Antmicro

Zephyr Sensor Anomalies

2026-05-21

CONTENTS

CHAPTER

ONE

INTRODUCTION

Zephyr Sensor Anomalies is a Zephyr RTOS library that allows to track anomalies in a provided set of sensors using machine learning models running on Kenning Zephyr Runtime or emlearn. It provides tools for:

- *Reading data from sensors* and using it to generate datasets,
- *Real-time anomaly detection* using trained neural networks.

CHAPTER

TWO

GETTING STARTED

2.1 Installing dependencies

To get started with the library, ensure that uv is installed first. Also, ensure that your system has Zephyr dependencies installed.

Clone the repository to get started:

```
mkdir -p zephyr-workspace && cd zephyr-workspace
git clone https://github.com/antmicro/zephyr-sensor-anomalies
cd zephyr-sensor-anomalies
```

Initialize the Python virtual environment:

```
uv venv --python=3.11
source .venv/bin/activate
uv pip install west
```

Initialize the `west` build tool and the Zephyr project:

```
west init -l
west update
west patch apply
west zephyr-export
uv pip install -r ../zephyr/scripts/requirements.txt
# Install the toolchain for your board
west sdk install -t arm-zephyr-eabi
```

Because the example application uses Kenning, you need to install it:

```
uv pip install 'kenning[renode,reports,tensorflow,tflite,torch,uart,zephyr] @␣
↪git+https://github.com/antmicro/kenning'
```

2.2 Building the anomaly detection sample

This section provides a walkthrough of building one of the examples provided in the repository. The final output is an emulated Zephyr application running on Renode, which detects the presence of anomalies from data obtained by sensors.

2.2.1 Demo Application for reading and printing sensor data

The example consists of training an anomaly detection model for a set of two LIS2DS12 accelerometers. The sample app in `samples/sensors` will read values from all available sensors and print them to the Zephyr console UART.

First, build the application:

```
west build -p -b stm32f746g_disco samples/sensors
```

The application can be run on physical hardware and collect actual sensor readings. For the purposes of this demonstration, the `run_renode.py` script may be used to launch a Renode simulation and feed data from a CSV file to the simulated sensors.

Download sample data:

```
curl https://dl.antmicro.com/kenning/datasets/anomaly_detection/minispot.csv -o␣
↪data.csv
```

Inspecting the CSV file, the first 8 columns correspond to the dataset features; while the last column indicates whether the features in the given row is anomalous.

```
roll,pitch,gx,gy,gz,ax,ay,az,anomaly
-0.0000,0.0013,-0.0042,0.1317,-0.0200,-0.0039,-0.0007,-0.0744,0
-0.0001,0.0025,-0.0063,0.1134,-0.0177,0.0005,0.0001,-0.1005,0
-0.0002,0.0033,-0.0055,0.0890,-0.0151,0.0010,0.0001,-0.1006,0
-0.0002,0.0040,-0.0052,0.0669,-0.0132,0.0009,0.0001,-0.1002,0n
-0.0002,0.0041,-0.0024,0.0107,-0.0126,0.0030,0.0000,-0.1017,0
-0.0003,0.0034,-0.0043,-0.0708,-0.0156,0.0044,-0.0001,-0.1026,0
-0.0003,0.0020,-0.0051,-0.1391,-0.0165,0.0034,-0.0000,-0.1020,0
-0.0004,0.0002,-0.0052,-0.1827,-0.0155,0.0017,0.0000,-0.1010,0
-0.0004,-0.0019,-0.0054,-0.2100,-0.0139,0.0005,0.0000,-0.1001,0
```

Download Renode and set necessary environment variables:

```
./scripts/prepare_renode.sh
source ./scripts/activate_renode.sh
```

Then use the script to run a simulation:

```
python ./scripts/run_renode.py --timeout 5 --frequency 100 --data data.csv
```

This will run the simulation for 5 seconds, feeding sensor data from the `data.csv` file with the frequency of 100Hz. One line in the CSV represents one data sample (for all sensors). If no file is provided, random values will be used.

2.2.2 Creating an anomaly detection model

Once data is gathered from the sensors, a model can be trained to detect anomalies from the sensors in real time. In this example, a simple binary classifier is used to train the anomaly detection model. A Kenning scenario (a YAML file with Kenning configuration) from one of the example `workflows` provided in this repository can be used.

The model can be trained with a labeled dataset that Kenning will automatically download from `https://dl.antmicro.com/kenning/datasets/anomaly_detection/minispot.csv`.

```
pushd workflows/minispot
kenning train --cfg scenario.yml
```

After training the model, compile it for efficient execution using `TFLite Micro` runtime:

```
kenning optimize --cfg scenario.yml
```

This will generate two files: `fp32.1.tflite` and `fp32.1.tflite.json`. Both will be saved in `build`.

To run this model, another example application may be used - `samples/anomaly`. While `samples/sensors` simply prints the sensor outputs to UART, `samples/anomaly` will run them through the anomaly detection model.

The Zephyr Sensor Anomalies library will do the following:

- **Pre-Processing**: creating a sliding window,
- **Post-Processing**: "smoothing" the output by removing outliers and using output from the binary classifier to compute an anomaly "metric" of data.

The library does so automatically, so this step does not require any additional setup from the user.

Go back to the main project folder:

```
popd
```

And build the application with the model:

```
west build -p -b stm32f746g_disco samples/anomaly -- \
  -DCONFIG_KENNING_TFLITE_BUFFER_SIZE=100 \
  -DCONFIG_ANOMALY_LIB_KENNING_MAX_INPUT_COUNT=1024 \
  -DCONFIG_KENNING_PROTOCOL_INTEGRATION=n  \
  -DEXTRA_CONF_FILE=kenning.conf \
  -DCONFIG_KENNING_MODEL_PATH=\"$(realpath ./workflows/minispot/build/fp32.1.
↪tflite)\"
```

Then, use the mentioned script and run it on random data:

```
python ./scripts/run_renode.py --timeout 5 --frequency 20
```

Alternatively, run the app on the dataset downloaded earlier:

```
python ./scripts/run_renode.py --timeout 5 --frequency 20 --data data.csv
```

The output should look as such:

```
0.775274,0.105284,0.373280,0.637687,0.612562,0.991824,0.999648
0.857826,0.588634,0.695115,0.809970,0.736989,0.967896,0.989194
0.714257,0.872183,0.009571,0.843470,0.387637,0.071784,0.999842
0.866201,0.814756,0.120837,0.952343,0.656829,0.719043,0.998614
0.330209,0.293120,0.789631,0.211764,0.090927,0.996610,0.934414
0.229710,0.454636,0.053838,0.971486,0.853041,0.230907,0.248174
```

(continues on next page)

(continued from previous page)

```
0.674775,0.010767,0.943968,0.551545,0.585045,0.202193,0.996867
0.909272,0.440279,0.384048,0.100498,0.167497,0.967896,0.998475
0.369691,0.632901,0.144765,0.924825,0.708275,0.843470,0.912978
0.066999,0.503689,0.966700,0.398404,0.451047,0.267996,0.934081
0.044267,0.234496,0.492921,0.173479,0.652044,0.546760,0.622759
0.622133,0.579063,0.191425,0.943968,0.378065,0.494117,0.789022
0.068195,0.373280,0.407976,0.805184,0.717846,0.173479,0.303264
0.722632,0.154337,0.551545,0.617348,0.722632,0.933200,0.001524
0.494117,0.427118,0.044267,0.770488,0.665204,0.240478,0.003188
0.775274,0.090927,0.971486,0.135194,0.541974,0.028713,0.985690
0.618544,0.513260,0.369691,0.436690,0.082552,0.311066,0.999548
0.747757,0.449850,0.369691,0.202193,0.867398,0.851844,0.991345
```

The first 6 columns are sensor values. The rightmost column is the model's prediction (e.g., the probability of the anomaly).

