Run program" respectively. It is worth mentioning that with the current version of Mbed OS (6.17.0) the I2C-code contains a bug, so that to be able to run the programming code provided, one should edit file "mbed-os/targets/TARGET_STM/i2c_api.c" by commenting out lines 450 and 464 that look as follows: "sleep_manager_lock_deep_sleep();". Otherwise, there will be an error when running the program in the Are use case. The bug has to do with getting the MCU to deep sleep to save power: without this change and manual sleep management in our code the MCU will not enter deep sleep as the I2C code blocks it. For convenience sake, all the links for the resources mentioned above are listed below.

- Github repository: https://github.com/jyuseclab/coaddva
- STM32CubeProgrammer: https://www.st.com/en/development-tools/stm32cubeprog.html
- Mbed Studio: https://os.mbed.com/studio

1) Windside: In this and the next subsection, we briefly overview the programming code used in each of the use cases studied separately. The latest version of the programming code used in the Windside use case can be found in directory mbed/windside of our Github repository. As one can notice when looking through the source files, the program is quite straightforward. The main function in file src/main.cpp starts two threads: one is used to collect the measurements from the current and voltage sensors and another estimates average wind speed based on the current measurements and sends it as well as the current and voltage average values over LoRaWAN. The measurement collection thread runs continuously on the board with interval of 1 second between two adjacent iterations during which the data is collected. To collect the electric current and voltage measurements, analog inputs 2 and 3 are respectively used. The values are collected in unsigned short format, i.e. they can vary between 0 and 65535. These values are appended to two separate circular buffers of size 60, i.e. the buffers are able to track 60 measurements collected during the last 1 minute interval as mentioned in Section II. Some steps of the data collection process are shown in Listing 1.

Listing 1: Collecting measurements from the electric current and voltage sensors in the Windside use case.

```
#define MEASUREMENT_COUNT 60

...

CircularBuffer<uint16_t, MEASUREMENT_COUNT> ain2_buf;
CircularBuffer<uint16_t, MEASUREMENT_COUNT> ain3_buf;

AnalogIn ain2(PB_3);
AnalogIn ain3(PB_4);

...

uint16_t current_adc = ain2.read_u16();
uint16_t voltage_adc = ain3.read_u16();

...

ain2_buf.push(current_adc);
ain3_buf.push(voltage_adc);
```

The thread responsible for sending the results over LoRaWAN runs with interval of 60 seconds between two adjacent iterations. During each iteration, average electric current and voltage values are calculated. In addition, for the electric current, we transform each value I_{sensor} observed with the sensor into amperes $I_{calibrated}$ as follows:

$$I_{calibrated} = f_{calibrate}(I_{sensor} * (3.3/65535.0) * (50/4.0)) \quad (1)$$

where

$$f_{calibrate}(I) = 1.01964624 * I + 0.26233607 \quad (2)$$

In Equation 1, the first multiplier, i.e. $(3.3/65535.0)$, is used to transform the value observed on the sensor to measured analog voltage, and the second multiplier, i.e. $(50/4.0)$, is employed to transform the resulting value into amperes. The latter values are defined based on the sensor specifications. Calibration coefficients in Equation 2 are

obtained experimentally by using a power source with known current value and observing measurements returned by the sensor connected to the board. The array of calibrated electric current values is used as an input to the ML model in order to estimate the average wind speed value observed during the latest one minute interval. This estimation as well as average values of the electric current and voltage transformed to ADC millivolts as described above act as the payload that is sent over LoRaWAN. For this purpose, lorawan_emini library is used. The process of sending results over LoRaWAN is shown in Listing 2.

Listing 2: Sending the results over LoRaWAN in the Windside use case.

```
#define INPUT_SIZE 60

...

float current[INPUT_SIZE];
float wind_speed[1];

uint8_t lora_payload[6];

...

lwemini.init();

...

uint16_t analog_2 = convert2current_and_average(current, ain2_buf);
uint16_t analog_3 = get_average(ain3_buf);

predict(current, wind_speed);
uint16_t windspeed = uint16_t(wind_speed[0] * 1000);

analog_2 = uint16_t(analog_2 * (3.3 / 65535) * 1000);
analog_3 = uint16_t(analog_3 * (3.3 / 65535) * 1000);

if (lwemini.joined) {
    lora_payload[0] = analog_2 >> 8 & 0xFF;
    lora_payload[1] = analog_2 & 0xFF;
    lora_payload[2] = analog_3 >> 8 & 0xFF;
    lora_payload[3] = analog_3 & 0xFF;
    lora_payload[4] = windspeed >> 8 & 0xFF;
    lora_payload[5] = windspeed & 0xFF;

    int16_t retcode = lwemini.send_message(lora_payload, sizeof(lora_payload));

    ...

}
```

Speaking of the function ”predict” used to estimate the average wind speed, it is generated using our custom script prepare_sketch.py which can be found in our Github repository. It is able to convert fully-connected and convolutional neural network models to C file src/wind_prediction/model.h that is stored in the device’s flash memory. The file contains feature standardisation coefficients, weight and bias values for each neural network layer, as well as function ”predict” that uses a float array of length 60 which contains input features, i.e. electric current values, and produces an array of length 1 which is essentially the average wind speed estimate. An example of the predict function for a fully-connected model obtained with the script can be found in Listing 3.

