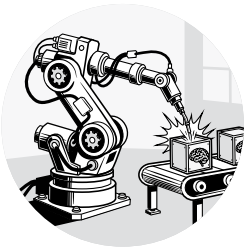
9

SENSOR MACHINE LEARNING

The greatest enemy of knowledge is not ignorance; it is the illusion of knowledge.
—Stephen Hawking



This is the final frontier! There is nothing between the sensor and the real world, and it doesn't get any edgier than this. If you thought that microprocessors were resource-constrained, wait until you try to run machine learning algorithms on a sensor. You know somewhere a developer is attempting to port *DOOM* to a sensor, just to see if they can.

Before exploring what you can achieve with sensor ML, we need to explain how we ended up here. This chapter provides a brief recap of sensor evolution and explains why some of the methods you have already used to process sensor data are not always sufficient. You will learn about the common sensor ML algorithms and build a project that uses these techniques to raise an alert when it detects an anomaly in the movement of a robot arm.

The Rise of Sensor Intelligence

For a long time, sensors were the silent partners of embedded systems. A thermocouple crossed a threshold; a relay toggled. An accelerometer spiked; an alarm fired. The logic was simple, deterministic, and hardwired. The sensor itself had no opinion about what was happening. You explored the evolution of smart sensors in Chapter 6.

Then, the data got harder. An IMU tracking human motion produces continuous, high-frequency output across three axes. A gas sensor responding to volatile organic compounds in a noisy industrial environment generates unpredictable signals. Processing this kind of data in real time, at the edge, exposed the limits of hard-coded logic and simple filters.

The first serious response was sensor fusion. By combining data from multiple sources, such as accelerometers, gyroscopes, and magnetometers, it became possible to produce more accurate estimates of orientation and movement than from any single sensor alone. But while fusion improved accuracy and resilience, it remained fundamentally a statistical process governed by static models. What if the patterns in the sensor data weren't just noise or drift but signals that could reveal deeper insights about user intent, device health, environmental conditions, or even human behavior?

This is where sensor ML steps in. Instead of programming a device to react to predefined thresholds or rules, you enable it to learn from data. The embedded system no longer simply fuses inputs; it recognizes patterns, predicts outcomes, and adapts to context. Machine learning allows sensor-rich systems to classify activities, detect anomalies, anticipate failures, and personalize responses.

Fitness trackers can count steps using a crude threshold on acceleration data. But when the same sensor data is fed through a trained ML model, the device can distinguish between walking, running, cycling, climbing stairs, and even dancing—all using the same accelerometer. The intelligence lies in the pattern of movement, not the absolute values. And because the model is trained from real-world examples, it can adapt to the nuances of each user.

The same principle applies in industrial settings. A vibration sensor mounted on a motor might register subtle shifts in amplitude or frequency long before a mechanical fault is visible. A traditional system would miss these changes as insignificant or noisy. But an ML model trained on historical failure data can learn to detect the early signatures of bearing wear or misalignment, enabling intervention before catastrophic failure. In both cases, the sensor system shifts from passive observation to active interpretation, processing raw data into actionable information at the edge, without cloud connectivity or human oversight.

This chapter explores how sensor ML builds on the foundations of sensor fusion and embedded AI to deliver intelligent behavior. It explains the workflow required to train and deploy models on embedded platforms, the types of problems that sensor ML is particularly suited for, and the practical constraints that must be addressed. Finally, you will construct a project

that can recognize movement anomalies in a robot arm, demonstrating how data collected from sensors can become a source of embedded intelligence.

Why Traditional Methods Are Insufficient

Classical signal processing and sensor fusion techniques such as Kalman filters, complementary filters, and statistical thresholding have long served as the foundation of embedded sensing. These methods work well enough in controlled environments where sensor noise is minimal, relationships between variables are linear, and system dynamics are well understood. However, these methods depend on assumptions that increasingly break down as systems grow more complex, more autonomous, and more exposed to real-world variability.

At the core of traditional methods lies a dependency on models that must be defined explicitly. As seen in the previous chapter, Kalman filters require a priori knowledge of the system's state-space model, including accurate noise covariance matrices and transition dynamics. In practice, this level of precision is rarely achievable. Sensors are subjected to varying temperatures, humidity, and electromagnetic interference. Their outputs may degrade over time or respond nonlinearly to physical inputs. Under such conditions, handcrafted models often fail to represent the true behavior of the system, leading to poor estimation accuracy or instability.

Figure 9-1 illustrates this concept. A Kalman filter tuned for Gaussian noise performs well under those conditions but deteriorates sharply when the noise profile changes to impulse noise. An ML-based approach, trained on diverse noise conditions, generalizes more effectively and tracks the true signal more closely.

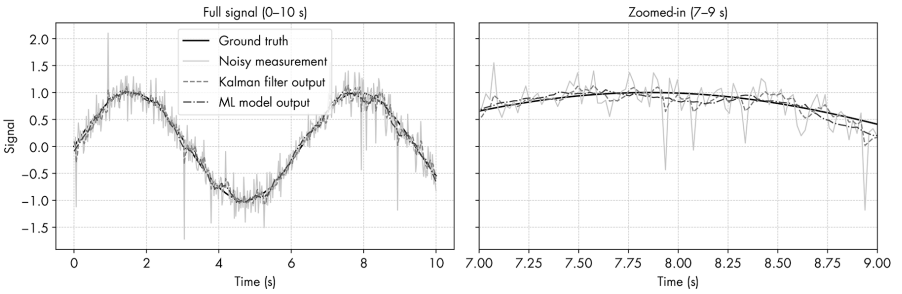


Figure 9-1: A plot illustrating that a Kalman filter optimized for Gaussian noise struggles with impulse noise, while an ML model can adapt

Think about the challenge of controlling an autonomous drone through a dense urban environment. A classical approach would combine IMU data with GPS and barometric pressure, filtered through a sensor fusion algorithm. But what happens when GPS signals become unreliable due to multipath reflections or occlusion? Or when rapid movements, airframe damage, and vibrations introduce nonlinearities that the filter

cannot model? Traditional algorithms struggle in these edge cases, requiring excessive tuning or additional failsafe logic that increases complexity without guaranteeing robustness.

Replace a sensor on a production line, and your carefully calibrated Kalman filter may need to be retuned from scratch. Change the operating environment, and the noise covariance matrices become invalid. Traditional algorithms do not adapt; they must be explicitly recalibrated. ML models sidestep this problem by learning directly from data. They capture the statistical structure of the signal, not the calibration assumptions of a specific deployment, and can be retrained when conditions change.

Traditional methods have an undeniable advantage in simplicity: A Kalman filter is fast and uses kilobytes of RAM. But for problems where those methods produce inadequate results, the alternative has historically been to move computation off-device. Techniques such as quantization, pruning, and knowledge distillation have changed this. They allow ML models to run on microcontrollers at inference costs that, while higher than a simple filter, remain within the power and latency budgets of battery-operated edge devices.

Machine Learning Models Commonly Used with Sensors

So which models work at the sensor layer? The answer depends on the data, the task, and the hardware budget. Here are some of the workhorses.

When interpretability and fast inference matter more than raw accuracy, tree-based models are a good place to start. Decision trees are simple to deploy and inspect. Their ensemble variants—random forests and gradient-boosted trees (such as XGBoost or LightGBM)—improve accuracy while retaining much of that transparency.

You could deploy random forest on an agricultural sensor node to classify soil conditions based on moisture, temperature, and pH readings. LightGBM has been used in smart irrigation systems to learn watering schedules from historical weather and soil sensor data, reducing water use while maintaining yield.

For binary classification tasks, particularly anomaly detection, support vector machines (SVMs) remain effective. You first met these in Chapter 3. SVMs are computationally light at inference time, which makes them attractive for resource-constrained deployments where the classification boundary is well defined. For example, in industrial condition monitoring, SVMs have been applied to vibration and acoustic sensor data to detect early signs of bearing failure. A linear SVM can be used on time-domain features extracted from accelerometer signals to identify machine faults with high precision.

For problems involving time-series data or sequential dependencies, recurrent neural networks (RNNs) and their more robust variants, such as long short-term memory (LSTM) networks, are a natural fit (see Chapter 4 for a deeper treatment of neural network architectures). These models learn temporal patterns, making them ideal for motion recognition, speech

detection, or battery state-of-charge estimation. A practical example is wearable activity recognition, where LSTMs are trained on IMU data to classify activities such as walking, running, or cycling. In one healthcare project, an LSTM was deployed on an Arm Cortex-M4 microcontroller to monitor Parkinson's tremors in real time using wrist-worn sensors, enabling patients to track symptom severity outside clinical settings.

More recently, convolutional neural networks (CNNs) have been adapted for sensor data, particularly when signals are represented in the frequency domain or as images. For instance, environmental sound classification tasks often convert microphone signals into spectrograms, allowing CNNs to detect events such as breaking glass, alarms, or animal calls. The Edge Impulse platform has showcased such models running efficiently on microcontrollers like the STM32 series, enabling real-time audio scene understanding on battery-powered edge devices.

Labeled fault data is often scarce. You may have thousands of hours of normal operation recorded but only a handful of actual failures. Autoencoders are designed for exactly this situation. They learn to compress and reconstruct normal sensor behavior. When an input deviates from that learned baseline, the reconstruction error spikes, signaling an anomaly without the need for labeled examples of every possible fault. This technique has been used in smart factories to catch early drift in motor currents or temperature profiles. On embedded hardware, a standard autoencoder with a narrow bottleneck layer is practical to deploy; the more complex variational autoencoder (VAE) adds a probabilistic latent space but is harder to justify given the additional memory and compute overhead.

For low-power, interpretable applications, k-nearest neighbors (KNN) remains a surprisingly effective option. Although memory intensive in its vanilla form, optimizations such as prototype selection and quantized feature storage enable KNN to run in real time on devices like the Arduino Nano 33 BLE Sense. It has been used in gesture recognition by comparing real-time IMU feature vectors against a known library of gestures.