The application will emit a warning if your model is too slow; the basis for this is the expected inference time provided by the `CONFIG_PROCESSING_DELAY` Kconfig option (the default value is 50 ms).

The `sample/anomaly` application can easily be used on a different model and sensor array. If you wish to do that, the code of the application does not need to be modified in any way (since it automatically detects available sensors); however, be sure the sensor you are using is described in `lib/provider_lib/sensor_map.yaml`. All you need to do is change the structure of the model in the `model.py` file.

Evaluating the model in Kenning

The `samples/anomaly` application supports Kenning Protocol. Because of that, it can be used for not only training and optimization, but also for evaluating the accuracy of the model and generating a report with Kenning.

For that, first build the application with Kenning Protocol integration enabled. To achieve that, the value of `DCONFIG_KENNING_PROTOCOL_INTEGRATION` is changed:

```
west build -p -b stm32f746g_disco samples/anomaly -- \
  -DCONFIG_KENNING_TFLITE_BUFFER_SIZE=100 \
  -DCONFIG_ANOMALY_LIB_KENNING_MAX_INPUT_COUNT=1024 \
  -DCONFIG_KENNING_PROTOCOL_INTEGRATION=y  \
  -DEXTRA_CONF_FILE=kenning.conf \
  -DCONFIG_KENNING_MODEL_PATH="$(realpath ./workflows/minispot/build/fp32.1.
↪tflite)"
```

In previous examples, `run_renode.py` script was used, which generated a Renode .repl file (file describing the simulated platform) in the `/tmp` directory, for the board based on Zephyr DTS file, using `gen_repl.py` script. Kenning requires the `.repl` file to be located in the `build directory`, so call `gen_repl.py` directly:

```
python ./scripts/gen_repl.py --repl build/stm32f746g_disco.repl
```

Please note, that the filename must be the same as name of the board being used.

A Kenning scenario will be used. It can be found in one of the example workflows:

```
pushd workflows/minispot
kenning test report \
  --report-path reports/minispot.md \
  --to-html \
  --cfg scenario.yml \
  --measurements results.json
```

In `report/report/report.html`, you can find an HTML version of the report that contains:

- confusion matrix,
- inference quality metrics,
- detection rate (number of anomalies detected),
- false alarm rate (number of false positives),
- detection delay depending on detection threshold.

To run this evaluation on a different board, create an overlay file telling Kenning which UART port to use by adding the `kcomms` alias. An example can be found in `samples/anomaly/boards/stm32f746g_disco.overlay`. You should use this file to define any additional sensors that are not part of the board by default. In such case, please remember you need to modify the `gen_repl.py` script in order to add these sensors to the `.repl`, and update the sensor information in `scenario.yml`.

2.3 Zephyr sensor anomalies structure

The `zephyr-sensor-anomalies` library contains the following directory structure:

- `docs` with markdown files for the documentation,
- `include` with library headers, divided into two sub-libraries:
  - `anomaly_lib` which provides headers for detecting anomalies based on a trained anomaly detection model,
  - `provider_lib` which provides headers for a lightweight abstraction to register multiple heterogenous data sources and read all provider data as a single continuous float array.
- `lib` containing the library for `anomaly_lib` and `provider_lib`,
- `samples` with samples of applications that use the library,
- `scripts` with helper scripts to aid in development,
- `workflows` containing an end-to-end build system showcasing how the sample application, library, and Kenning work together,
- `zephyr` containing Zephyr-specific settings.

CHAPTER

THREE

READING SENSOR DATA

This tutorial outlines how to build a C application from scratch using `zephyr-sensor-anomalies` to read sensor data from the LIS2DS12 accelerometer on the STM32F746NG board.

3.1 Quickstart Example

Ensure that you have taken the necessary steps to *install all dependencies*. This tutorial assumes that you have installed `west` and have the `uv` virtual environment in place.

```
# If you have not yet activated the virtual environment:
source .venv/bin/activate
# Create our private workspace:
mkdir -p sensor_app && cd sensor_app
```

In the `sensor_app/` directory, create the following files:

```
sensor_app/
    boards/
        stm32f746g_disco.overlay
    src/
        main.c
    CMakeLists.txt
    prj.conf
```

This tutorial shows how to write a simple Zephyr application that prints out the obtained values of two accelerometers attached to the board. Below are the full contents of `main.c`:

```
#include <zephyr/kernel.h>

#include <stdio.h>

#include <provider_lib/provider.h>
#include <provider_lib/providers/sensor.h>

#define CSV_READER_MAX_ROW_SIZE 128

int main() {
    static struct provider_reader pr = {0};

    static float buffer[CSV_READER_MAX_ROW_SIZE] = {0};
```

(continues on next page)

(continued from previous page)

```
    static struct provider_hdr_entry hdr[CSV_READER_MAX_ROW_SIZE] = {0};

    int ret = provider_reader_register_all_sensor(&pr);
    provider_reader_hdr(&pr, hdr);

    if (sizeof buffer / sizeof *buffer < pr.dst_N) {
        printk("Insufficient buffer size: %u < %u", sizeof buffer / sizeof␣
↪*buffer, pr.dst_N);
        return -1;
    }

    /* Fill in the headers */
    for (int i = 0; i < pr.dst_N; ++i) {
        if (i > 0) {
            printk(",");
        }
        printk("%u_%u", hdr[i].provider_id, hdr[i].chan);
    }
    printk("\n");

    while (1) {
        provider_reader_read_all(&pr, buffer);

        /* Print out the results */
        for (int i = 0; i < pr.dst_N; ++i) {
            if (i > 0) {
                printk(",")
            }
            printk("%f", (double) buffer[i]);
        }
        printk("\n");

        k_msleep(200); /* Sleep for 200 ms */
    }

    return 0;
}
```

3.1.1 Step-by-step of the code example

This subsection will go into the code in more detail.

The code below initializes the primary struct for the provider reader context. This struct contains: `pr.ps_N`, which is the *number of providers* and `pr.dst_N`, which is the *total number of channels*. For example, if there are two providers, one with two channels and another with three, then `pr.dst_N` is set to 5 by this library.

When reading from a provider, the obtained sensor data is saved into `buffer[]`. Ensure that there is enough memory that can hold the data from all sensors. This buffer is overwritten whenever the sensors are sampled; callers must copy the sensor data if persistence is required.