Listing 3: Dense neural network model used in the Windside use case.

```
#define pgm_read_float_near(addr) (*(const float *)(addr))
```

```
#include <math.h>

const int INPUT_SIZE = 60;
const int DENSE0_SIZE = 246;
const int DENSE1_SIZE = 1;

const float x_mean_data[] = {0.34893409f};
const float x_std_data[] = {0.62050781f};
const float W0_data[] = { ... }
const float b0_data[] = { ... }
const float W1_data[] = { ... }
const float b1_data[] = { ... }

inline float x_mean(int i) {
  return pgm_read_float_near(x_mean_data + i);
}

inline float x_std(int i) {
  return pgm_read_float_near(x_std_data + i);
}

inline float W0(int i, int j) {
  return pgm_read_float_near(W0_data + i * DENSE0_SIZE + j);
}

inline float b0(int i) {
  return pgm_read_float_near(b0_data + i);
}

inline float W1(int i, int j) {
  return pgm_read_float_near(W1_data + i * DENSE1_SIZE + j);
}

inline float b1(int i) {
  return pgm_read_float_near(b1_data + i);
}

inline float relu(float x) {
    return fmaxf(0.0f, x);
}

inline float linear(float x) {
    return x;
}

void predict(float* x, float* y) {

    float h0[DENSE0_SIZE];
    for (int i = 0; i < DENSE0_SIZE; ++i) {
        h0[i] = 0.0;
        for (int j = 0; j < INPUT_SIZE; ++j) {
            h0[i] += (x[j] - x_mean(0)) / x_std(0) * W0(j, i);
       }
       h0[i] = relu(h0[i] + b0(i));
    }
```

```
    float h1[DENSE1_SIZE];
    for (int i = 0; i < DENSE1_SIZE; ++i) {
        h1[i] = 0.0;
        for (int j = 0; j < DENSE0_SIZE; ++j) {
            h1[i] += h0[j] * W1(j, i);
        }
        y[i] = linear(h1[i] + b1(i));
    }

}
```

2) Are: In the Are use case, the program is slightly more complicated. The main challenge in this use case scenario is the fact that since the data is not available in advance, the training is supposed to run on the board in online mode. To solve this problem, we set a certain time interval (currently it is 24 hours) during which the model is supposed to be trained. During this training interval, the models are assumed to be fed with data samples that represent normal behavior. Once the training has been completed, the inference thread starts. This thread is responsible for calculating anomaly scores for new data samples collected and sending them over LoRaWAN. The latest version of the programming code used in the Are use case can be found in directory mbed/are.

As one can notice when looking through the program, the main function in file src/main.cpp starts with models training. For this purpose, we collect accelerometer measurements from the IMU connected via I2C lines SDA and SCL. The measurements collected are used as an input to function "put" of an auxiliary feature preprocessor class defined in file src/anomaly_detection/Preprocessor.c. This function appends the accelerometer measurements to a first-in-first-out (FIFO) queue of size 1024. When the queue is full, the mean and standard deviation values are calculated and used as standardisation coefficients for all future data samples collected. Once the standardisation parameters have been found, after every 8 iterations, the latest 8 measurement samples are used to calculate fast Fourier Transform (FFT) for each acceleration component, i.e. x, y and z. The first 1024 of such FFT series are used to calculate standardisation coefficients. After that, the resulting FFT series of length 4 for each of the 3 components are stored in another FIFO queue of size 32. Once this queue has become full, every 8 iterations, the vector that contains maximum FFT values is extracted, standardised and used as the feature vector. The preprocessor is ready when there is a new batch of feature vectors available. Data collection and preprocessing process are shown in Listing 4.

Listing 4: Collecting and preprocessing measurements from the IMU in the Are use case.

```
#define X_DIM 3

...

float x[X_DIM];
bool status = false;

...

I2C i2c(I2C_SDA, I2C_SCL);
LSM9DS1 imu(i2c);

if (((imu.begin() >> 8) & 0xFF) != 0x68) {
    debug("Failed to communicate with LSM9DS1.\n");
}

imu.readAccel();
x[0] = imu.ax;
x[1] = imu.ay;
x[2] = imu.az;

...
```

```
Preprocessor prep = Preprocessor();

...

status = prep.put(x);
```

As mentioned in Section II, there are three models selected to be deployed in the Are use case: SKM, CS and DSVDD. During the training stage, when a new batch of feature vectors is calculated using the preprocessor, the models can be trained using these vectors one by one. The training stage is divided into two parts: training and validation. During the former, trainable parameters of the models are calculated, whereas during the latter, we estimate anomaly score thresholds for the normal samples. In the case of SKM and CS clustering, this score value is the distance between a sample and its closest cluster centroid; positions of the cluster centroids are calculated during the training stage. In the case of DSVDD model, this value is the distance between the neural network output for a sample and the centre of the model. Both the neural network weights and the centre value are calculated during the training. The training and validation processes are shown in Listing 5.

Listing 5: Model training stage in the Are use case.

```
Model* models[] = {new Svdd(), new Skmpp(), new Clstrm()};

...

if (status) {

    t = timer.read_ms();

    if (t - t_stage_start <= d_train_total) {

        for (short i=0; i < prep.get_batch_size(); i++) {
            for (short j=0; j < n_models; j++) {
                models[j]->train(prep.get_feature_vector(i));
            }
        }

    } else if (t - t_stage_start <= d_train_total + d_val_total) {

        for (short i=0; i < prep.get_batch_size(); i++) {
            for (short j=0; j < n_models; j++) {
                models[j]->validate(prep.get_feature_vector(i));
            }
        }

    } else {

        break;

    }

} else {

    ...

}
```

Speaking on the ML algorithms SKM, CS and DSVDD, their implementations can be found respectively in files src/anomaly_detection/Skmpp.c, src/anomaly_detection/Clstrm.c and src/anomaly_detection/SVDD.c. SKM is essentially a lightweight version of popular k-means clustering algorithm: it first samples several centroid

candidates from the data batch based on the distance from the sample to the current cluster centroids. After that, new cluster centroids are selected by using weighted k-means where the weight for each sample is calculated as the sum of weights of the samples which are the closest to the corresponding candidates. CS is the algorithm which relies on creating and updating micro-clusters: each new sample in the data stream is either assigned to the closest micro-cluster if it does not increase the average distortion of this micro-cluster significantly, or a new micro-cluster is created using this new sample as its centroid. In the latter case if the number of micro-clusters exceeds the predefined threshold, two closest micro-clusters merge together. At the end of the clustering process, centroids of the resulting micro-clusters are again clustered using the same weighted k-means. Finally, DSVDD learns a neural network transformation from an input space to an output space that attempts to map most of the data network representations into a hypersphere of minimum volume. Mappings of normal examples fall within, whereas mappings of anomalies fall outside the hypersphere. Since the training in this use case is carried out on the device, backpropagation algorithm for the neural network model is implemented. In our implementation, to avoid exploding gradients, their values are clipped so that they are in range [-0.1, 0.1]. Furthermore, momentum is used to speed up learning and to avoid getting stuck in a local minima.