The models surveyed here share a common trait: They have all been successfully deployed on microcontroller-class hardware for real-time sensor tasks. The field is not standing still. Transformer- and attention-based architectures, once confined to the cloud, are being compressed for edge deployment through techniques such as token pruning and linear attention approximations. But for the project that follows, we will use a decision tree, the simplest and most interpretable model in this list, deployed directly on an IMU to detect movement anomalies in a robot arm.

Project 15: Robot Arm Anomaly Detection

Pick-and-place robots are ubiquitous in automated assembly, and unplanned downtime is expensive. In this project, you will build a sensor-based machine learning system that monitors a robot arm's movement and raises an alert when it detects behavior consistent with a developing fault. The test platform is a 6-DOF arm kit driven by six MG996R digital servos

(Figure 9-2). An Arduino UNO sends movement commands to the servos through a Pololu Maestro six-channel servo controller, while an ISM330BX IMU mounted on the arm captures the motion data used for both model training and anomaly detection.

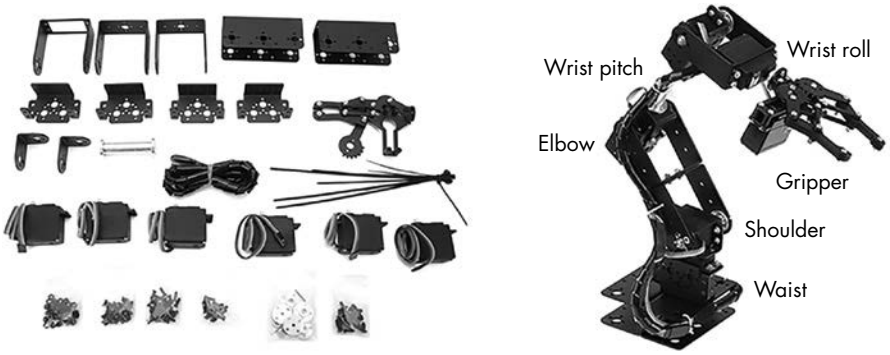


Figure 9-2: A 6-DOF robot arm kit

WHAT IS 6-DOF?

6-DOF stands for *six degrees of freedom*. This refers to the number of independent movements the arm can make in 3D space. Each degree of freedom corresponds to a specific type of motion, either translational (movement along an axis) or rotational (rotation about an axis). This configuration allows a 6-DOF robot arm to position its gripper anywhere within its working envelope and orient it in any direction. The human arm also exhibits six degrees of freedom.

A block diagram of the robot arm control and monitoring system is provided in Figure 9-3. The robot control algorithms will run on the Arduino UNO and control the arm servos using a servo controller. The IMU will be monitored via an ST evaluation kit using ST MEMS-Studio running on a PC.

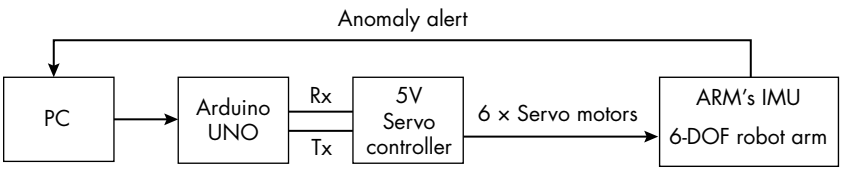


Figure 9-3: An anomaly detection block diagram

What You'll Learn

By the end of this project, you will have built a working anomaly detection system that runs entirely on the sensor, with no inference load on the host microcontroller. Along the way, you'll learn how to:

- Control a multi-servo robotic arm and program repeatable movement sequences.
- Collect and label IMU sensor data for both normal and anomalous operating conditions.
- Train a decision-tree-based model using ST's MEMS-Studio.
- Deploy the trained model to the ISM330BX's machine learning core (MLC), a dedicated hardware block inside the sensor that evaluates decision trees at the IMU's output data rate.

The MLC is what makes this project distinct from a conventional TinyML deployment. The intelligence does not run on the Arduino. It runs on the sensor itself.

Parts List

Apart from the IMU evaluation kit, all the other parts can be substituted with generic equivalents. This is what you will need:

- 6-DOF robot arm kit with six MG996R servos
- Pololu Maestro six-channel servo controller
- Arduino UNO
- ISM330BX IMU evaluation kit (MKI245KA + MKI109V3 motherboard)
- 5 V, 15 A regulated power supply (for example, MEAN WELL LRS-75-5)
- USB cable and serial connection leads
- PC running macOS, Windows, or Linux
- Servo cables and hookup wire

There are several 6-DOF robot arm kits available; do an internet search and look for something like Figure 9-2. The robot arm uses six MG996R high-torque digital servo motors. The key specifications of this servo are provided in Table 9-1.

Table 9-1: MG996R Servo Motor Key Specifications

Specification	Value
Torque	9.4 kg • cm at 4.8 V, 11 kg • cm at 6.0 V
Speed	0.17 s/60 degrees at 4.8 V, 0.14 s/60 degrees at 6.0 V
Operating voltage	4.8 V to 7.2 V

(continued)

Table 9-1: MG996R Servo Motor Key Specifications (*continued*)

Specification	Value
Stall current	2.5 A at 6 V
Running current	500 to 900 mA
Rotation range	Approx. 120 degrees total
Weight	55 g
Dimensions	40.7 × 19.7 × 42.9 mm

To estimate the required capacity of the 5 V power supply for six MG996R servos operating simultaneously, assume a worst-case scenario where all six approach stall conditions during peak load. The stall current is quoted at 2.5 A at 6 V, and for 5 V the estimate is 2.3 A:

$$\text{Total peak current} = 6 \times 2.3 = 13.8 \text{ A}$$

Based on this, we have specified a 5 V servo power supply rated at 15 A to allow some margin for error. The LRS-75-5 (Figure 9-4) that we sourced is a beefy supply and requires you to connect the 110-240 VAC cable, which leaves these terminals exposed! After connection, cover the high-voltage terminals with an insulator.

Figure 9-4: The MEAN WELL LRS-75-5 (5 V, 15 A) power supply

WARNING *High-voltage caution! Use a different supply with the input voltage cable already connected if you are not qualified to connect 110/240 VAC. This is a potentially lethal voltage. A suitably rated bench power supply would be an alternative.*

Tools and Software Setup

Install the Arduino IDE and then the Pololu Maestro and AltSoftSerial libraries using the Library Manager. On your PC, install ST's MEMS-Studio, which provides configuration tools, data logging, visualization, and ML features for ST sensors. In this project, you'll use MEMS-Studio for configuring the ISM330BX sensor parameters, logging IMU data during arm movement, and training the MLC decision tree. The software also includes AlgoBuilder and other analysis tools, but these are not required for this project. Links to the software downloads are provided in Appendix A.

Once installed, connect the MKI109V3 motherboard via USB and confirm that MEMS-Studio detects the ISM330BX. If the sensor appears in the device list, your setup is ready. If not, check that the USB drivers are installed and that the MKI245KA adapter board is properly seated on the motherboard.

WHY THE SOFTWARE SERIAL LIBRARY ISN'T USED

For the Pololu Maestro Arduino library examples, we suggest using SoftwareSerial when controlling with an UNO. Ideally, use an MCU with multiple hardware serial ports, but if you are using an UNO, then use the AltSoftSerial library, which offers several advantages over SoftwareSerial. It provides true full-duplex communication, which means it can transmit and receive simultaneously without needing to switch between listening modes. Unlike SoftwareSerial, which disables interrupts during transmission and can interfere with timing-sensitive tasks, AltSoftSerial uses hardware timers to maintain accurate timing with minimal CPU overhead. As a result, it produces fewer errors and dropped characters, particularly at higher baud rates. The trade-off is that AltSoftSerial works only on fixed pins: transmit on pin 9 and receive on pin 8 for the UNO. The biggest advantage from your perspective is that you can test whether it is working! Because SoftwareSerial is half-duplex, you can't send and receive in a loopback configuration. Try this with the echo example from the book repo (*c09001.ino*) to test serial transmission; connect pin 8 to pin 9 when doing this.

Setup and Configuration

Mount the ISM330BX on the forearm segment near the gripper. This position captures the largest range of motion across the most joints, which produces the most distinctive signatures for anomaly detection. If your cable length doesn't reach the gripper, the elbow joint is an acceptable alternative, but you may need to adjust the MLC decision tree thresholds during training. Use double-sided tape for mounting. The IMU must be firmly attached, as a loose joint will cause errors.

Connect each servo to the correct channel on the Maestro controller using the information in Tables 9-2 and 9-3. The Maestro communicates with the Arduino using serial communication, with wiring as detailed in Figure 9-5. As part of the first step of the build (see "Control the Arm" on page XX), you will verify that each servo moves correctly before continuing.

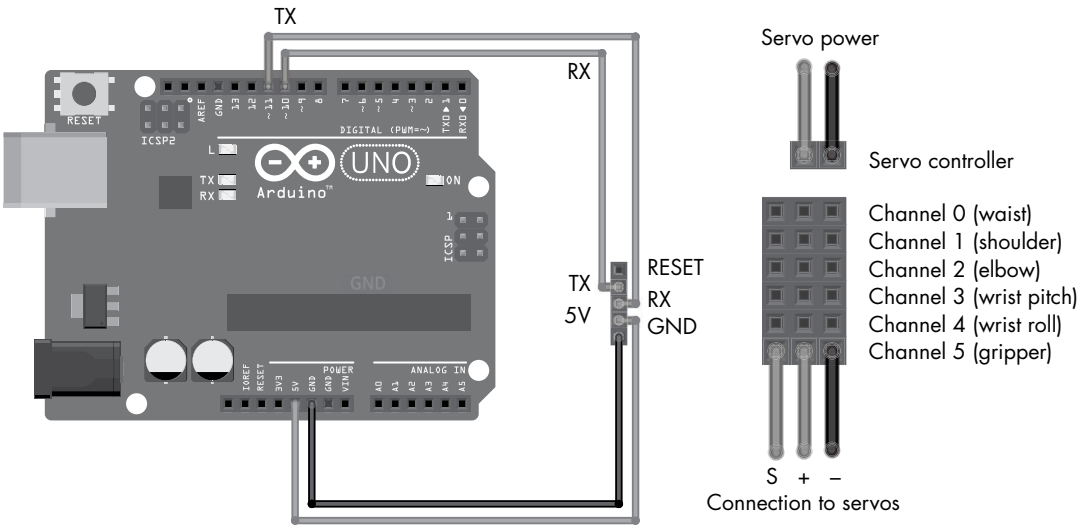


Figure 9-5: Connecting the Arduino UNO to the Maestro servo controller

If you need to troubleshoot individual servos, the Pololu Maestro Control Center software allows you to drive each channel manually from a PC. This can be useful for verifying that a servo responds before writing Arduino code.