The third line is an array of the headers of each provider. This contains the `hdr.provider_id` member variable and the `hdr.chan`, which indicates which channel it is.

```
static struct provider_reader pr = {0};
static float buffer[CSV_READER_MAX_ROW_SIZE] = {0};
static struct provider_hdr_entry hdr[CSV_READER_MAX_ROW_SIZE] = {0};
```

This line attempts to register all the available sensors stated in the config file `kenning-zephyr-sensor-anomalies/lib/provider_lib/sensor_map.yaml`:

```
int ret = provider_reader_register_all_sensors(&pr);
```

The config file contains the name of the sensor and its channels:

```
- st_lis2ds12:
  - ACCEL_X
  - ACCEL_Y
  - ACCEL_Z

- adi_axl345:
  - ACCEL_X
  - ACCEL_Y
  - ACCEL_Z
```

The name of the sensor maps to the `compatible` field in each of the sensor in the DTS overlay with the , replaced with _. In this instance, `st_lis2ds12` is the sensor name which maps to `st,lis2ds12` in the DTS overlay file. More information about the DTS overlay can be found in the Zephyr documentation. The DTS overlay is laid out in `boards/stm32f746g_disco.overlay`:

```
&i2c1 {
    lis2ds12N1: lis2ds12@3d {
        compatible = "st,lis2ds12";
        reg = <0x3d>;
        status = "okay";
    };
    lis2ds12N2: lis2ds12@2d {
        compatible = "st,lis2ds12";
        reg = <0x2d>;
        status = "okay";
    };
};
```

Once the sensors are registered, the function `provider_reader_hdr` reads the headers. This action is achieved in the following lines:

```
provider_reader_hdr(&pr, hdr);

for (int i = 0; i < pr.dst_N; ++i) {
    if (i > 0) {
        printk(",");
    }
```

(continues on next page)

(continued from previous page)

```
    printk("%u_%u", hdr[i].provider_id, hdr[i].chan);
}
```

Now that the headers are in place, the values can be obtained. The next loop reads the current values of all sensors' channels into the buffer:

```
provider_reader_read_all(&pr, buffer);

/* Print out the results */
for (int i = 1; i < pr.dst_N; ++i) {
    if (i > 0) {
        printk(",");
    }
    printk("%f", (double) buffer[i]);
}
```

Before the application can be built, it requires a `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.13)
find_package(Zephyr REQUIRED HINTS $ENV{ZEPHYR_BASE})

project(app LANGUAGES C)

target_sources(app PRIVATE src/main.c)
```

and `prj.conf`:

```
CONFIG_SENSOR=y # enable zephyr sensors
CONFIG_PROVIDER_LIB=y # build our library
CONFIG_LIS2DS12=y # build the drivers for the LIS2DS12 sensor
CONFIG_CBPRINTF_FP_SUPPORT=y
```

To build the project, execute:

```
west build -p -b stm32f746g_disco sensor_app
```

3.2 Simulating on Renode

To simulate the application on Renode, first download Renode. The script for that is provided in the source tree:

```
./scripts/prepare_renode
source ./scripts/activate_renode.sh
```

Sample data has to be provided in order to fed it to the emulated sensors. Your own data can also be provided.

```
curl https://dl.antmicro.com/kenning/datasets/anomaly_detection/minispot.csv -o␣
↪data.csv
```

To run in Renode (the emulation terminates after 5 seconds):

```
python ./scripts/run_renode.py \
    --timeout 5 \
    --frequency 100 \
    --data data.csv
```

You should see a similar output to the one below:

```
Starting Renode simulation. Press CTRL+C to exit.
*** Booting Zephyr OS build v4.2.2-1-g17081cb00713 ***
0_0,0_1,0_2,1_0,1_1,1_2
0.000000,0.000000,-0.008374,0.063409,-0.013160,0.000000
0.000000,-0.004785,-0.008374,-0.210568,-0.014356,0.000000
-0.037088,-0.041874,-0.453439,-0.439082,0.331405,0.001196
0.004785,-0.003589,-0.128016,0.081355,0.124426,-0.027517
0.005982,-0.003589,-0.075373,-0.003589,-0.086141,0.001196
0.001196,0.000000,-0.195015,0.153140,0.340977,-0.032303
-0.004785,0.001196,0.177068,0.076570,-0.080159,-0.105284
-0.017946,-0.022731,-0.124426,-0.362512,-0.118444,0.001196
-0.003589,0.000000,0.044267,-0.013160,-0.095712,0.005982
0.005982,0.001196,-0.017946,-0.028713,-0.057427,0.004785
-0.008374,0.004785,-0.051445,0.038285,-0.028713,0.004785
```

3.3 Using a different sensor

For now, we have been performing an example that is considered default by the `samples/sensors` directory. This section presents how to use a different sensor; namely, ADXL345.

First, modify the `stm32f746g_disco.overlay` file:

```
&i2c1 {
    adi_adxl345N1: adi_adxl345@3d {
        compatible = "adi,adxl345";
        reg = <0x3d>;
        status = "okay";
    };
    adi_adxl345N2: adi_adxl345@2d {
        compatible = "adi,adxl345";
        reg = <0x2d>;
        status = "okay";
    };
};
```

Next, modify the `prj.conf` file:

```
CONFIG_SENSOR=y
CONFIG_PROVIDER_LIB=y
CONFIG_ADXL345=y
CONFIG_CBPRINTF_FP_SUPPORT=y
```

Rebuild the project using the command below:

```
west build -b stm32f746g_disco sensor_app
```

Renode needs to be informed of the specific sensors by passing the `--sensor` flag to `run_renode.py` which has the following fields:

- the name of the sensor which will be used in renode, internally,
- the sensor's renode C# class,
- the peripheral to use,
- and the (simulated) physical address of the sensor. Execute that using the following command:

```
python ./scripts/run_renode.py \
    --timeout 5 \
    --frequency 100 \
    --sensor 'adxl345_1,Sensors.ADXL345,i2c1,0x3d' \
    --sensor 'adxl345_2,Sensors.ADXL345,i2c1,0x2d'
```

This should run the application smoothly using a different sensor. You should see the following output:

```
Starting Renode simulation. Press CTRL+C to exit.
*** Booting Zephyr OS build v4.2.2-1-g17081cb00713 ***
0_0,0_1,0_2,1_0,1_1,1_2
64.662598,57.920525,32.178070,19.613300,38.920143,30.952238
59.759274,22.984335,36.468479,44.436382,69.565926,38.613686
36.162022,7.661445,33.097443,46.581589,69.872383,73.856331
62.210934,41.371803,73.549873,22.064962,14.403517,14.097059
73.549873,14.097059,9.193734,15.935806,61.291561,71.711128
0.000000,14.709975,25.435999,18.081011,48.726791,28.807034
66.501343,21.145590,1.225831,3.064578,57.307610,53.630116
39.533058,34.323277,37.387852,44.742840,51.791370,32.178070
17.161636,3.064578,0.919373,64.356140,25.129539,12.258312
2.451662,47.194504,48.113876,67.420715,46.581589,35.549107
```

CHAPTER

FOUR

REAL-TIME ANOMALY DETECTION

This tutorial assumes that you have a fully working setup of this library. For more information, check the page on *reading sensor data*.

Another purpose of this library is allowing the detection of anomalies from provider readings. The process begins by training a basic supervised anomaly detection model using Kenning.

The provider library can be used in two ways:

- implementing code to gather sensor data and detecting in your own loop (in which case, you will need to handle detection and threshold manually),
- using the callback system where detection, buffering, and sliding windows are implictly defined, with some degree of flexibility.

This tutorial follows an example from `workflows/minispot`, showcasing the first method, as well as showing how to perform the same task using the callback method.

4.1 Training an anomaly detection model

This section provides a walkthrough on embedding a trained anomaly detection model using the provider library, starting with training a simple model up to running a simulation with Renode. For more information on how implement, train, and optimize more complex models, check the Kenning documentation.

In a workspace directory (e.g., `anomaly_app/`), create the Kenning scenario `scenario.yml`:

```
platform:
  type: ZephyrPlatform
  parameters:
    name: stm32f746g_disco
    simulated: True
    zephyr_build_path: ../build
    sensors:
      - i2c1.lis2ds12_1
      - i2c1.lis2ds_12_2
    sensors_frequency: 99.5
    runtime_init_log_msg: "*** Booting Zephyr OS build"
    disable_profiler: true
model_wrapper:
  type: PyTorchGenericClassification
  parameters:
```

(continues on next page)

(continued from previous page)