During the inference, the batch size is reduced to one, i.e. the preprocessor becomes ready once a new feature vector has become available. Other than that, the inference is very similar to the validation stage. In the case of SKM and CS clustering, we calculate the distance between a new sample and its closest cluster centroid found during the training stage. In the case of DSVDD model, we calculate the distance between the centre of the model and the model output given the new sample being fed as the input. The inference stage lasts for one minute. During this time interval, we calculate anomaly score value for each new feature vector obtained with the preprocessor and divide it by the score threshold value obtained earlier for the normal data during the validation stage. After that, the average and the maximum value of this ratio are calculated. The inference process is shown in Listing 6. After that, the device enters sleep mode which lasts for four minutes.

Listing 6: Model inference stage in the Are use case.

```
t_stage_start = timer.read_ms();

while(1) {

    ...

    status = prep.put(x);
    if (status) {
        for (short i=0; i < n_models; i++) {
            models[i]->predict(prep.get_feature_vector());
            score = models[i]->get_score();
            score_thr = models[i]->get_score_thr();
            if (score_thr > 0) {
                score_to_thr_ratio_avg[i] += score / score_thr;
                score_to_thr_ratio_max[i] = (score / score_thr) >
                    score_to_thr_ratio_max[i] ? score / score_thr :
                        score_to_thr_ratio_max[i];
                n_scores[i] += 1;
            }
        }
    }

    t = timer.read_ms();
    if (t - t_stage_start > d_inf) {
        break;
    }
}

...

for (short i=0; i < n_models; i++) {
```

```
    score_to_thr_ratio_avg[i] = n_scores[i] > 0 ?
        score_to_thr_ratio_avg[i] / n_scores[i] : 0.0;
    ...
}
```

The procedure of sending the inference results over LoRaWAN is similar to the first use case. As previously, we use library lorawan_emini. The major part of the message payload in this case consists of the average and maximum anomaly score ratio values calculated during the last inference interval for each of the three anomaly detection models used. For this purpose, these floating point values are multiplied by 1000 and converted to unsigned integers. In addition, the first byte of the payload is set to be equal to the number of times the board does not receive data from the IMU. If this number is high, it is indicative of a problem with the connection between the board or the IMU. The process of sending results over LoRaWAN is shown in Listing 2.

Listing 7: Sending the results over LoRaWAN in the Are use case.

```
uint8_t test_payload[13];

...

uint16_t ar_a[3] = {(uint16_t)(anomaly_ratio_avg[0]*1000),
    (uint16_t)(anomaly_ratio_avg[1]*1000),
        (uint16_t)(anomaly_ratio_avg[2]*1000)};
uint16_t ar_m[3] = {(uint16_t)(anomaly_ratio_max[0]*1000),
    (uint16_t)(anomaly_ratio_max[1]*1000),
        (uint16_t)(anomaly_ratio_max[2]*1000)};

if (lwemini.joined) {
    test_payload[0] = alert;
    test_payload[1] = ar_a[0] >> 8 & 0xFF;
    test_payload[2] = ar_a[0] & 0xFF;
    test_payload[3] = ar_a[1] >> 8 & 0xFF;
    test_payload[4] = ar_a[1] & 0xFF;
    test_payload[5] = ar_a[2] >> 8 & 0xFF;
    test_payload[6] = ar_a[2] & 0xFF;
    test_payload[7] = ar_m[0] >> 8 & 0xFF;
    test_payload[8] = ar_m[0] & 0xFF;
    test_payload[9] = ar_m[1] >> 8 & 0xFF;
    test_payload[10] = ar_m[1] & 0xFF;
    test_payload[11] = ar_m[2] >> 8 & 0xFF;
    test_payload[12] = ar_m[2] & 0xFF;
    int16_t retcode = lwemini.send_message(test_payload, sizeof(test_payload));
}
```

V. Data aggregation

The device prototypes developed during this project utilise a nationwide network provided by Digita. To use their LoRaWAN services one should create a user account by contacting their customer support service. Once such an account has been created, a new LoRaWAN-enabled device can be added to the managed devices via Digita Thingpark web portal. When adding a new device, the most important data fields that must be set are DevEUI, JoinEUI (AppEUI) and AppKey. DevEUI, or device extended unique identifier, is an ID in the IEEE EUI64 address space used to identify a device. The JoinEUI (formerly called AppEUI) is a global application ID in the IEEE EUI64 address space identifying the join server during the over the air activation. AppKey is the encryption key used for messages during every over the air activation. It is worth mentioning that the same values of DevEUI, JoinEUI and AppKey should be entered in file ”mbed_app.json” (can be found in the main directories containing the programming code, i.e. mbed/windside and mbed/are) as values of variables ”lora.device-eui”, ”lora.application-eui” and ”lora.application-key-eui” respectively.

In this project, to accumulate the data sent over LoRaWAN, Microsoft Azure cloud services are used. First, to collect the messages sent by devices, one should create an IoTHub resource in Azure. To redirect the messages

from Digita's Thingpark service to the IoTHub, one may follow the tutorials provided by Digita (see the first link below) or contact their customer support service. It is also possible to forward the messages to a service outside Digita by using a different method with application servers and routing profiles, but such routing is beyond the scope of this report as this approach has not been employed in our project. The management interfaces for Digita services can be accessed as follows:

- Thingpark portal, where LoRa-devices can be added to the network: https://digita.thingpark.com/portal/web.
- Digita's Thingpark X service for setting up the message forwarding to Microsoft Azure: https://digita.thingpark.com/tpx/login

Below are the links to the Digita's Thingpark tutorial how to create an IoTHub on Azure and connect it to the Digita's Thingpark as well as other tutorials available on Digita's Zendesk web portal. The latter require one to have a Digita user account.