Table 9-2: Servo Channel Map and Code Label

Maestro channel	Joint	Code label
0	Base rotation	WAIST
1	Shoulder	SHOULDER
2	Elbow	ELBOW
3	Wrist pitch	WRIST_PITCH
4	Wrist roll	WRIST_ROLL
5	Gripper	GRIPPER

When connecting the arm servos to the controller, use Table 9-3 to identify the correct wires. The MG996R servos generally use the JR convention: orange (signal), red (power), brown (ground).

Table 9-3: Servo Cable Connections by Manufacturer

Manufacturer	Signal (PWM)	V+ (power)	GND (ground)
Futaba	White	Red	Black
JR	Orange	Red	Brown
Hitec	Yellow	Red	Black
TowerPro	Orange	Red	Brown
Spektrum	Orange	Red	Brown

Connect the 5 V, 15 A supply directly to the Maestro's servo power rail. The Maestro's logic can be powered from the Arduino UNO's +5 V pin. Ensure the servo power jumper on the Maestro is removed so that servo power and logic power are on separate rails. Connect all grounds together: the UNO ground, the Maestro ground, and the power supply ground.

There are several moving parts to get right here. The build steps that follow take you through them one at a time: servo control first, then data collection, model training, and on-sensor deployment. Each step includes a verification check so you can confirm that everything works before moving on.

Control the Arm

You need to write some code to control the robot arm. To be more specific, you must write a sketch that sends commands to the Maestro servo controller, which will then drive the servos on the robot arm to the desired position. Sending commands is easy, as Pololu has written an Arduino library for this purpose.

Servo motors, such as the MG996R, are controlled using pulse-width modulation (PWM). A PWM signal consists of a repeating pulse, typically every 20 milliseconds (a frequency of 50 Hz), where the width of the pulse determines the servo position (Figure 9-6).

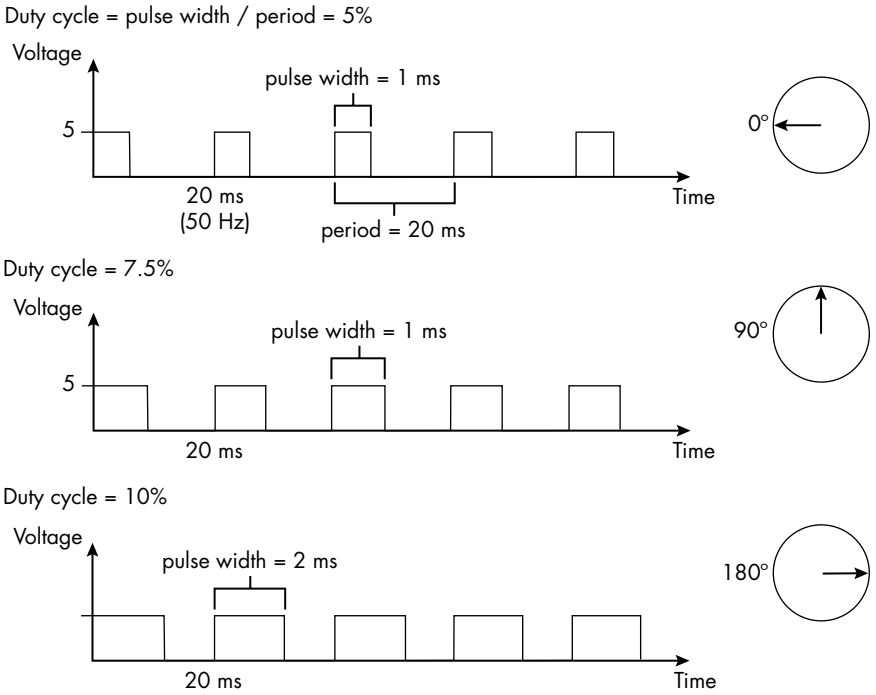


Figure 9-6: Servo control using pulse width to encode position (not to scale)

The Pololu Maestro Servo Controller generates these pulses and can be controlled from an Arduino using the Pololu Maestro library via a serial interface. The Arduino communicates with the Maestro at 9600 baud and with the Serial Monitor at 115200 baud. These are independent serial connections running at different rates. The Maestro sends pulses to the servo pin every 20 ms by default. You can configure the servo pulse period, speed, and acceleration limits via software.

Within the Pololu Maestro library, there is a `setTarget()` command that takes two parameters: the channel and the position. Channel refers to the servo channel number (0–5), and position is the desired pulse width. Typical servos respond to target position values of 4000 to 8000 (Table 9-4).

Table 9-4: Servo Angle Mapping for the Pololu Maestro Servo Controller

Maestro position value	Pulse width (µs)	Servo angle (degrees)
4000	1000	0
6000	1500	90
8000	2000	180

The Maestro represents pulse widths in units of 0.25 µs. A target value of 6000 corresponds to $6000 \times 0.25 = 1500$ µs, which places the servo at its

center position. This allows for very fine control with 0.25 µs resolution, though most servos cannot physically respond to this level of precision. For example, to move a servo in an Arduino sketch, use:

```
// moves servo 0 to its center position (1.5 ms pulse)
servoController.setTarget(0, 6000);
```

The `setTarget()` command sends a position value of 6000 (1500 µs) to servo channel 0. The Maestro then handles timing, pulse generation, and regulation of motion, offloading this real-time task from the Arduino.

Upload test sketch *c09002.ino* from the project repository. This sketch initializes the serial connection to the Maestro, checks for controller errors, and then centers five of the six servos (shoulder, elbow, wrist pitch, wrist roll, and gripper) at position 6000 (90 degrees). For safety, the waist servo is left at its current position. Open the Serial Monitor at 115200 baud. You should see the error code reported as 0 and a position readback for all six channels.

The position readback for channels 1 through 5 should report values close to 6000. Channel 0 (waist) will report whatever position the servo was in at power-up. If any channel reports 0, the Maestro is not receiving feedback from that servo. Check the connection.

Once you have confirmed that all servos center correctly, experiment by changing the target values in the sketch. Try commanding each servo to its minimum (4000) and maximum (8000) positions to verify the full range of motion. Pay attention to any mechanical binding or stalling at the extremes, as not all arm configurations allow a full 180 degrees on every joint.

If the error code is nonzero, consult the Pololu Maestro documentation for the specific error. If a servo does not move to center, verify its channel wiring against Table 9-2 and confirm that the 5 V supply is connected and delivering power under load. A servo that buzzes but does not move typically indicates insufficient current.

Create Movement Routines

In this step, you will write three sketches that generate the arm movements used during data collection: one for a stationary baseline, one for normal pick-and-place operation, and one that introduces simulated anomalies. These routines run on the actual hardware because the training data must reflect the real vibration, timing, and mechanical characteristics that the sensor will encounter in deployment.

You do not need the IMU connected during this step. The goal here is to verify that the movement routines execute correctly and consistently before collecting sensor data in “Collect the Training Data” on page XX.

The Stationary Sample

This is the simplest case; you power up the servos but hold the robot arm in a stationary pose. You could do this with the arm switched off, but

even with the servos not moving, there is still low-level vibration when voltage is applied, which you want to capture. An energized servo holds its position using a feedback loop that produces low-level vibration even when stationary. This vibration is part of the sensor's normal operating signature, and the MLC needs to learn it. If you collected the stationary baseline with the servos powered down, the model would flag normal powered-idle as anomalous. The stationary pose sketch is provided in repo file *c09003.ino*.

The Nominal Sample

Our robot arm is emulating a pick-and-place robot. The robot picks up an object, moves its arm to the right, places the object down, lifts the arm back up, returns to the left, and then repeats the entire sequence. As shown in Table 9-5, each pick-and-place cycle takes around 19 seconds. The delay shown is the time between each action; this allows the arm time to move into position before executing the next action.

Table 9-5: Pick-and-Place Cycle Timing

Function	Description	Added delay(s)
`up()`	Shoulder and elbow	2.0
`forward()`	Shoulder and final pose	3.0
`close_gripper()`	Gripper close	0.0
`up()`	Shoulder and elbow	2.0
`roll_right()`	Wrist roll	0.0
`swing_right()`	Waist sequence	2.0
`roll_straight()`	Wrist roll	0.0
`forward()`	Shoulder and final pose	3.0
`up()`	Shoulder and elbow	2.0
`swing_straight()`	Waist to center	1.0
`down()`	Shoulder and elbow	2.0

Each function in the sequence maps to a combination of servo movements. The `up()` function raises the shoulder and elbow to a clearance height. The `forward()` function lowers the shoulder to position the gripper over the target. The `swing_right()` function rotates the waist to move the arm to the drop-off position.

The sketch is like that used for the stationary pose but loops the pick-and-place action `REPS` times. The sketch uses a `Timer2` interrupt to blink the onboard LED at a rate that matches the current cycle number: one blink per second during the first cycle, two during the second, and so on. After the final cycle, the LED switches to a rapid blink to signal that the routine is complete. This gives you a visual reference during data collection without needing to watch the Serial Monitor.