```
    model_path: ./net.pt
    model_source: ./model.py
    training_batch_size: 512
    num_epochs: 10
dataset:
  type: TabularDataset
  parameters:
    dataset_root: build/data
    dataset_path: https://dl.antmicro.com/kenning/datasets/anomaly_detection/
↪minispot.csv
    cols_x: ["roll", "pitch", "gz", "ax", "ay", "az"]
    col_y: "anomaly"
    window_size: 8
    shuffle_data: False
    download_dataset: True
inference_loop:
  type: AnomalyDetectionInferenceLoop
  parameters:
    inference_limit: 512
optimizers:
- type: TFLiteCompiler
  parameters:
    compiled_model_path: fp32.1.tflite
    inference_input_type: float32
    inference_output_type: float32
    target: default
```

The `model.py` will contain a basic Pytorch ResNet. For now, copy the model from `workflows/minispot/model.py` to the `anomaly_app` directory.

The model is then ready to be trained and optimized:

```
pushd anomaly_app
kenning train --cfg scenario.yml
kenning optimize --cfg scenario.yml
popd anomaly_app
```

You should see the optimized model `fp32.1.tflite` in the same directory.

4.2 Basic inference

After training the model, the next step is setting up an application that will use the model for inference. For the purposes of this tutorial, the anomaly detection model used will be a supervised model with a sigmoid activation function and threshold of `0.5`.

Create the `main.c` as follows:

```
#include <zephyr/kernel.h>

#include <stdio.h>
```

(continues on next page)

(continued from previous page)

```
#include <assert.h>

#include <anomaly_lib/detector.h>
#include <provider_lib/provider.h>
#include <provider_lib/providers/sensor.h>

#include <kenning_inference_lib/core/kenning_protocol.h>
#include <kenning_inference_lib/core/loaders.h>

#define READER_DELAY 100

int main(void)
{
    /* Initialize the provider to read from the sensor */
    static struct provider_reader pr = {0};
    static float buffer[128] = {0};
    static struct provider_hdr_entry hdr[128] = {0};

    provider_reader_register_all_sensor(&pr);
    provider_reader_hdr(&pr, hdr);

    /* Fill in the headers */
    for (int i = 0; i < pr.dst_N; ++i) {
        if (i > 0) {
            printk(",");
        }
        printk(",%u_%u", hdr[i].provider_id, hdr[i].chan);
    }
    printk("\n");

    /* Receive sensor data */
    while (1)
    {
        provider_reader_read_all(&pr, buffer);

        /* Print out the results */
        for (int i = 0; i < pr.dst_N; ++i) {
            if (i > 0) {
                printk(",");
            }
            printk(",%f", (double) buffer[i]);
        }
        printk("\n");

        /* Let us put some delay here */
        k_msleep(READER_DELAY);
    }

    return 0;
```

(continues on next page)

(continued from previous page)

```
}
```

The DTS overlay to be used should be written in `anomaly_app/boards/stm32f46g_disco.overlay` with the following contents:

```
&i2c1 {
    lis2ds12N1: lis2ds12@3d {
        compatible = "st,lis2ds12";
        reg = <0x3d>;
        status = "okay";
    };
    lis2ds12N2: lis2ds12@2d {
        compatible = "st,lis2ds12";
        reg = <0x2d>;
        status = "okay";
    };
};
```

By now, your directory structure should be similar to the following:

```
anomaly_app/
    boards/
        stm32f746g_disco.overlay
    src/
        main.c
    prj.conf
    CMakeLists.txt
    scenario.yml
    fp32.1.tflite
    model.py
    ... <other files>
```

To build the application, use:

```
west build -p -b stm32f746g_disco anomaly_app
```

An example time-series dataset is minispot which used OpenQuadruped to record data. The timestep is considered anomalous when the quadruped stumbles during the simulation. A prepared dataset can be downloaded as a time-series CSV file:

```
curl https://dl.antmicro.com/kenning/datasets/anomaly_detection/minispot.csv -o␣
↪data.csv
```

Renode is required to be able to run the simulation. The next commands download and activate Renode:

```
./scripts/prepare_renode.sh
source ./scripts/activate_renode.sh
```

Now that the sensor setup and dataset preparation are complete, the anomaly detection logic can be integrated in the main processing loop:

```
/* In the defines portion of the application */
#define ANOMALY_THRESHOLD 0.5


/* More code here */

detector_init();

while (1)
{
    /* Read sensor data here */

    float score;
    detector_detect(buffer, &score);

    if (score > ANOMALY_THRESHOLD)
    {
        printf("Detected anomaly with confidence: %f\n", (double) score);
    }

    /* Delay here */
    k_msleep(READER_DELAY);
}
```

However, more libraries have to be enabled for the inference. In `prj.conf`, add the following:

```
# KENNING_ML_RUNTIME_TFLITE requires C++
CONFIG_CPP=y
CONFIG_STD_CPP17=y

# Enable the library for anomaly detection
CONFIG_ANOMALY_LIB=y
# Enable the zephyr library for loading models and running inference
CONFIG_KENNING_INFERENCE_LIB=y
# kenning_inference_lib will use tflite
CONFIG_KENNING_ML_RUNTIME_TFLITE=y
# Size in kb of the buffer for the tensor arena allocator
CONFIG_KENNING_TFLITE_BUFFER_SIZE=128
# Use kenning as the backend for anomaly_lib.
CONFIG_ANOMALY_LIB_BACKEND_KENNING=y
# The input to the model is larger than the default input count.
# Let us give a larger number to accomodate the model's input size
CONFIG_ANOMALY_LIB_KENNING_MAX_INPUT_COUNT=256
```

The following command will build the application alongside the compiled tflite model:

```
west build -p -b stm32f746g_disco workspace -- \
    -DCONFIG_KENNING_MODEL_PATH=\"$(realpath ./workspace/fp32.1.tflite)\"
```

Assuming that `LIS2DS12` sensors are used, run the application using Renode with the following command:

```
python ./scripts/run_renode.py \
    --timeout 5 \
    --frequency 100 \
    --data data.csv
```

You should see an output similar to the one below.

```
Starting Renode simulation. Press CTRL+C to exit.
*** Booting Zephyr OS build v4.2.2-1-g17081cb00713 ***
0_0,0_1,0_2,1_0,1_1,1_2
0.000000,0.001196,-0.004785,0.129212,-0.022731,-0.003589,0.628548
Detected anomaly with confidence: 0.628548
0.000000,0.000000,-0.008374,0.110069,-0.017946,0.000000,0.884736
Detected anomaly with confidence: 0.884736
0.000000,0.000000,-0.008374,0.110069,-0.017946,0.000000,0.949333
Detected anomaly with confidence: 0.949333
0.000000,0.000000,-0.008374,0.086141,-0.017946,0.000000,0.942706
Detected anomaly with confidence: 0.942706
0.000000,0.001196,-0.004785,0.009571,-0.013160,0.000000,0.948326
Detected anomaly with confidence: 0.948326
0.000000,0.000000,-0.004785,-0.070588,-0.017946,0.001196,0.958815
Detected anomaly with confidence: 0.958815
0.000000,0.000000,-0.004785,-0.185443,-0.017946,0.001196,0.902647
```

The anomaly detection application is complete.

4.3 Using Callbacks

In some applications, the existing control flow and processing logic may make direct integration of the anomaly detection loop impractical. For example, you may prefer to focus on core application functionality rather than managing anomaly detection explicitly within the application logic.

To address these scenarios, you may use the callback mechanism to run anomaly detection asynchronously. This approach allows the application to react to detected anomalies through user-defined callbacks without the primary control flow. To address these scenarios, the callback mechanism enables anomaly detection to run asynchronously. This approach allows the application to react to detected anomalies through user-defined callbacks while keeping the main application logic unchanged.

The following example demonstrates a callback-based anomaly detection workflow. Here, the main loop represents the primary application logic, while anomaly detection is executed asynchronously through the callback mechanism.

This approach allows anomaly detection to operate independently of the application's main logic.