- How to create Microsoft Azure IoTHub and connect it to Digita Thingpark: https://docs.thingpark.com/thingpark-x/latest/Connector/AZURE
- How to create an application server: https://digitaiot.zendesk.com/hc/fi/articles/360014260918-Uuden-Application-serverin-m%C3%A4%C3%A4ritt%C3%A4minen
- How to create a routing profile: https://digitaiot.zendesk.com/hc/fi/articles/360014260958-Routing-Profilen-luominen-muokkaaminen
- An overview of the roles of routing profiles and application servers: https://digitaiot.zendesk.com/hc/fi/articles/360020240018-Periaate-esimerkkikuva-erilaisista-AS-ja-RP-konfiguroinneista

Another Microsoft Azure resource on which we rely in this project is Stream Analytics. In both use cases, the messages sent by device prototypes over LoRaWAN are routed from IoTHub to a Stream Analytics job, purpose of which is to parse the messages to extract the data values sent and insert these values to the corresponding tables in an SQL database. Each Stream Analytics job consists of four main components: input, function, query and output. In our use cases, the input is IoTHub, whereas the output contains two tables in the SQL database: one for each use case. A function is employed to parse the payloads of the messages received from the IoTHub. Finally, a query is used to insert the data values extracted into the corresponding tables. Below are the links to some relevant tutorials from Microsoft Azure Documentation portal.

- How to create a Stream Analytics job: https://learn.microsoft.com/en-us/azure/stream-analytics/stream-analytics-quick-create-portal
- How to create an SQL database: https://learn.microsoft.com/en-us/azure/azure-sql/database/single-database-create-quickstart

A. Windside

In this and the next subsection, we provide commands that can be used to create output tables in the SQL database as well as details for the functions and queries used in the Stream Analytics job for each of our use cases separately. In the Windside use case, the table needed can be created by navigating to the SQL database resource's query editor and executing the SQL query shown in Listing 8. As one can see, the table contains columns for the electric current, voltage, power and estimated wind speed values sent by the device over LoRaWAN. In addition, each row in the table includes sensor ID, timestamp and the raw payload received as well as the row ID.

Listing 8: Creating the output table in the Windside use case.

```
CREATE TABLE WindsideIoT (
    ID INT IDENTITY PRIMARY KEY,
    SensorID NVARCHAR(128) NOT NULL,
    date DATETIME,
    data NVARCHAR(128),
    Voltage FLOAT,
    Current FLOAT,
    Power FLOAT,
    Windspeed FLOAT
)
```

Listing 9 shows the function that can be used to parse the LoRaWAN messages sent by the device prototype. This function needs to be created in the Function section of the Stream Analytics job. As one can notice, it uses

exactly the same transformations as the ones applied in the program running on the device, namely the values in ADC millivolts are transformed into volts and amperes and the resulting values are adjusted by using the calibration coefficients found experimentally by using the electricity source with known current and voltage values and observing measurements returned by the sensors connected to the board. The query to push the data values extracted from the messages parsed into the corresponding table of the SQL database can be found in Listing 10. This query needs to be added in the Query section of the Stream Analytics job.

Listing 9: Parsing the messages sent by the device in the Windside use case.

```
function dataParserWindside(rawdata) {

    var data = rawdata;
    if (!data) return [null];
    var n = data.length;
    var result = [];

    var voltage_b = 0.85108956;
    var current_k = 1.01964624;
    var current_b = 0.26233607;

    switch (n) {
        case 12:
            var voltage = parseInt(data.substr(4,4), 16)/1000/0.025;
            var current = parseInt(data.substr(0,4), 16)/1000/0.08;
            if (voltage != 0.0) {
                voltage = voltage_b + voltage;
            }
            if (current != 0.0) {
                current = current_b + current_k*current;
            }

            result[0] = voltage;
            result[1] = current;
            result[2] = voltage * current;
            result[3] = (parseInt(data.substr(8,4), 16)/1000);
            break;
        default:
            result[0] = null;
            break;
    }
    return result;
}
```

Listing 10: Inserting the result values into the SQL table in the Windside use case.

```
SELECT DevEUI_uplink.DevEUI as SensorID, DevEUI_uplink.payload_hex as data,
DATEADD(hour,3,EventEnqueuedUtcTime) as date,
getArrayElement(UDF.dataParserWindside(DevEUI_uplink.payload_hex), 0) AS Voltage,
getArrayElement(UDF.dataParserWindside(DevEUI_uplink.payload_hex), 1) AS [Current],
getArrayElement(UDF.dataParserWindside(DevEUI_uplink.payload_hex), 2) AS [Power],
getArrayElement(UDF.dataParserWindside(DevEUI_uplink.payload_hex), 3) AS Windspeed
INTO WindsideIoT
FROM IoTHub
WHERE getArrayElement(UDF.dataParserWindside(DevEUI_uplink.payload_hex), 0) IS NOT NULL
```

B. Are

Listing 11 shows how to create the table using Azure SQL database's query editor in the Are use case scenario. In this case, the table contains columns for the average and maximum anomaly score ratio values calculated with

three different anomaly detection algorithms on the device during the inference stage as well as the number of measurement errors caused by a faulty connection between the device and the IMU. Similarly to the previous scenario, each row in the table includes sensor ID, timestamp, the raw payload received and the row ID.

Listing 11: Creating the output table in the Are use case.

```
CREATE TABLE AreIoT (
    ID INT IDENTITY PRIMARY KEY,
    SensorID NVARCHAR(128) NOT NULL,
    date DATETIME,
    data NVARCHAR(128),
    Anomaly INT,
    AvgScore1 FLOAT,
    AvgScore2 FLOAT,
    AvgScore3 FLOAT,
    MaxScore1 FLOAT,
    MaxScore2 FLOAT,
    MaxScore3 FLOAT
)
```

The function that is used to parse the LoRaWAN messages sent by the device prototype in the Are use case can be found in Listing 12. This function is supposed to be created in the Function section of the Stream Analytics job. As one can notice, the function simply extracts the integer values from the raw byte payload and transforms them into original floating point numbers by dividing them by 1000. When using such transformation the precision will obviously be decreased but the reduction is tolerable in most of the cases. Finally, the query to push the data values extracted from the messages parsed into the corresponding table of the SQL database can be found in Listing 13. As previously, the query is supposed to be added in the query section of Stream Analytics job.