In `setup()`, all servos are configured with a speed limit of 5 and an acceleration limit of 127. These settings produce smooth, controlled movements rather than the abrupt snaps that would occur at default settings. The motion profile directly affects the IMU data, so these values must remain consistent between training and deployment.

The `swing_right()` function reduces the waist acceleration to 64 for a smoother rotation. The waist carries the most inertia when the arm is extended, so a lower acceleration limit prevents overshoot and mechanical stress. The `start()` function moves the arm to its initial pose before the first cycle and returns it to the same pose after the last. The `goHome()` call at the end sends all servos to the home positions stored in the Maestro's configuration.

You may notice that the gripper closes during each cycle but never explicitly reopens. For anomaly detection purposes, the IMU signature is dominated by the larger arm movements, so the gripper state has minimal effect on the training data. If you want to add an `open_gripper()` call at the place position for realism, it will not affect the model. The nominal pick-and-place sketch can be found in the *c09004.ino* file in the repo.

During data collection (see "Collect the Training Data" on page XX), you will record four 20-second samples from the IMU while this routine runs, capturing different phases of the pick-and-place cycle. This ensures the training data includes the full range of nominal motion patterns.

The Anomaly Sample

To simulate anomalies for our pick-and-place robot, you randomly stop the servos in the middle of a pick-and-place cycle. This emulates a control fault where the arm freezes mid-motion, analogous to a communication failure or a firmware crash. Physical collisions produce different signatures (current spikes, mechanical deflection) that would require additional training classes. In a real anomaly detection situation, you would likely monitor the servo current to detect surges and shut down the robot to prevent damage. Each arm action has a 20 percent chance of simulating an anomaly. When an anomaly event occurs, the robot halts, and the sketch must be reset to continue. An extract from the anomaly sketch appears in Listing 9-1, which shows the error code generation process. The complete sketch is in the *c09005.ino* file in the repo.

```
bool check_for_anomaly(const char* step) {
  if (random(0, 10) < 2) { // 20% chance to simulate an anomaly
    Serial.print("⚠  Anomaly detected at: ");
    Serial.println(step);
    while (true); // Simulate freeze (halt system)
  }
  return false;
}

...
```

```
void up() {
  check_for_anomaly("up");
  servoController.setTarget(SHOULDER, 4000);
  delay(1000);
  servoController.setTarget(ELBOW, 6000);
  delay(1000);
  servoController.setTarget(WRIST_PITCH, 6000);
}
...
```

Listing 9-1: The anomaly-generating code used to interrupt the pick-and-place movement

Collect the Training Data

Your complete hardware setup should look like Figure 9-7. From left to right in this image, you can see the robot arm with the IMU mounted, the ST evaluation kit, the servo controller, the Arduino UNO, and the servo power supply.

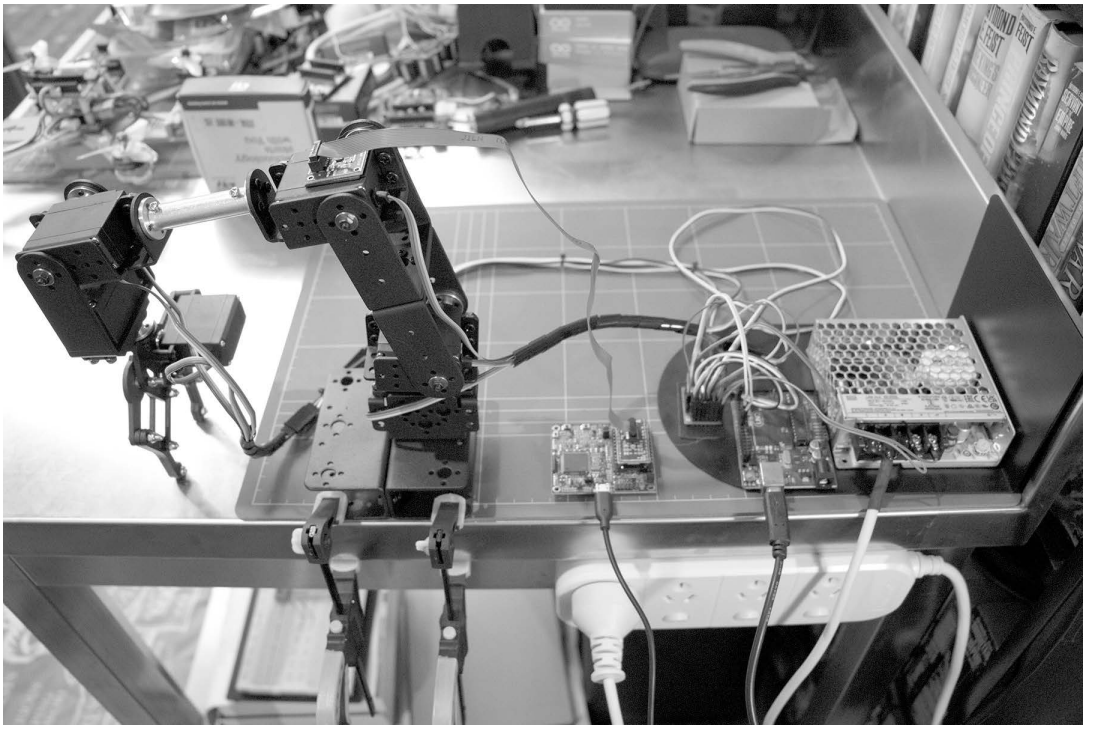


Figure 9-7: The robot arm, controller, and IMU arrangement

Using this setup, you can capture three training datasets: stationary, nominal, and anomaly. In our build, the IMU is mounted on the elbow joint because the sensor cable was not long enough to reach the gripper. This position still captures significant motion from the shoulder and elbow

joints, which dominate the pick-and-place cycle. Depending on the types of anomalies you are trying to detect, somewhere closer to the gripper may be better.

The sensor evaluation kit consists of two boards: the MKI245KA, which carries the ISM330BX IMU on a DIL24 adapter, and the MKI109V3 motherboard (Figure 9-8), which handles USB communication with the PC. Plug the adapter into the motherboard and connect via USB.

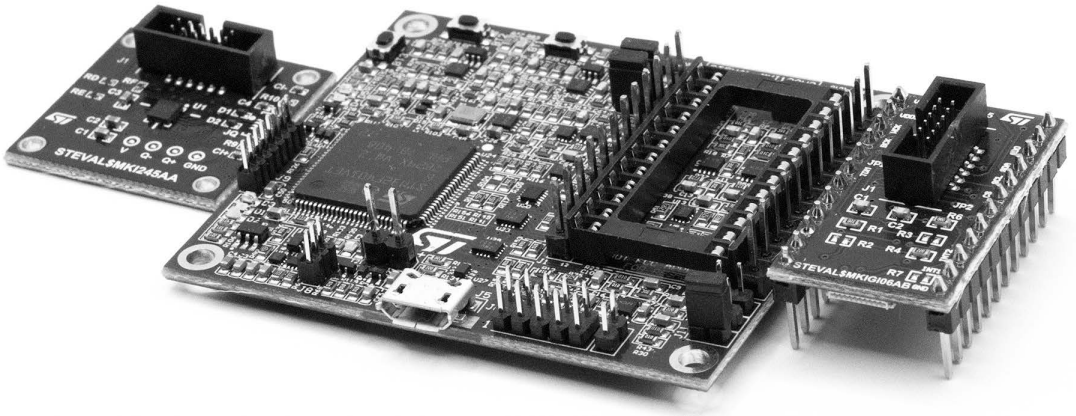


Figure 9-8: The ST MEMS sensor evaluation boards (MKI245KA, MKI109V3, and MKIGI06A)

The firmware handles the communication between the board and the PC through the USB bus. Figure 9-9 shows a block diagram of the motion data acquisition and anomaly detection circuit. This is independent of the robot arm control.

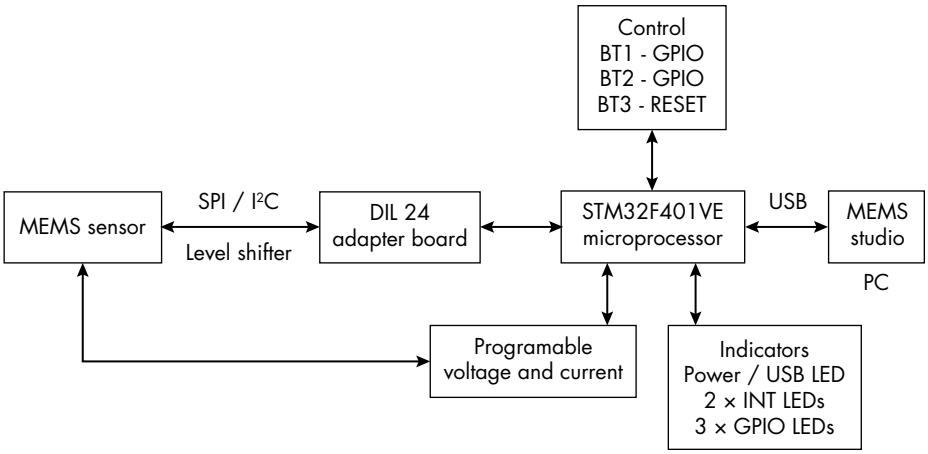


Figure 9-9: A block diagram of data acquisition and anomaly detection

Open **MEMS Studio** (Figure 9-10) and connect to the MKI109V3 motherboard via serial communication. Select the **ISM330BX** from the device

list. If the sensor does not appear, check the USB connection and ensure the adapter board is seated correctly.