```
#include <zephyr/kernel.h>

#include <stdio.h>
#include <assert.h>

```

(continues on next page)

(continued from previous page)

```c
#include <anomaly_lib/detector.h>
#include <provider_lib/provider.h>
#include <provider_lib/providers/sensor.h>

#include <kenning_inference_lib/core/kenning_protocol.h>
#include <kenning_inference_lib/core/loaders.h>

#define READER_DELAY 100
#define CALLBACK_PRIORITY 6

static void log_score_cb(void *ctx, float score);

int main(void)
{
    /* Initialize the provider to read from the sensor */
    static struct provider_reader pr = {0};
    static float buffer[128] = {0};
    static struct provider_hdr_entry hdr[128] = {0};

    provider_reader_register_all_sensor(&pr);
    provider_reader_hdr(&pr, hdr);

    detector_init();

    /* Context for the callback */
    struct detector_classifier dc = {0};

    detector_classifier_init(
        &g_detector,              /* Global detector object (initialized by␣
↪detector_init()) */
        &dc,                      /* The detector context */
        &pr,                      /* The provider reader */
        DETECTOR_SMOOTHING_NONE,  /* Smoothing method */
        100,                      /* Detector processing delay in milliseconds */
        0.5                       /* Threshold */
    );

    /* Registering a callback that activates when an anomaly is detected. */
    detector_classifier_register_cb(
        &dc,           /* The detector context */
        log_score_cb,  /* Name of the callback */
        NULL           /* Opaque context specific to the callback */
    );

    /* Starting the detection loop */
    detector_classifier_start(&dc, CALLBACK_PRIORITY, DETECTOR_DEFAULT_OPTS);

    while (1)
    {
```

(continues on next page)

(continued from previous page)

```
        /* Complex application-specific code */
        k_msleep(1000);
    }

    detector_classifier_deinit(&dc);

    return 0;
}

/* Function that gets called when an anomaly is detected */
static void log_score_cb(void *ctx, float score)
{
    (void) ctx;
    /* Logging the anomaly */
    printk("Anomaly detected with confidence: %.4f\n", (double) score);
}
```

You may pass up to `ANOMALY_LIB_DETECTION_CALLBACKS_MAX_CB_HDLRS` for every `detector_classifier` object. By default, you can pass up to `8` callbacks that will be called sequentially everytime an anomaly is detected. Increasing the `MAIN_STACK_SIZE` is necessary; otherwise, the kernel will quickly run out of memory. See here for further details. Furthermore, `CONFIG_ANOMALY_LIB_DETECTION_CALLBACKS` enables the callback mechanism:

```
CONFIG_ANOMALY_LIB_DETECTION_CALLBACKS=y
CONFIG_MAIN_STACK_SIZE=4096
```

Rebuild the application and run again with Renode:

```
west build -p -b stm32f746g_disco workspace -- \
    -DCONFIG_KENNING_MODEL_PATH=\"(realpath ./workspace/fp32.1.tflite)\"

python ./scripts/run_renode.py --timeout 5 --frequency 20 --data data.csv
```

You should see something similar to:

```
Starting Renode simulation. Press CTRL+C to exit.
*** Booting Zephyr OS build v4.2.2-1-g17081cb00713 ***
Anomaly detected with confidence: 0.6285
Anomaly detected with confidence: 0.9971
Anomaly detected with confidence: 0.5771
Anomaly detected with confidence: 0.5232
Anomaly detected with confidence: 0.9755
```

4.4 Evaluating the trained model with Kenning

The evaluation of the accuracy of the model within the application is supported with the use of the Kenning Protocol. After training and optimizing the model, one can generate a report with Kenning which includes the following:

- confusion matrix,

- inference quality metrics,
- detection rate (number of anomalies detected),
- false alarm rate,
- detection delay depending on detection threshold.

The Kenning scenario has the following YAML block:

```yaml
platform:
  type: ZephyrPlatform
  parameters:
    name: stm32f746g_disco
    simulated: True
    zephyr_build_path: <path/to/workspace>/build
    sensors:
      - i2c1.lis2ds12_1
      - i2c1.lis2ds_12_2
    sensors_frequency: 99.5
    runtime_init_log_msg: "*** Booting Zephyr OS build"
    disable_profiler: true
```

This dictates how the model get tested using Renode. Once you have a trained model, you can begin the test with the following command:

```
cd workspace
kenning test report \
    --report-path reports/anomaly.md \
    --to-html \
    --cfg scenario.yml \
    --measurements results.json
```

Opening the report at `reports/anomaly/report.html` should show a complete report of the model performance. You can also see an example real-time anomaly detection evaluation report *here*.

Check out the kenning docs for more information on generating reports with kenning.

CHAPTER

FIVE

API DOCUMENTATION

5.1 Provider lib

Public API for the provider library. Details on how to use this can be found *here*.

Enums

enum **provider_status**

Values:

enumerator **PROVIDER_STATUS_OK**

enumerator **PROVIDER_STATUS_ERROR**

enumerator **PROVIDER_STATUS_SAMPLE_FETCH_ERR**

enumerator **PROVIDER_STATUS_CHAN_GET_ERR**

enumerator **PROVIDER_STATUS_OOB_ERR**

Functions

int **provider_reader_register**(struct *provider_reader* *pr, struct *provider* *p)

Registers a single provider in the provider reader and increments `dst_N`.

Usually called from helpers like `provider_reader_register_all_sensor`.

int **provider_reader_read_all**(struct *provider_reader* *pr, float *dst)

Reads from all providers in the provider reader using `read_fn` and collates results into a float buffer.

`dst` size must be greater or equal than `dst_N`.

int **provider_reader_hdr**(struct *provider_reader* *pr, struct *provider_hdr_entry* *hdr)

Generates a header containing information on provider id and channel of every element in the float buffer returned by `provider_reader_read_all`.

`hdr` size must be greater or equal than `dst_N`. Header doesn't change between reads and should be generated only after provider registration.

struct **provider**

#include <provider.h>

Public Members

enum sensor_channel ***channels**

uint32_t **channels_N**

int (***read_fn**)(struct *provider**, float*)

void ***info**

struct **provider_reader**

#include <provider.h>

Public Members

struct *provider* ***ps**[CONFIG_MAX_PROVIDER_COUNT]

uint32_t **ps_N**

uint32_t **dst_N**

struct **provider_hdr_entry**

#include <provider.h>

Public Members

uint32_t **provider_id**

enum sensor_channel **chan**

5.2 Anomaly lib

Anomaly lib documentation. Details on usage are found *here*.

Defines

DETECTOR_SMOOTHING_NONE

No smoothing used for anomaly detection.

DETECTOR_SMOOTHING_EXP_SMOOTHING

Apply exponential moving average smoothing. Ensure to set the smoothing factor:

```
struct detector_classifier dc;
detector_classifier_init(&dc, ...);
dc.exp_moving_avg_opts.smoothing_factor = 0.3f;
```

DETECTOR_DEFAULT_OPTS

Typedefs

typedef void (***detector_cb_t**)(void *ctx, float score)

Callback type when classifier threshold is met.

Param ctx
Opaque context passed by caller.

Param score
Logit output of the classifier.

typedef float (***detector_smoothing_cb_t**)(struct *detector_classifier* *dc, float score, void *ctx)

Type for the smoothing function.

Enums

enum **detector_status**

Values:

enumerator **DETECTOR_STATUS_OK**

enumerator **DETECTOR_STATUS_ERROR**

enumerator **DETECTOR_STATUS_SIZE_ERR**

enumerator **DETECTOR_STATUS_EINVAL**

enumerator **DETECTOR_STATUS_ERANGE**

Functions

float **detector_smoothing_exp_smoothing**(struct *detector_classifier* **dc*, float *score*, void **ctx*)

static inline int32_t **detector_detect**(const float **buffer*, float **score*)

Calculates anomaly score of provided data

`buffer` size depends on the detector. `score` is a single float.

static inline int32_t **detector_init**()

static inline int32_t **detector_get_info**(void *_info_)

int32_t **detector_classifier_init**(struct _detector_ *_d_, struct _detector_classifier_ *_dc_, struct _provider_reader_ *_pr_, _detector_smoothing_cb_t_ _smoothing_, int64_t _processing_delay_, float _threshold_)

Initialize the detector classifier struct.

Parameters

struct _detector_ *d
Initialized detector

struct _detector_classifier_ *dc
Uninitialized detector classifier

struct _provider_reader_ *pr
Pointer to the initialized provider_reader object

_detector_smoothing_cb_t_ smoothing
Function used for smoothing scores. Possible values are:

- DETECTOR_SMOOTHING_NONE - no smoothing.
- DETECTOR_SMOOTHING_EXP_SMOOTHING - exponential smoothing.

int64_t processing_delay
Cadence for obtaining new values from the detector.

float threshold
Probability threshold in (0, 1) to determine whether a score is anomaly or not.

int32_t **detector_classifier_start**(struct _detector_classifier_ *_dc_, int _priority_, int _thread_opts_)

Start the callback handler in the background.

Parameters

struct _detector_classifier_ *dc
Initialized classifier object

int priority
Priority given to thread.

int thread_opts
Pass `DETECTOR_DEFAULT_OPTS` for the default settings. Otherwise, this contains standard Zephyr thread options.

int32_t **detector_classifier_deinit**(struct _detector_classifier_ *_dc_)

Stop the callback handler and cleanup

Parameters

struct _detector_classifier_ *dc
classifier object

int32_t **detector_classifier_register_cb**(struct *detector_classifier* **dc*, *detector_cb_t cb*, void **ctx*)

Register a callback that is run when an anomaly is detected.

Parameters

struct *detector_classifier* *dc
The detector object.

***detector_cb_t* cb**
Callback function

void *ctx
Opaque context passed to this specific callback.

void **detector_classifier_unregister_cb**(struct *detector_classifier* **dc*)

Unregister all callbacks

Parameters

struct *detector_classifier* *dc
The detector object

Variables

struct *detector* **g_detector**

struct **detector**

#include <detector.h>

Public Members

int32_t (***detect**)(struct *detector**, const float*, float*)

int32_t (***init**)(struct *detector**)

int32_t (***get_info**)(struct *detector**, void**)

struct **detector_smoothing_opts_exp_mav**

#include <detector.h>

Public Members

float **smoothing_factor**

Value between 0 and 1.

struct **detector_classifier**

#include <detector.h> Callback handler manager for anomaly detection.

Public Members

struct *detector* ***detector**

Detector object for this specific classifier

float **threshold**

Threshold for the anomaly detection

int64_t **process_ms**

Expected processing time

size_t **num_cbs**

Number of registered callbacks

void ***cb_ctxs**[CONFIG_ANOMALY_LIB_DETECTION_CALLBACKS_MAX_CB_HDLRS]

detector_smoothing_cb_t **smoothing**

Smoothing function

float **smoothed_moving_average**

Current value of the moving average

bool **smoothing_started**

Indicator whether smoothing started. Used to prevent initializing `smoothing_score` to some absurd number such as 0 or NaN.

struct *detector_smoothing_opts_exp_mav* **exp_moving_avg_opts**

Options when `DETECTOR_SMOOTHING_EXP_SMOOTHING` is used

union detector_classifier

Private Members

struct *provider_reader* ***pr**

Provider reader object

atomic_t **deinit_started**

started deinitialization process

struct k_work_q **wq**

internal work queue

float **buffer**[128]

internal buffer for the provider

struct *provider_hdr_entry* **hdr**[128]
: internal header buffer

struct k_thread **thread_data**
: thread data

k_tid_t **tid**
: thread ID

detector_cb_t
cb_hdlrs[CONFIG_ANOMALY_LIB_DETECTION_CALLBACKS_MAX_CB_HDLRS]
: Struct containing the registered callbacks

CHAPTER

SIX

SAMPLE REAL-TIME ANOMALY DETECTION EVALUATION REPORT

This section contains a sample report, with real-time anomaly detection evaluation metrics, generated during CI.

The CI is set up as follows:

- *Environment*: Github Actions
- *Task*: Anomaly Detection on minispot dataset
- *ML training framework*: PyTorch, ran with Kenning's generic CNN model wrapper
- *ML execution framework*: `TFLite Micro`, running under `kenning-inference-lib` from Kenning Zephyr Runtime,
- *Platform*: `stm32f746g_disco` board with two I2C accelerometers

6.1 Real-time model evaluation, with time-based metrics

6.1.1 Commands used

Note

This section was generated using:

```
kenning test \
    --cfg \
        scenario.yml \
    --measurements \
        results.json

kenning report \
    --report-name \
        Real-time model evaluation, with time-based metrics \
    --report-path \
        /home/runner/work/zephyr-sensor-anomalies/zephyr-sensor-anomalies/docs/
↪source/generated/sample-real-time-evaluation-report/report.md \
    --root-dir \
        /home/runner/work/zephyr-sensor-anomalies/zephyr-sensor-anomalies/docs/
↪source \
    --img-dir \
        /home/runner/work/zephyr-sensor-anomalies/zephyr-sensor-anomalies/docs/
↪source/generated/sample-real-time-evaluation-report/img \
```

```
    --measurements \
        results.json \
    --smaller-header
```

6.1.2 General information for results.json

Model framework:

- torch ver. 2.8.0+cu128

Input JSON:


```
{
    "dataset": {
        "type": "kenning.datasets.tabular_dataset.TabularDataset",
        "parameters": {
            "dataset_path": "https://dl.antmicro.com/kenning/datasets/anomaly_
↪detection/minispot.csv",
            "cols_x": [
                "roll",
                "pitch",
                "gz",
                "ax",
                "ay",
                "az"
            ],
            "col_y": "anomaly",
            "window_size": 8,
            "shuffle_data": false,
            "expand_classes": true,
            "dataset_root": "build/data",
            "inference_batch_size": 1,
            "download_dataset": true,
            "force_download_dataset": false,
            "external_calibration_dataset": null,
            "split_fraction_test": 0.2,
            "split_fraction_val": null,
            "split_seed": 1234,
            "reduce_dataset": 1.0
        }
    },
    "dataconverter": {
        "type": "kenning.dataconverters.modelwrapper_dataconverter.
↪ModelWrapperDataConverter",
        "parameters": {}
    },
    "optimizers": [
        {
            "type": "kenning.optimizers.tflite.TFLiteCompiler",
            "parameters": {
                "model_framework": "any",
```


(continues on next page)

(continued from previous page)