Listing 12: Parsing the messages sent by the device in the Are use case.

```
function dataParserAre(rawdata) {

    var data = rawdata;
    if (!data) return [null];
    var n = data.length;
    var result = [];

    switch (n) {
        case 26:
            result[0] = (parseInt(data.substr(0,2), 16));
            result[1] = (parseInt(data.substr(2,4), 16)/1000.0);
            result[2] = (parseInt(data.substr(6,4), 16)/1000.0);
            result[3] = (parseInt(data.substr(10,4), 16)/1000.0);
            result[4] = (parseInt(data.substr(14,4), 16)/1000.0);
            result[5] = (parseInt(data.substr(18,4), 16)/1000.0);
            result[6] = (parseInt(data.substr(22,4), 16)/1000.0);
            break;
        default:
            result[0] = null;
            break;
    }
    return result;
}
```

Listing 13: Inserting the result values into the SQL table in the Are use case.

```
SELECT DevEUI_uplink.DevEUI as SensorID, DevEUI_uplink.payload_hex as data,
DATEADD(hour,3,EventEnqueuedUtcTime) as date,
getArrayElement(UDF.dataParserAre(DevEUI_uplink.payload_hex), 0) AS Anomaly,
```

```
getArrayElement(UDF.dataParserAre(DevEUI_uplink.payload_hex), 1) AS AvgScore1,
getArrayElement(UDF.dataParserAre(DevEUI_uplink.payload_hex), 2) AS AvgScore2,
getArrayElement(UDF.dataParserAre(DevEUI_uplink.payload_hex), 3) AS AvgScore3,
getArrayElement(UDF.dataParserAre(DevEUI_uplink.payload_hex), 4) AS MaxScore1,
getArrayElement(UDF.dataParserAre(DevEUI_uplink.payload_hex), 5) AS MaxScore2,
getArrayElement(UDF.dataParserAre(DevEUI_uplink.payload_hex), 6) AS MaxScore3
INTO AreIoT
FROM IoTHub
WHERE getArrayElement(UDF.dataParserAre(DevEUI_uplink.payload_hex), 0) IS NOT NULL
```

VI. Data visualisation

In this section, we briefly describe how the data collected from the devices implemented can be visualised by using serverless computing applications such as Azure functions. The easiest way to create a function in Microsoft Azure is to use Visual Studio Code. For this purpose, one can create a function in a local workspace, then sign in to Azure, create a new Azure Function resource in the Azure account and deploy the function from the local workspace to the Azure one. The detailed tutorials how to create such a function can be found on the Microsoft Azure documentation web pages. Alternatively, one can also create functions in the command terminal or using the Azure portal directly. Below the links to the resources mentioned above are listed.

- Visual Studio Code: https://code.visualstudio.com
- How to create a function in Azure with Python using Visual Studio Code: https://learn.microsoft.com/en-us/azure/azure-functions/create-first-function-vs-code-python
- How to create a Python function in Azure from the command line: https://learn.microsoft.com/en-us/azure/azure-functions/create-first-function-cli-python
- How to create a function in the Azure portal: https://learn.microsoft.com/en-us/azure/azure-functions/functions-create-function-app-portal

A. Windside

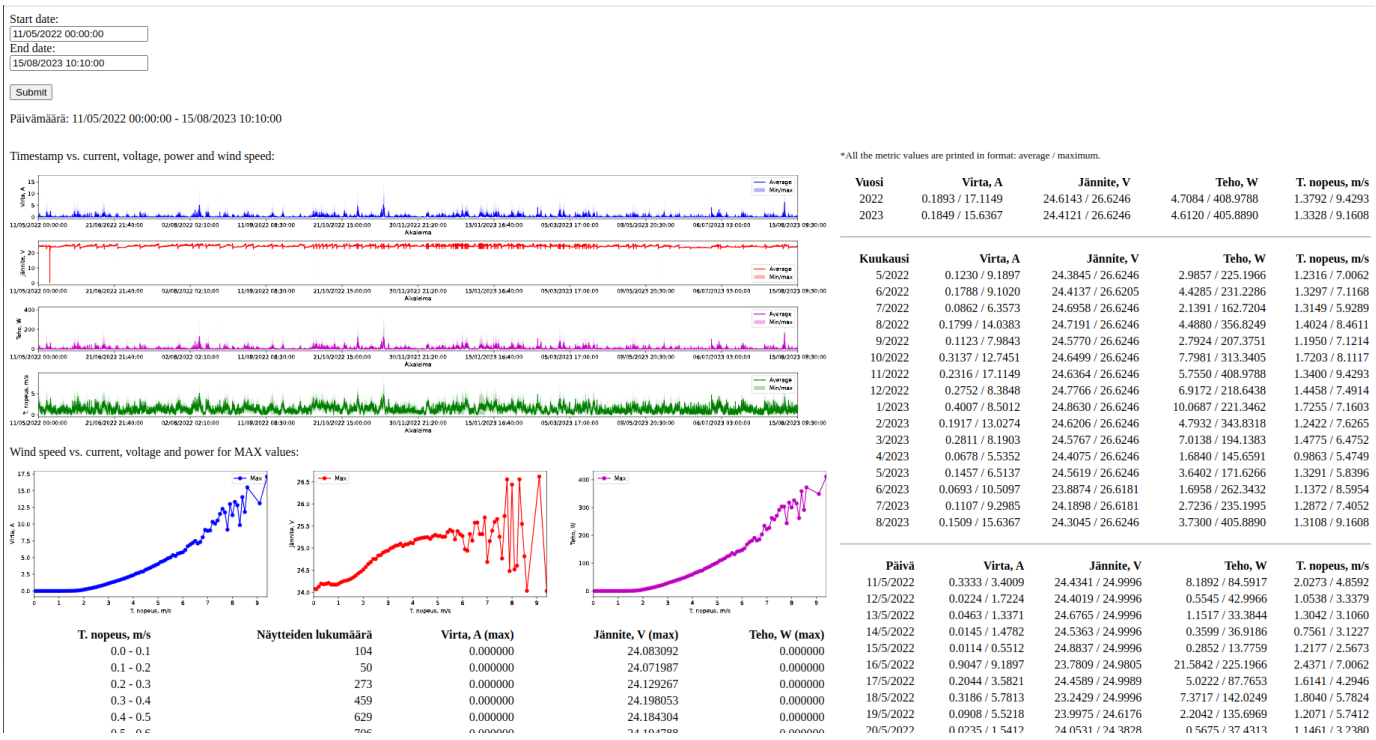
The Azure function for the Windside use case can be found in directory azure/windside/WindsidePowerSummary of our Github repository. This function is triggered by an HTTP request and returns an HTML page. The Python function "generate_html" that generates this page can be found in file http_interval_data/utils.py. As one can notice when looking through the programming code provided, the data is extracted from the database using a simple SQL query that looks for data samples collected in a given time interval between timestamps "sdate" and "edate". The process of data extraction from the database table is shown in Listing 14.