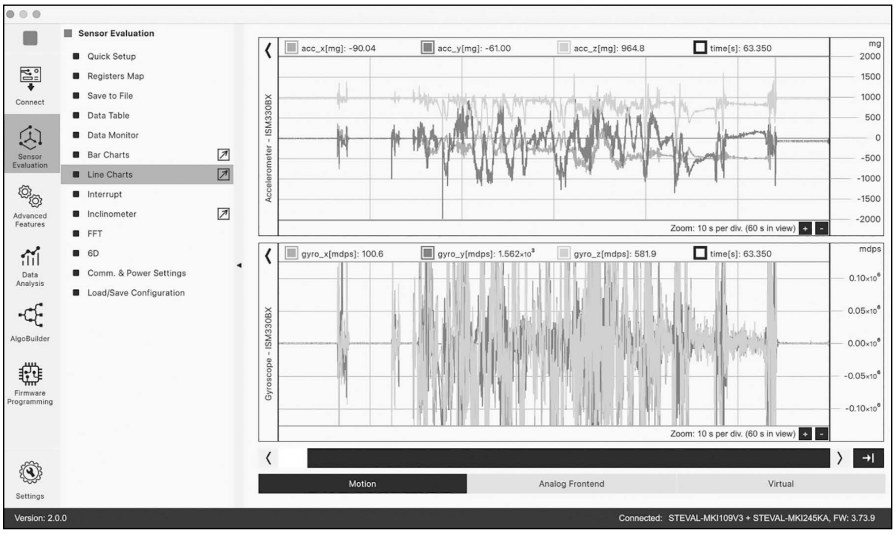


Figure 9-10: MEMS-Studio real-time sensor evaluation

You will use MEMS-Studio to log three datasets: stationary, nominal, and anomaly. Each dataset is saved as a CSV file containing timestamped accelerometer and gyroscope readings. The process is the same for all three: Run the appropriate sketch on the Arduino, start a timed recording in MEMS-Studio, and save the logfile.

Stationary Dataset Acquisition

Download and run the sketch described earlier (*c09003.ino*) for the stationary state on the UNO. Wait for the robot arm to assume the pose, connect the sensor motherboard to your PC, and fire up MEMS-Studio. You then must connect with the motherboard using serial communication and select the ISM330BX sensor. Click **Sensor Evaluation** in the left-hand navigation pane and **Easy Configuration** under **Quick Setup**.

MEMS-Studio requires you to configure logging before you start data acquisition (Figure 9-11).

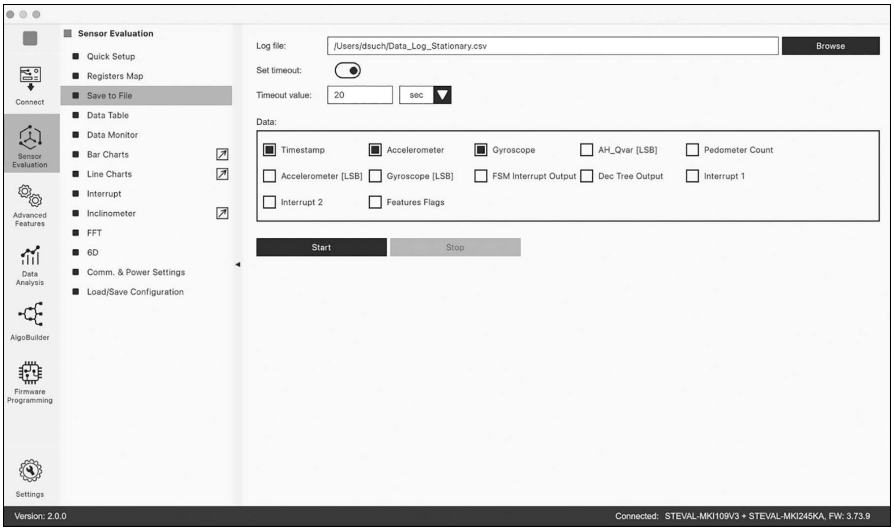


Figure 9-11: The MEMS-Studio data logging screen

The workflow is:

1. Click **Save to File** and choose your output filename.
2. Select the data items (timestamp, accelerometer, or gyroscope).
3. Click the **Play** button in the top left to begin acquisition.
4. Click the **Record** button to start logging. If you click Play before configuring the logfile, you will need to stop and restart.

Not much is happening with the stationary measurements, so try a 20-second sample period. The output will look something like Table 9-6. The gyroscope readings may appear high for a stationary sensor. This is typical of uncalibrated MEMS gyroscopes and reflects the sensor's bias offset rather than actual rotation. The MLC training process accounts for this; the model learns from the pattern of values, not from their absolute magnitude.

Table 9-6: Extract from *Data_Log_Stationary.csv*

Time (µs)	Acc X (mg)	Acc Y (mg)	Acc Z (mg)	Gyro X (mdps)	Gyro Y (mdps)	Gyro Z (mdps)
336683489	–990.396	–30.988	44.530	109.375	1351.875	494.375
336683638	–989.481	–30.195	45.811	43.750	1295.000	511.875
336687765	–989.359	–31.293	45.994	153.125	1316.875	529.375

Nominal Dataset Acquisition

Next, load the pick-and-place sketch (*c09004.ino*) onto the UNO and run it. Connect the IMU to your PC and start MEMS-Studio. If you look at the real-time Line Charts tool in MEMS-Studio (Figure 9-12) as the arm moves, you can see the IMU is initially stationary, then there is a noisy start-up phase, and after about 10 seconds the IMU settles into a consistent pattern once the arm is in the pick-and-place cycle. The screenshot shows almost three pick-and-place cycles.

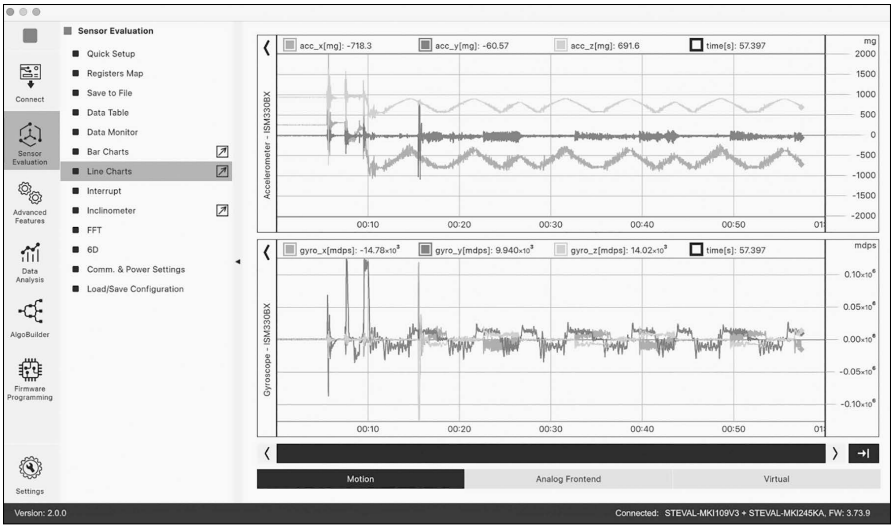


Figure 9-12: Real-time monitoring of the pick-and-place IMU pattern

Run the sketch and let the arm complete at least one full cycle to move past the start-up transient. Then, start a 20-second recording in MEMS-Studio. Restart the sketch and repeat, beginning the recording at a different point in the cycle each time. Collect four recordings in total. This ensures the training data captures all phases of the pick-and-place motion, not just one segment.

Anomaly Dataset Acquisition

Finally, load the anomaly sketch (*c09005.ino*) onto the UNO and run it. As before, use MEMS-Studio to capture the anomaly data logs. When the arm freezes, stop the recording. You will have to reset the UNO after an anomaly occurs to restart the sketch. Use the RESET button on the Arduino. With each move having a 20 percent chance of triggering an anomaly, you should get one anomaly every cycle. Collect at least six anomaly samples, since the halt occurs at different points in the cycle each time due to the

random trigger. Discard any recording where the anomaly occurred during the start-up phase rather than during steady-state motion. In a real-time plot, an anomaly will look like Figure 9-13.

If you prefer, you can also collect a single continuous recording for all states and label the segments in MEMS-Studio.

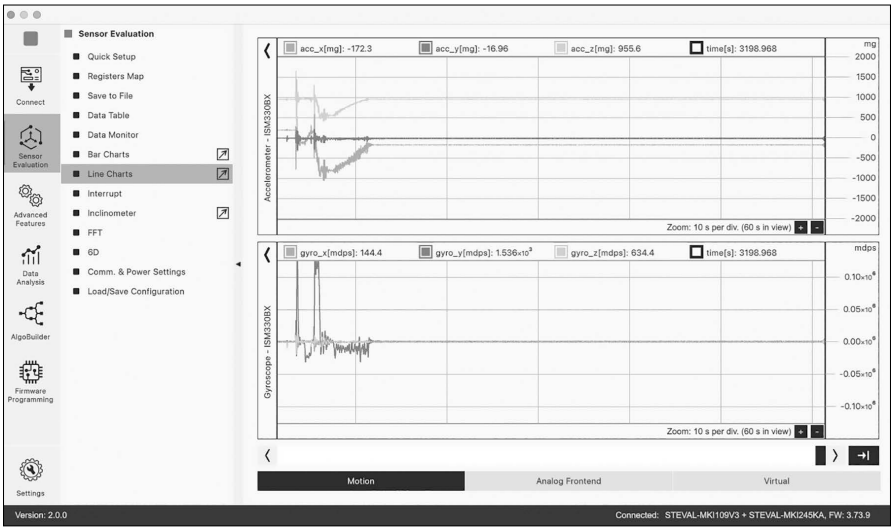


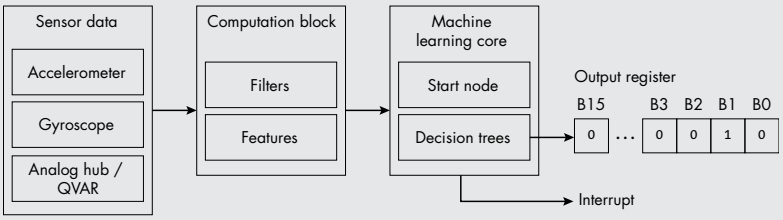
Figure 9-13: IMU anomaly detection in real time

WHAT IS A MACHINE LEARNING CORE?