```
                "target": "default",
                "inference_input_type": "float32",
                "inference_output_type": "float32",
                "dataset_percentage": 0.25,
                "quantization_aware_training": false,
                "use_tf_select_ops": false,
                "resolver_template_path": null,
                "resolver_output_path": null,
                "epochs": 3,
                "batch_size": 32,
                "optimizer": "adam",
                "disable_from_logits": false,
                "save_to_zip": false,
                "compiled_model_path": "build/fp32.1.tflite",
                "location": "host"
            }
        }
    ],
    "platform": {
        "type": "kenning.platforms.zephyr.ZephyrPlatform",
        "parameters": {
            "zephyr_build_path": "../../build",
            "llext_binary_path": null,
            "sensors": [
                "i2c1.lis2ds12_1",
                "i2c1.lis2ds12_2"
            ],
            "sensors_frequency": 99.5,
            "enable_zephelin_gdb": false,
            "enable_zephelin": false,
            "zephyr_base": null,
            "uart_port": "/tmp/renode_uart_mlwydezj/uart",
            "uart_baudrate": 115200,
            "uart_log_port": "/tmp/renode_uart_mlwydezj/uart_log",
            "uart_log_baudrate": 115200,
            "auto_flash": false,
            "openocd_path": "openocd",
            "sensor": null,
            "number_of_batches": 16,
            "simulated": true,
            "runtime_binary_path": null,
            "platform_resc_path": "gh://antmicro:kenning-zephyr-runtime/renode/
↪scripts/stm32f746g_disco.resc;branch=main",
            "resc_dependencies": [],
            "post_start_commands": [],
            "disable_opcode_counters": false,
            "disable_profiler": true,
            "profiler_dump_path": null,
            "profiler_interval_step": 10.0,
```


(continues on next page)

(continued from previous page)

```
            "runtime_init_log_msg": "*** Booting Zephyr OS build",
            "runtime_init_timeout": 30,
            "gdb_port": 3333,
            "name": "stm32f746g_disco",
            "platforms_definitions": [
                "kenning:///platforms/platforms.yml",
                "/home/runner/work/zephyr-sensor-anomalies/zephyr-sensor-
↪anomalies/.venv/lib/python3.11/site-packages/kenning/resources/platforms/
↪platforms.yml"
            ]
        }
    },
    "protocol": {
        "type": "kenning.protocols.uart.UARTProtocol",
        "parameters": {
            "port": "/tmp/renode_uart_mlwydezj/uart",
            "baudrate": 115200,
            "error_recovery": true,
            "timeout": 30
        }
    },
    "model_wrapper": {
        "type": "kenning.modelwrappers.classification.pytorch_generic.
↪PyTorchGenericClassification",
        "parameters": {
            "training_batch_size": 512,
            "model_source": "./model.py",
            "learning_rate": 0.01,
            "num_epochs": 10,
            "export_dict": true,
            "model_path": "./net.pt",
            "model_name": null
        }
    },
    "runtime": {
        "type": "kenning.runtimes.tflite.TFLiteRuntime",
        "parameters": {
            "save_model_path": "build/fp32.1.tflite",
            "delegates_list": null,
            "num_threads": 4,
            "llext_binary_path": null,
            "batch_size": 1,
            "disable_performance_measurements": false
        }
    },
    "inference_loop": {
        "type": "kenning.inferenceloops.anomaly_realtime.
↪AnomalyDetectionInferenceLoop",
        "parameters": {
```

(continues on next page)

(continued from previous page)