Listing 14: Extracting the data from the SQL database in the Windside use cases.

```
server = 'tcp:jyusqlserver.database.windows.net'
database = 'IoTSQL'
driver = '{ODBC Driver 17 for SQL Server}'
username = ...
password = ...
db_connection_str = f'DRIVER={driver};SERVER={server};PORT=1433;DATABASE={database};\
    UID={username};PWD={password}'

table = ...
date_col = 'date'

...

rows = []
query = [f"select * from {table}"]
if sdate is not None or edate is not None:
    query.append(' where ')
    if sdate is not None:
        sdate_formatted = datetime.strptime(sdate, '%d/%m/%Y %H:%M:%S').\
            strftime('%Y-%m-%d %H:%M:%S')
```

```
        query.append(f"{date_col} >= '{sdate_formatted}'")
        if edate is not None:
            edate_formatted = datetime.strptime(edate, '%d/%m/%Y %H:%M:%S').\
                strftime('%Y-%m-%d %H:%M:%S')
            query.append(f" AND {date_col} <= '{edate_formatted}'")
    else:
        edate_formatted = datetime.strptime(edate, '%d/%m/%Y %H:%M:%S').\
            strftime('%Y-%m-%d %H:%M:%S')
        query.append(f"{date_col} <= '{edate_formatted}'")

query.append(f' order by {date_col}')
query = ''.join(query)

with pyodbc.connect(db_connection_str) as conn:
    with conn.cursor() as cursor:
        cursor.execute(query)
        cols = [col[0] for col in cursor.description]
        for row in cursor.fetchall():
            rows.append(list(row))
```

After that, the data extracted can be visualised by using library matplotlib. In Listing 15, an example of how to plot the time series extracted is shown. In this example, we plot dependence of the current, voltage, power and wind speed values on time. In particular, for each of these metrics, we plot average values and the range between the minimal and the maximal value in the given time interval. The resulting figure with four subplots is base64-encoded and embedded into the output HTML file. In addition to plotting the time series, in the Windside use case, we also plot the dependence of the average electric current on the wind speed. Furthermore, we print the average values of current, voltage, power and wind speed observed during each day, month and year of the time interval given. A screenshot of the resulting HTML page generated with the Azure function in the Windside use case can be found in Figure 7.



Vuosi	Virta, A	Jännite, V	Teho, W	T. nopeus, m/s
2022	0.1893 / 17.1149	24.6143 / 26.6246	4.7084 / 408.9788	1.3792 / 9.4293
2023	0.1849 / 15.6367	24.4121 / 26.6246	4.6120 / 405.8890	1.3328 / 9.1608

Kuukausi	Virta, A	Jännite, V	Teho, W	T. nopeus, m/s
5/2022	0.1230 / 9.1897	24.3845 / 26.6246	2.9857 / 225.1966	1.2316 / 7.0062
6/2022	0.1788 / 9.1020	24.4137 / 26.6205	4.4285 / 231.2286	1.3297 / 7.1168
7/2022	0.0862 / 6.3573	24.6958 / 26.6246	2.1391 / 162.7204	1.3149 / 5.9289
8/2022	0.1799 / 14.0383	24.7191 / 26.6246	4.4880 / 356.8249	1.4024 / 8.4611
9/2022	0.1123 / 7.9843	24.5770 / 26.6246	2.7924 / 207.3751	1.1950 / 7.1214
10/2022	0.3137 / 12.7451	24.6499 / 26.6246	7.7981 / 313.3405	1.7203 / 8.1117
11/2022	0.2316 / 17.1149	24.6364 / 26.6246	5.7550 / 408.9788	1.3400 / 9.4293
12/2022	0.2752 / 8.3848	24.7766 / 26.6246	6.9172 / 218.6438	1.4458 / 7.4914
1/2023	0.4007 / 8.5012	24.8630 / 26.6246	10.0687 / 221.3462	1.7255 / 7.1603
2/2023	0.1917 / 13.0274	24.6206 / 26.6246	4.7932 / 343.8318	1.2422 / 7.6265
3/2023	0.2811 / 8.1903	24.5767 / 26.6246	7.0138 / 194.1383	1.4775 / 6.4752
4/2023	0.0678 / 5.5352	24.4075 / 26.6246	1.6840 / 145.6591	0.9863 / 5.4749
5/2023	0.1457 / 6.5137	24.5619 / 26.6246	3.6402 / 171.6266	1.3291 / 5.8396
6/2023	0.0693 / 10.5097	23.8874 / 26.6181	1.6958 / 262.3432	1.1372 / 8.5954
7/2023	0.1107 / 9.2985	24.1898 / 26.6181	2.7236 / 235.1995	1.2872 / 7.4052
8/2023	0.1509 / 15.6367	24.3045 / 26.6246	3.7300 / 405.8890	1.3108 / 9.1608