The ISM330BX IMU includes a dedicated MLC designed to offload specific classification tasks from the main application processor to the sensor itself. The MLC operates by identifying whether a stream of sensor data, typically from the onboard accelerometer and gyroscope, matches one of several user-defined motion classes. In addition to inertial data, the core can also process signals from external sources such as the analog hub or QVAR sensors, making it suitable for a range of pattern recognition tasks beyond basic motion detection.

To prepare the input data for classification, the MLC employs a configurable computation block. This block allows you to apply signal filtering and to extract features over a fixed time window, both of which are critical for robust ML inference. These processed values, along with raw filtered data, are accessible via the sensor's FIFO buffer, making it easy to retrieve the classification result.

(continued)



The decision-making mechanism within the MLC is based on a logic tree composed of a sequence of nodes, each defined by *if-then-else* conditions. At each node, feature values are compared to user-defined thresholds to determine the next step in the decision process. This structure allows for the creation of efficient, low-latency decision trees tailored to specific classification tasks, all running directly on the sensor with minimal computational overhead.

Configure the Machine Learning Core

The ISM330BX can run up to four decision trees simultaneously, each producing 16 classification results. The results are stored in a dedicated output register that can be read at any time. You don't need the sensor physically connected for this step; MEMS-Studio can generate and validate decision trees offline using the CSV data you collected in "Collect the Training Data" on page XX.

To make things easier, you logged training samples for the three states: stationary, nominal, and anomaly. But you could have one sample with all three states and use MEMS-Studio to label the data (Figure 9-14). If you use the single-data-log approach, once you have labeled the classes, click **Save Labeled Data** and a file will be generated for each training class.

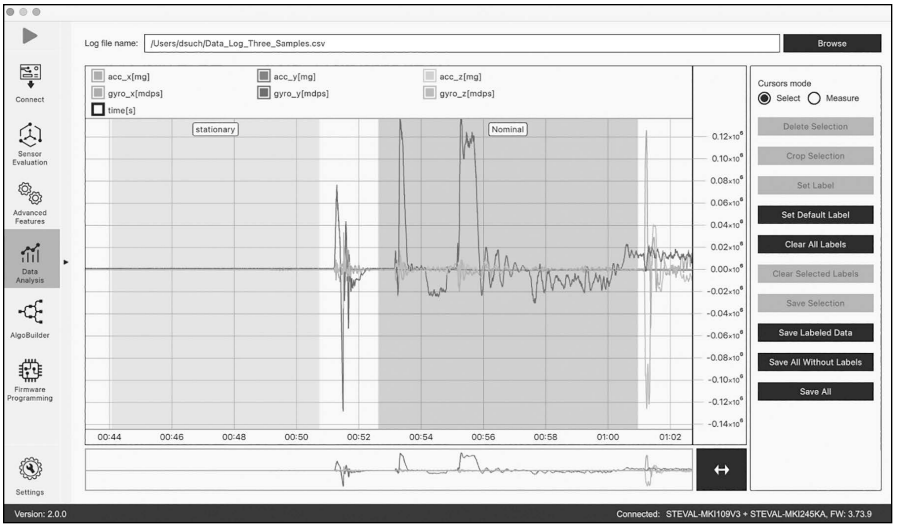


Figure 9-14: Using MEMS-Studio to label data log samples

The Data Analysis tab is a good place to check your training data logs before importing them into the MLC tool (Figure 9-15). Plot each dataset and verify that the stationary samples show low variance, the nominal samples show a repeating cyclic pattern matching the pick-and-place cycle, and the anomaly samples show a clear transition from motion to stillness. If any sample contains recording artifacts (for example, spikes at the start or end from clicking the Record button), trim or discard it.

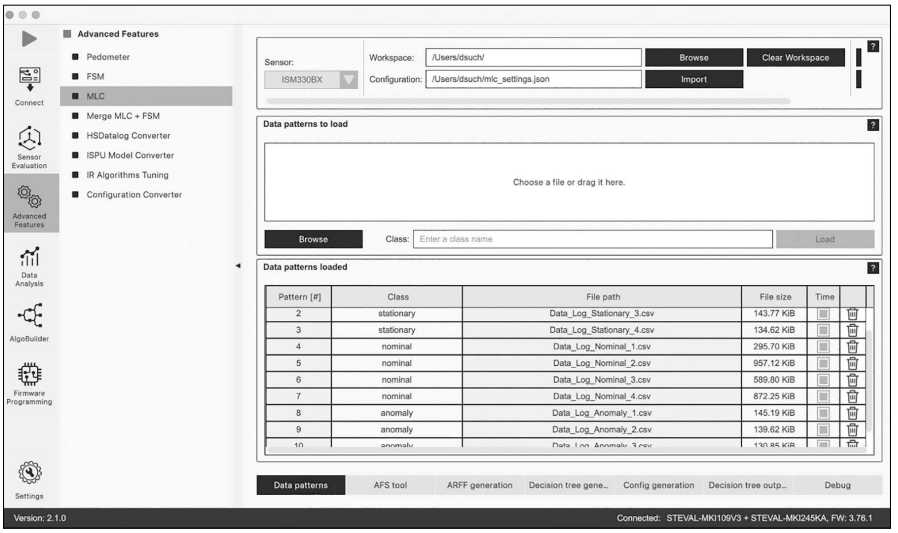


Figure 9-15: The MLC tool in MEMS-Studio

In the MLC tool, click **Import** to import your training data. Load each CSV file and assign it the corresponding class label: stationary, nominal, or anomaly.

After loading the data patterns you generated in "Collect the Training Data" on page XX, check the Class labels, and then click the **AFS Tool** tab (Figure 9-16). This is where you do automatic filter and feature selection (AFS). To start, use the default values, but there are heaps of options if you want to tweak the ML features. In general, selecting a higher number of features increases the risk of overfitting and results in a larger decision tree. This can lead to a model that performs well on training data but generalizes poorly to new, unseen data.

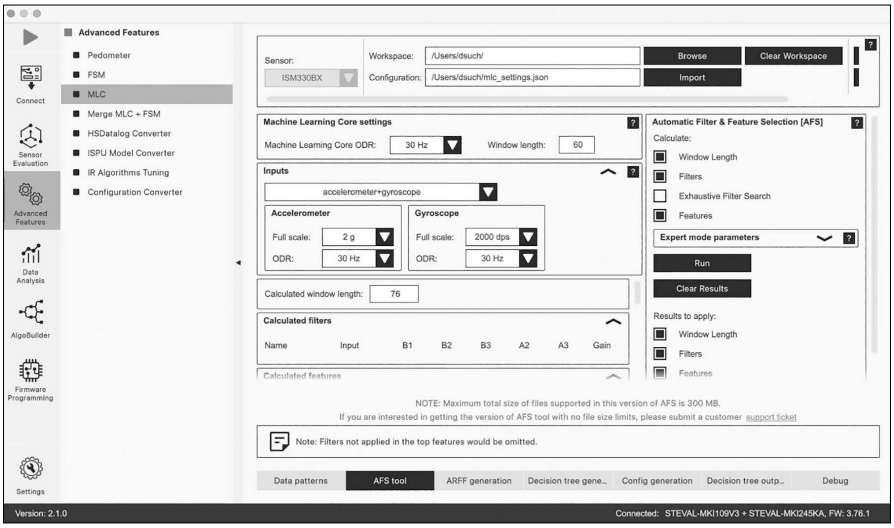


Figure 9-16: Automatic Filter and Feature Selection options

The window length defines how many samples the MLC tool accumulates before computing features and making a classification. MEMS-Studio calculates this automatically based on the lowest-frequency component in the training data. A longer window captures more context but increases classification latency. For the pick-and-place cycle with its multi-second phases, the default calculation should produce a window of several seconds, which is appropriate.

Automatic filter selection can be performed using two different methods. The first is basic search, which identifies the strongest peaks in the frequency spectrum. This approach is computationally efficient but performs poorly on signals that are relatively flat and lack distinct peaks. The second method is exhaustive search, which takes a more comprehensive approach. It evaluates all possible combinations of frequency bands (arranged logarithmically) and selects the frequency that best separates the target class from others by minimizing entropy. While this method is more accurate, it

requires significantly more processing power. Try the basic search as a first pass.

Click **Run** to execute the filter and feature selection, then click **Apply to Settings** to transfer the results to the MLC configuration. The generated window length, filter coefficients, and selected features appear in the ARFF Generation tab (Figure 9-17).

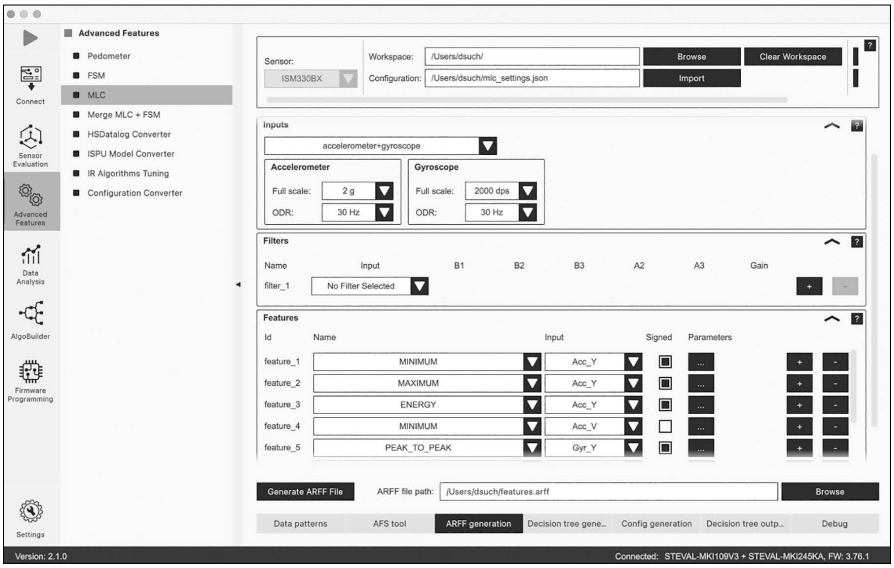


Figure 9-17: The MEMS-Studio ARFF Generation tab

Click **Generate ARFF File** to create a copy of your training data. Open the generated ARFF file in a text editor and verify that it contains rows for all three classes. The class distribution should be roughly balanced. If one class dominates (it appears in more than 60 percent of the rows), consider collecting additional samples for the underrepresented classes before proceeding.