```
            "smoothing_window_size": 10,
            "inference_limit": 512
        }
    }
}
```

6.2 Inference quality metrics for results.json

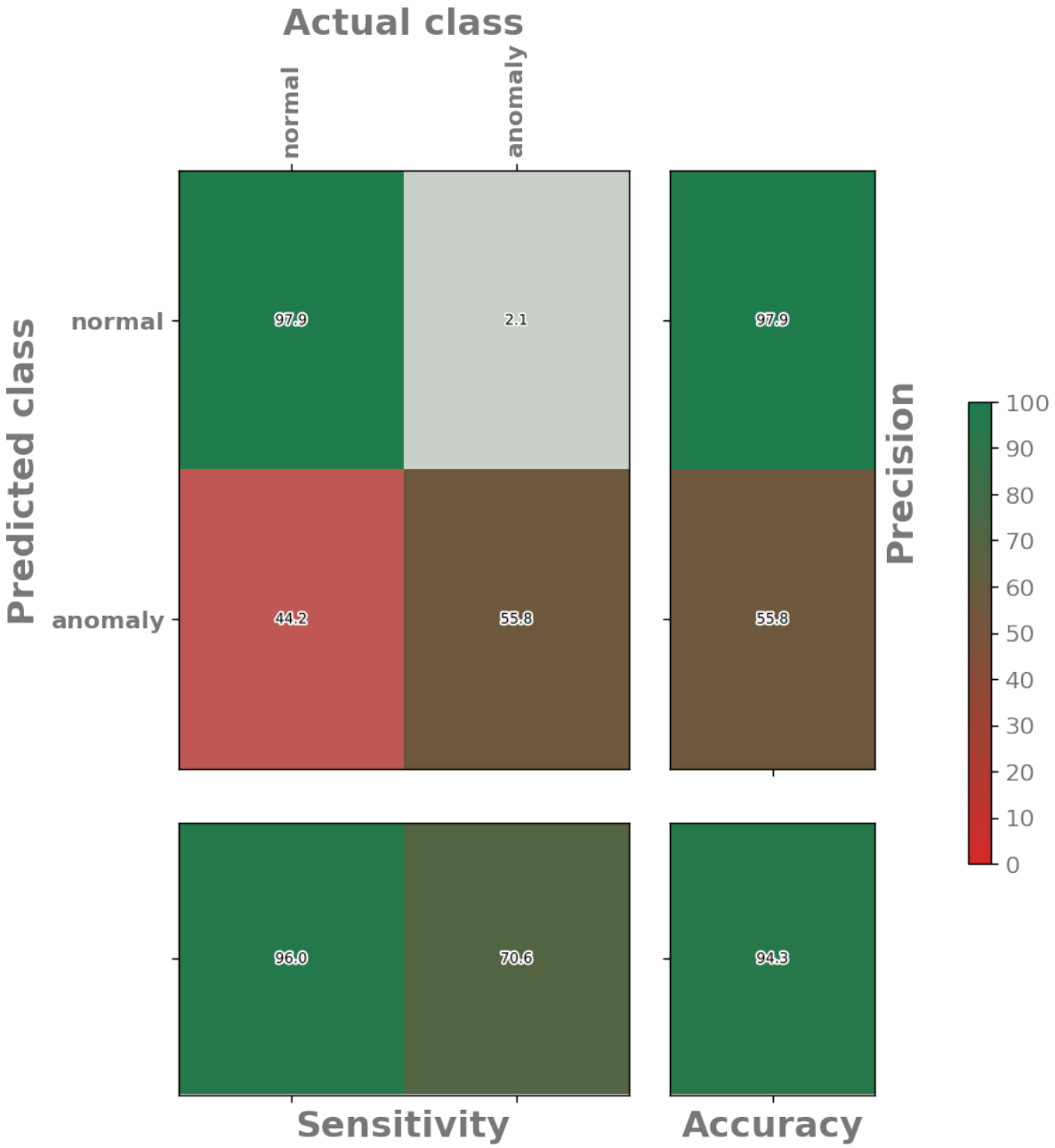

Fig. 6.1: Confusion matrix

Table 6.1: Inference quality metrics

Statistic	Value
Accuracy	**0.943249**
Mean precision	**0.768386**
Mean sensitivity	**0.833025**
G-mean	**0.823265**

6.3 Detection quality metrics for results.json

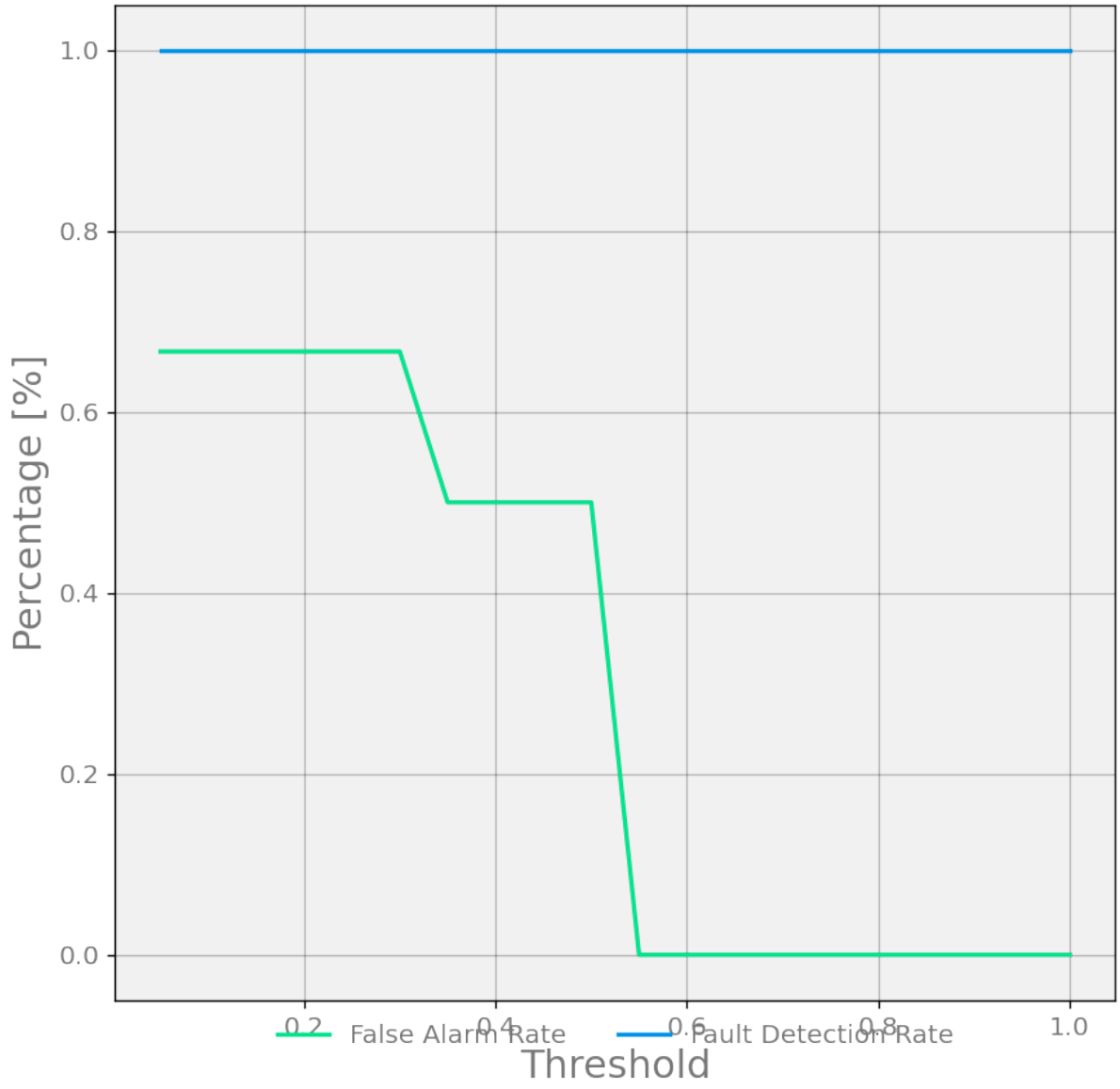

Fig. 6.2: Anomaly Fault Detection Rate and False Alarm Rate plot

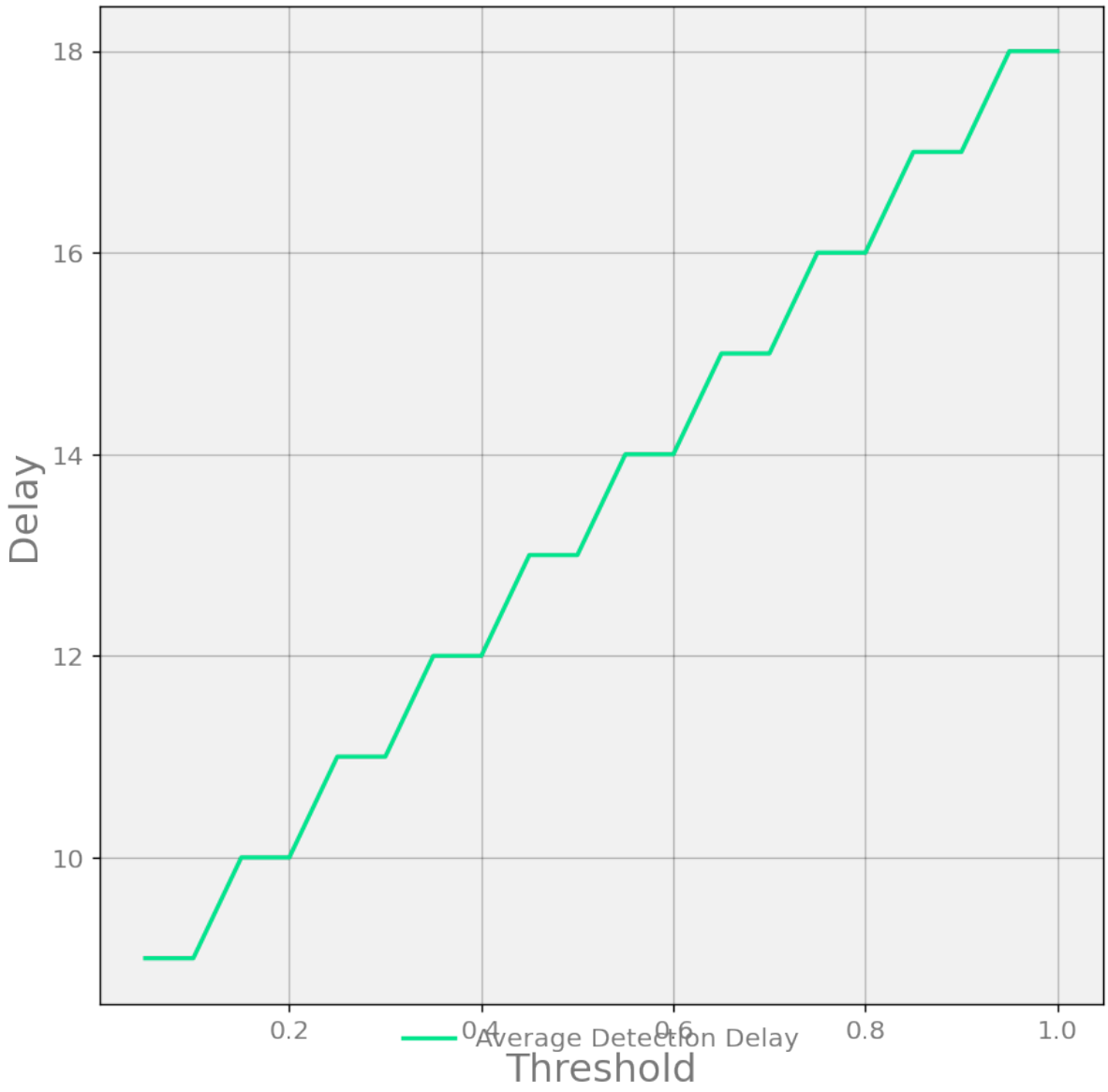

Fig. 6.3: Anomaly Average Detection Delay plot

INDEX

D

G

P