Päivä	Virta, A	Jännite, V	Teho, W	T. nopeus, m/s
11/5/2022	0.3333 / 3.4009	24.4341 / 24.9996	8.1892 / 84.5917	2.0273 / 4.8592
12/5/2022	0.0224 / 1.7224	24.4019 / 24.9996	0.5545 / 42.9966	1.0538 / 3.3379
13/5/2022	0.0463 / 1.3371	24.6765 / 24.9996	1.1517 / 33.3844	1.3042 / 3.1060
14/5/2022	0.0145 / 1.4782	24.5363 / 24.9996	0.3599 / 36.9186	0.7561 / 3.1227
15/5/2022	0.0114 / 0.5512	24.8837 / 24.9996	0.2852 / 13.7759	1.2177 / 2.5673
16/5/2022	0.9047 / 9.1897	23.7809 / 24.9805	21.5842 / 225.1966	2.4371 / 7.0062
17/5/2022	0.2044 / 3.5821	24.4589 / 24.9989	5.0222 / 87.7653	1.6141 / 4.2946
18/5/2022	0.3186 / 5.7813	23.2429 / 24.9996	7.3717 / 142.0249	1.8040 / 5.7824
19/5/2022	0.0908 / 5.5218	23.9975 / 24.6176	2.2042 / 135.6969	1.2071 / 5.7412
20/5/2022	0.0235 / 1.5412	24.0531 / 24.3828	0.5675 / 37.4313	1.1461 / 3.2380

T. nopeus, m/s	Näytteiden lukumäärä	Virta, A (max)	Jännite, V (max)	Teho, W (max)
0.0 - 0.1	104	0.000000	24.083092	0.000000
0.1 - 0.2	50	0.000000	24.071987	0.000000
0.2 - 0.3	273	0.000000	24.129267	0.000000
0.3 - 0.4	459	0.000000	24.198053	0.000000
0.4 - 0.5	629	0.000000	24.184304	0.000000

Fig. 7: Results visualisation with Azure function in the Windside use case.

Listing 15: Plotting the time series extracted from the SQL database in the Windside use case. In the Are use case, the process is very similar to this one, we therefore omit the details.

```
data_to_plot = [currents, voltages, powers, windspeeds]
_, axes = pp.subplots(4, 1, figsize=(25, 8))

timestamps_u, timestamps_u_idx = np.unique(timestamps, return_index=True)
n_u = len(timestamps_u)

x = [i for i in range(n_u)]
xticks = [i for i in range(0, n_u, np.maximum(1, n_u // 10))]
xticklabels = [dates[timestamps_u_idx[i]] for i in xticks]

for i, ax in enumerate(axes):
    ax.plot(x, data_to_plot[i]['avg'][timestamps_u_idx], colors[i], \
        label='Average')
    ax.fill_between(x, data_to_plot[i]['min'][timestamps_u_idx], \
        data_to_plot[i]['max'][timestamps_u_idx], facecolor=colors[i], \
            alpha=0.3, label='Min/max')
    ax.set_xlim([x[0], x[-1]])
    ax.set_xticks(xticks)
    ax.set_xticklabels(xticklabels)
    ax.set_xlabel('Aikaleima')
    ax.set_ylabel(labels[i])
    ax.legend()

buff = io.BytesIO()
pp.subplots_adjust(hspace=0.5)
pp.savefig(buff, format='svg', bbox_inches='tight', pad_inches=0)
```

Päivä	Deep SVDD	Scalable KM++	CluStream	Num. of anomalies
24/8/2023	0.2777 / 14.5470	0.4282 / 14.5880	0.4375 / 7.8600	7
25/8/2023	0.2049 / 10.4920	0.3696 / 14.1590	0.3840 / 14.9580	3
26/8/2023	0.1551 / 25.9670	0.3444 / 22.7730	0.3713 / 28.5860	0
27/8/2023	0.1558 / 10.8980	0.3395 / 10.4220	0.3682 / 7.6360	0
28/8/2023	0.2074 / 28.7650	0.3800 / 13.7350	0.4058 / 18.1850	3
29/8/2023	0.1945 / 8.7780	0.3522 / 8.8390	0.3636 / 6.6060	0
30/8/2023	0.2076 / 14.2750	0.3707 / 12.6330	0.3855 / 6.9250	3
31/8/2023	0.2159 / 17.2990	0.3845 / 17.9230	0.4246 / 7.9640	4
1/9/2023	0.1987 / 7.3200	0.3645 / 8.3320	0.3775 / 7.0230	1
2/9/2023	0.1585 / 11.4380	0.3587 / 16.1280	0.3820 / 6.6730	1
3/9/2023	0.1550 / 7.6820	0.3604 / 9.1850	0.3837 / 10.3980	1
4/9/2023	0.2011 / 8.0680	0.3553 / 9.0010	0.3664 / 7.8470	1
5/9/2023	0.1970 / 9.8540	0.3602 / 11.2510	0.3688 / 7.0490	1

Timestamp	Deep SVDD	Scalable KM++	CluStream	Avg score
24/08/2023 13:35:55	1.3470	1.8320	2.5670	1.9153
24/08/2023 13:40:55	1.3190	1.8110	2.5420	1.8907
24/08/2023 13:45:55	1.3460	1.8250	2.5620	1.9110
24/08/2023 14:10:55	0.9710	1.3760	1.8180	1.3883
24/08/2023 14:50:56	1.3320	1.8380	2.5700	1.9133
24/08/2023 14:55:56	1.3770	1.8760	2.6230	1.9587
24/08/2023 15:00:56	1.3590	1.8980	2.6470	1.9680
25/08/2023 11:36:07	1.0590	1.4340	1.9530	1.4820
25/08/2023 11:41:07	1.4190	1.9660	2.7380	2.0410
25/08/2023 11:46:07	1.3630	1.9600	2.7390	2.0207
28/08/2023 15:21:49	1.4980	2.1180	2.9210	2.1790
28/08/2023 15:26:49	1.3520	1.9740	2.7630	2.0297
28/08/2023 15:31:49	1.4650	2.0490	2.8280	2.1140
30/08/2023 11:12:13	1.5200	2.1530	2.9510	2.2080
30/08/2023 11:17:13	1.4660	2.0870	2.8950	2.1493
30/08/2023 11:22:13	1.3690	1.9450	2.6970	2.0037
31/08/2023 06:42:24	0.7960	1.0470	1.2740	1.0390
31/08/2023 15:32:28	1.2810	1.7330	2.4460	1.8200
31/08/2023 15:37:28	1.2990	1.7360	2.4470	1.8273
31/08/2023 15:42:28	1.4010	1.8420	2.5650	1.9360
01/09/2023 06:42:37	0.9300	1.0790	1.5140	1.1743
02/09/2023 08:42:51	1.0740	1.2880	1.7810	1.3810
03/09/2023 08:43:04	1.1300	1.4080	1.9420	1.4933
04/09/2023 06:43:16	1.2840	1.6550	2.2870	1.7420
05/09/2023 06:43:29	1.3210	1.6930	2.3380	1.7840
06/09/2023 06:43:43	1.3510	1.7930	2.4890	1.8777
07/09/2023 06:43:56	1.3180	1.7780	2.4550	1.8503
07/09/2023 09:53:57	1.4900	2.1580	2.9800	2.2093
07/09/2023 09:58:57	1.5850	2.2460	3.0780	2.3030
07/09/2023 10:03:57	1.4110	2.0800	2.8820	2.1243
08/09/2023 06:44:09	1.1740	1.4960	2.0830	1.5843
09/09/2023 08:44:23	0.9960	1.2690	1.7930	1.3527