The Attribute Relation File Format (ARFF) is a standard text-based file format commonly used to store datasets for ML applications. An ARFF file has two main parts: a header that defines the structure of the data, and a data section that lists the actual examples used for training or testing a model. An extract from our ARFF file is in Listing 9-2.

```
@relation 'MLC'

@attribute F1_MINIMUM_ACC_Y numeric
@attribute F2_MAXIMUM_ACC_Y numeric
@attribute F3_ENERGY_ACC_Y numeric
@attribute F4_ABS_MINIMUM_ACC_V numeric
@attribute F5_PEAK_TO_PEAK_GYR_Y numeric
@attribute F6_MINIMUM_ACC_Z numeric
```

```
@attribute F7_ABS_MEAN_GYR_V numeric
@attribute F8_ABS_MAXIMUM_ACC_V numeric
@attribute F9_PEAK_TO_PEAK_ACC_Y numeric
@attribute F10_MINIMUM_ACC_X numeric
@attribute F11_ABS_VARIANCE_ACC_V numeric
@attribute F12_ABS_ENERGY_GYR_V numeric
@attribute F13_ABS_MEAN_ACC_V numeric
@attribute F14_ABS_ENERGY_ACC_V numeric
@attribute F15_MAXIMUM_ACC_X numeric
@attribute class {stationary, nominal, anomaly}

@data
0.0285645, 0.0310669, 0.0684204, 0.974609, 0.00404358, 0.968262, 0.0242462, 0.978516,
0.00250244,
0.1073, 0.000976563, 0.0444946, 0.977539, 72.625, 0.110291, stationary
0.0275269, 0.0314331, 0.0685425, 0.975098, 0.0038147, 0.96875, 0.0242004, 0.978516, 0.00390625,
0.106689, 0.000976563, 0.0445251, 0.978516, 72.625, 0.110291, stationary
0.0286865, 0.0314331, 0.0691528, 0.975098, 0.00404358, 0.96875, 0.0241699, 0.978516,
0.00274658,
0.107666, 0.00195313, 0.044342, 0.978516, 72.5625, 0.110413, stationary
--snip--
```

Listing 9-2: An extract from an ARFF MLC file

In the context of the MLC, each row of the ARFF file represents a single time window of sensor data that has been processed into a set of features. These windows are the basic units the MLC uses to recognize patterns or classify behaviors. By storing each window as a separate instance in the ARFF file, the format provides a clear and organized way to train and evaluate ML models on sensor-based data.

The AFS tool selected 15 features for this dataset. This is toward the upper end for MLC deployment. If the decision tree generated in the next section is excessively large or generalizes poorly, return to the AFS tab and reduce the maximum number of features to 8 or 10.

Generate the Decision Tree

The next tab (Figure 9-18) is for decision tree generation, and this is where everything comes together.

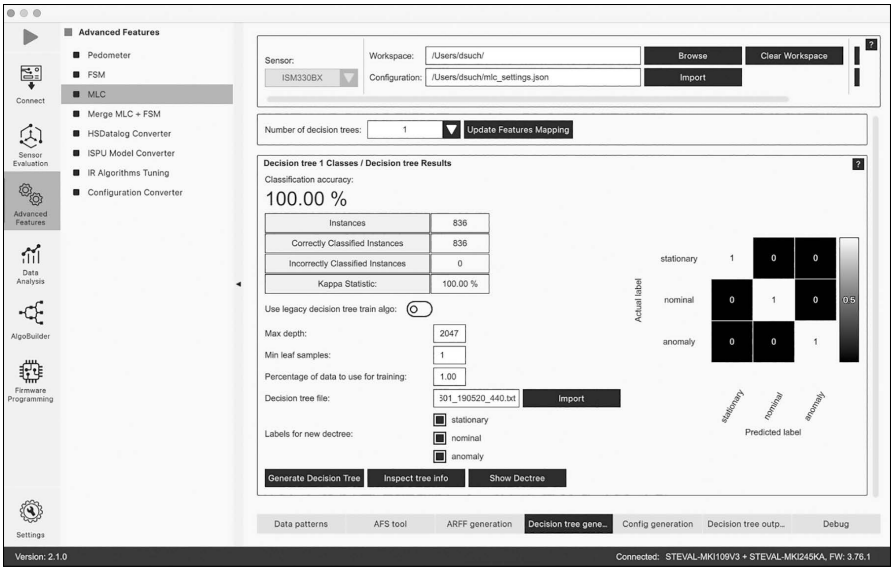


Figure 9-18: The decision tree generation and confusion matrix

The Decision Tree Generation tab provides tools for configuring, training, and evaluating a decision tree classifier based on the selected data classes. You can specify the number of decision trees to train, as well as fine-tune the model using parameters. Leave the number of decision trees at one. For this project, a single tree is sufficient to classify three motion states.

Starting with version 2.0.0 of MEMS-Studio, a new decision tree algorithm is available. Use this algorithm rather than the legacy option. The cross-validation it performs automatically helps prevent overfitting, which is important given the relatively small number of training instances in this project.

This updated algorithm supports both depth-first and best-first tree construction methods. It includes Breiman's cost-complexity pruning and uses cross-validation to determine the optimal tree size. You can adjust parameters such as the maximum tree depth (`Max depth`), the minimum number of samples required in a leaf node (`Min leaf samples`), and the proportion of data used for training (`Percentage of data to use for training`).

The default parameters are permissive: maximum depth of 2047 (effectively unlimited), minimum leaf samples of 1, and 100 percent of the data used for training. These defaults give the algorithm maximum freedom to fit the data. In many ML contexts, this would be a recipe for overfitting, but the v2.0.0 algorithm applies Breiman's cost-complexity pruning with internal cross-validation to keep the tree compact. For this project, the defaults produced a tree with just four leaves, which confirms that the three motion classes are well separated by the selected features. If your tree is significantly larger (more than 10 to 15 leaves), that may indicate noisy training data or poor class separation rather than a need to tighten the

parameters. In that case, return to "Collect the Training Data" on page XX and review your training samples before reducing max depth.

The earlier decision tree training algorithm can still be used by enabling the Use Legacy Decision Tree Train algo option. When this legacy method is selected, parameters such as the `Maximum number of nodes` and `Confidence factor` can be adjusted to control the pruning process.

After model configuration is complete, create the decision tree by clicking the **Generate Decision Tree** button. After your tree has been generated or imported, MEMS-Studio displays performance metrics including accuracy, total number of instances, and the confusion matrix. You can examine the resulting decision tree using Inspect Tree Info (Listing 9-3) for a textual breakdown or Show Decision Tree for a graphical view (Figure 9-19). Your tree will have different values based on the training datasets you provided. What matters is not an exact match but comparable accuracy and a confusion matrix without large off-diagonal values.

```
F6_MINIMUM_ACC_Z <= 0.934082
|   F15_MAXIMUM_ACC_X <= -0.882080
|   |   F3_ENERGY_ACC_Y <= 0.094879 : nominal (5)
|   |   F3_ENERGY_ACC_Y > 0.094879 : anomaly (127)
|   F15_MAXIMUM_ACC_X > -0.882080 : nominal (578)
F6_MINIMUM_ACC_Z > 0.934082 : stationary (126)

Number of Leaves: 4
Size of the tree: 7

Classes:
=> stationary, nominal, anomaly

=============== Complete training data statistics ===============

Confusion Matrix:
           stationary nominal anomaly   <- classified as
stationary 110         0       0
nominal    0           344     27
anomaly    0           0       110

Total Number of Instances: 591
Correctly Classified Instances: 564
Incorrectly Classified Instances: 27

Accuracy:       95.43 %
Cohen's kappa:  91.80 %

Report:    precision recall support
stationary 1.00       1.00   110
nominal    1.00       0.93   371
anomaly    0.80       1.00   110
avg/total  0.96       0.95   591
```

Listing 9-3: The MLC decision tree information

The confusion matrix reveals that the model's only weakness is the boundary between nominal and anomaly classes: 27 nominal instances were misclassified as anomalies. This makes sense intuitively. An anomaly is defined as the arm stopping mid-cycle, and some nominal windows captured near the end of a movement phase (when the arm is briefly stationary between actions) may resemble an anomaly halt. In a production system, this 7 percent false-positive rate on the nominal class could be addressed by collecting more training data that includes these transitional moments, or by adjusting the anomaly detection threshold.

Cohen's kappa of 0.918 indicates strong agreement beyond what would be expected by chance alone. Values above 0.80 are generally considered excellent for classification tasks.

For anomaly detection, recall on the anomaly class is the critical metric: A missed fault could result in equipment damage. The model achieves perfect recall (1.00) on the anomaly class, meaning it never fails to flag a genuine anomaly. The precision of 0.80 means roughly one in five anomaly alerts will be a false alarm during normal operation.

The generated tree uses only three of the 15 available features (`F6_MINIMUM_ACC_Z`, `F15_MAXIMUM_ACC_X`, and `F3_ENERGY_ACC_Y`) and has just four leaf nodes. This compactness is ideal for MLC deployment: Fewer nodes means faster classification and less memory consumption on the sensor. We redrew the decision tree in Figure 9-19 because the automatically generated tree is hard to read.