Fig. 8: Results visualisation with Azure function in the Are use case.

```
pp.close()
img_byte_arr = buff.getvalue()
img_str = base64.b64encode(img_byte_arr).decode()
series_img_tag = f'<img src="{img_str}" style="width:100%"/>'
```

B. Are

The Azure function for the Are use case can be found in directory azure/are/AreAnomalySummary of our Github repository. Similarly to the previous use case, the function is triggered by an HTTP request and returns an HTML page the process of generation of which can be found in file http_interval_data/utils.py. The data collected during a given time interval between timestamps "sdate" and "edate" is extracted from the database using exactly the same process as in the Windside use (see Listing 14). The data extracted from the database is then visualised in a very similar way as in the Windside use case (see Listing 15). In the Are use case, the time series to be plotted are average and maximum anomaly score ratio values observed on the device during the inference stage. As previously, the resulting figure is base64-encoded and then embedded into the output HTML page. We also print the average values of the anomaly score ratio values observed during each day and month of the time interval given. A screenshot of the resulting HTML page generated with the Azure function in the Are use case can be found in Figure 8.

VII. Conclusion

In our research, we have implemented and deployed several supervised and unsupervised machine learning models on tiny boards for two real-life data analysis applications. First, we have used the electric current and voltage time series observed on a wind turbine to estimate average wind speed. Second, we have developed a solution for training an unsupervised machine learning model on a tiny board that can be used for anomalous vibration detection. We have overviewed the hardware and software used for both the solutions developed as well as briefly described the process of the resulting data collection and visualisation using Microsoft Azure cloud services. In the future, we are planning to improve the device prototypes implemented based on the feedback provided by the companies that have participated in the project.

References

[1] Aggarwal, C. C., Philip, S. Y., Han, J., and Wang, J. A framework for clustering evolving data streams. In *Proceedings 2003 VLDB conference* (2003), Elsevier, pp. 81–92.

[2] Bahmani, B., Moseley, B., Vattani, A., Kumar, R., and Vassilvitskii, S. Scalable k-means++. *arXiv preprint arXiv:1203.6402* (2012).

[3] Chen, T., and Guestrin, C. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2016), KDD '16, ACM, pp. 785–794.

[4] Forest, F., Lebbah, M., Azzag, H., and Lacaille, J. Deep embedded som: joint representation learning and self-organization. *reconstruction 500* (2000), 500.

[5] Friedman, J. H. Stochastic gradient boosting. *Computational statistics & data analysis 38*, 4 (2002), 367–378.

[6] Fritzke, B. A growing neural gas network learns topologies. *Advances in neural information processing systems 7* (1994).

[7] Hochreiter, S., and Schmidhuber, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780.

[8] IoT Analytics. State of IoT - Spring 2022. https://iot-analytics.com/product/state-of-iot-spring-2022/, 2022. [Online; accessed 18-May-2022].

[9] Karras, A., Karras, C., Schizas, N., Avlonitis, M., and Sioutas, S. Automl with bayesian optimizations for big data management. *Information 14*, 4 (2023), 223.

[10] Katser, I. D., and Kozitsin, V. O. Skoltech anomaly benchmark (skab). https://www.kaggle.com/dsv/1693952, 2020.

[11] Kohonen, T. The self-organizing map. *Proceedings of the IEEE 78*, 9 (1990), 1464–1480.

[12] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems* (2012), F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., vol. 25, Curran Associates, Inc.

[13] Li, C.-H., Huang, W.-C., Kuo, B.-C., and Hung, C.-C. A novel fuzzy weighted c-means method for image classification. *International Journal of Fuzzy Systems 10*, 3 (2008), 168–173.

[14] Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J. E., and Stoica, I. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118* (2018).

[15] Rayana, S. Odds library. http://odds.cs.stonybrook.edu, 2016.

[16] Ren, H., Anicic, D., and Runkler, T. A. Tinyol: Tinyml with online-learning on microcontrollers. In *IJCNN* (2021), IEEE, pp. 1–8.

[17] Ruff, L., Vandermeulen, R., Goernitz, N., Deecke, L., Siddiqui, S. A., Binder, A., Müller, E., and Kloft, M. Deep one-class classification. In *ICML* (2018), PMLR, pp. 4393–4402.

[18] Silva, J. A., Faria, E. R., Barros, R. C., Hruschka, E. R., Carvalho, A. C. d., and Gama, J. Data stream clustering: A survey. *ACM Computing Surveys (CSUR) 46*, 1 (2013), 1–31.

[19] Wang, C., Wu, Q., Weimer, M., and Zhu, E. Flaml: A fast and lightweight automl library. *Proceedings of Machine Learning and Systems 3* (2021), 434–447.

[20] Zhang, X., Furtlehner, C., and Sebag, M. Data streaming with affinity propagation. In *Joint european conference on machine learning and knowledge discovery in databases* (2008), Springer, pp. 628–643.