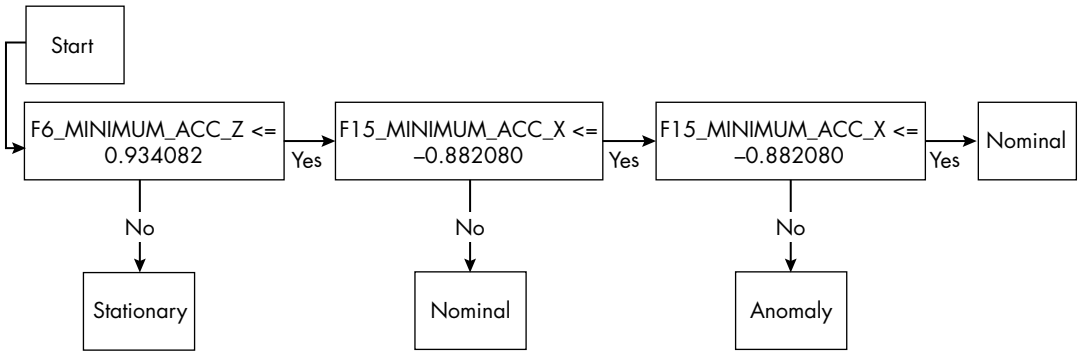


Figure 9-19: The generated decision tree for the machine learning core

If the accuracy is below 90 percent or the confusion matrix shows significant misclassification between all three classes, return to "Configure the Machine Learning Core" on page XX and try exhaustive search in the AFS tool, or reduce the number of features to focus the tree on the most discriminative signals. If the boundary between nominal and anomaly remains weak, collect additional training samples that capture the transitional moments.

Deploy the Model

At this point, the decision tree exists only on your PC. To run it on the ISM330BX, you need to generate a configuration file that encodes the tree structure, feature definitions, and classification parameters in a format the MLC hardware can execute.

The Config Generation tab (Figure 9-20) is used to configure the metaclassifiers and generate a *.json* file that defines the ML core settings. This file contains all the necessary information for deploying the trained decision tree to your sensor.

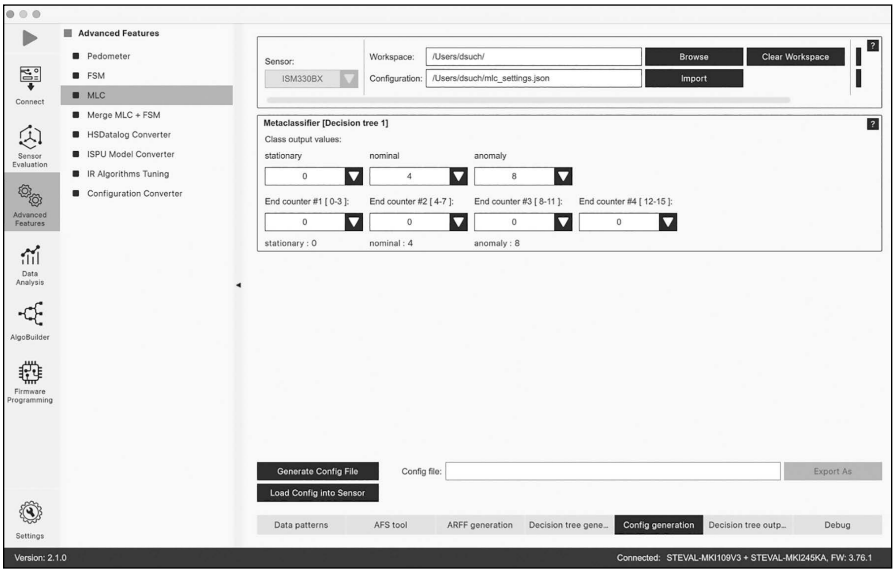


Figure 9-20: Creating a configuration file for deployment

A metaclassifier is a hysteresis filter that stabilizes the output of a decision tree. It works by using internal counters to monitor how often a particular class or group of similar classes is detected. Classes can be grouped into subgroups, and each subgroup has its own counter. This counter increases when the decision tree outputs a class from the subgroup and decreases when it does not. Once the counter reaches a preset value, called the *end counter* (set by the user and ranging from 0 to 14), the ML core updates its output. This approach helps reduce false positives and prevents the system from switching too quickly between classes when the data is noisy or unclear. The ISM330BX supports up to four subgroups, so you might need to group closely related classes when using the metaclassifier.

For this project, the three motion classes are sufficiently distinct that the decision tree output is stable without metaclassifier filtering. You can leave the end counter at 0, which disables the hysteresis and passes the raw classification through to the output register. If you later add classes with similar motion profiles or find that the output flickers between classes during transitions, increasing the end counter to 2 or 3 will stabilize the result at the cost of a slight delay in detecting class changes.

In the Config Generation tab, each class defined during training is assigned a unique output value (stationary = 0, nominal = 4, and anomaly = 8). These class output values are numerical identifiers that the ML core uses to represent classification results. When the sensor detects a pattern corresponding to one of the trained classes, it writes the associated output value to a designated register. These values can then be read by the microcontroller to determine which class the sensor has recognized.

In addition to JSON format, the configuration can also be exported as a *.h* header file using the Export As button. This is useful for embedding the configuration directly into firmware.

If a supported sensor is connected to the development board, the configuration can be uploaded directly to the sensor's memory using the Load Config into Sensor button. This step programs the MLC with the selected settings, enabling it to begin classification tasks without further input from the host processor. Go ahead and load your configuration into the IMU.

Test It Out

Before disconnecting the sensor from the evaluation motherboard, verify the model using the Decision Tree Output Viewer tab in MEMS-Studio. Run each of the three movement sketches on the Arduino and confirm that the viewer displays the correct classification. This is the fastest way to validate the model. Once verified, you can proceed to connect the sensor to the Arduino and read the classification via I^2C.

The Decision Tree Output Viewer tab (Figure 9-21) allows you to test the generated MLC configuration by displaying the output of each decision tree in real time on a chart. This helps you verify that the classifier is functioning correctly and responding as expected to different input patterns.

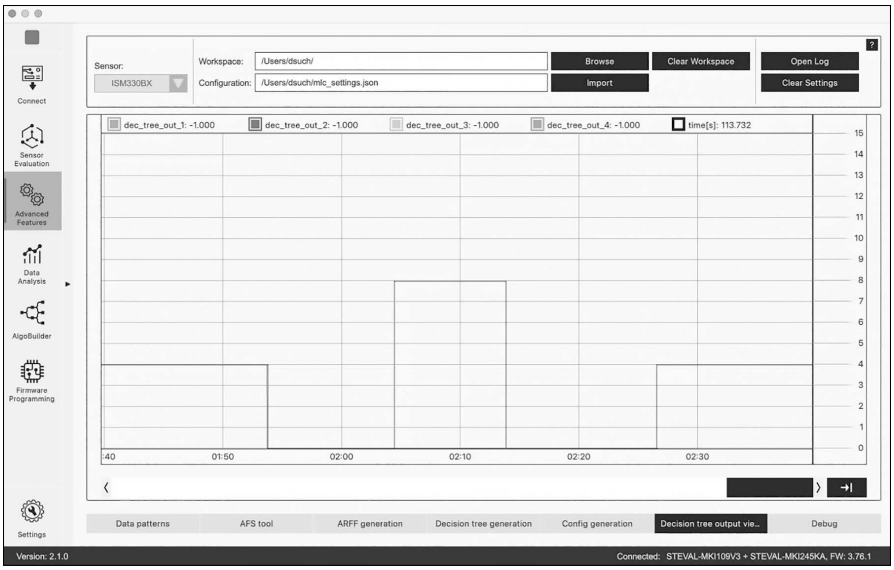


Figure 9-21: The decision tree output viewer

Once you have confirmed that the model classifies correctly in the Decision Tree Output Viewer, the next step is to read the classification result from the Arduino. This moves you from a PC-based verification to the standalone configuration your final system will use.

For hardware testing, disconnect the MKI245KA sensor board from the MKI109V3 motherboard and connect it to the Arduino UNO via I²C. Connect SDA to Arduino pin A4, connect SCL to Arduino pin A5, and provide 3.3 V power from the Arduino's 3V3 pin. The MLC configuration persists in the sensor's memory after programming, so the evaluation motherboard is no longer needed.

Repo file *c09006.ino* is a simple Arduino sketch that you can use to initialize I²C communication with an ISM330BX and read the MLC state from the relevant register. This assumes the sensor is connected using I²C and that you have already loaded an MLC configuration into the IMU. This is a very bare-bones sketch and doesn't do any IMU initialization; as a sanity check, you could read the `WHO_AM_I` register (0x0F) at startup and verify it returns the expected device ID for the ISM330BX.

The MLC output is stored in dedicated registers (Table 9-7), which contain the recognized class ID from the decision tree. The `MLC_STATUS` register (0x15) contains the interrupt status bits for changes in the decision tree result.

Table 9-7: The MLC Output Registers for ISM330BX Decision Trees

Decision tree	Register name	Address
Decision tree 1	MLC1_SRC	0x70
Decision tree 2	MLC2_SRC	0x71
Decision tree 3	MLC3_SRC	0x72
Decision tree 4	MLC4_SRC	0x73

Run the robot's motion sequence and observe when the model identifies a transition from normal movement to anomaly.

What's Next

Once your robot arm can detect motion anomalies using the ISM330BX's internal ML core, you can build on this foundation in several directions.

You can start by improving fault coverage. The model you created distinguishes between normal and abnormal motion, but you can expand it to identify specific fault types such as servo failure, loose joints, or mechanical obstructions. This requires collecting additional labeled datasets that represent each failure mode. MEMS-Studio makes this process straightforward by enabling you to label multiple classes and retrain the decision tree without rewriting any code.

Another useful extension is to monitor servo current. Many robotic faults first appear as a rise in current before motion changes. Adding current sensing with small inline shunt resistors or dedicated current sensors such as the INA219 allows you to correlate electrical behavior with motion data. Combining these signals in a multi-sensor ML model increases reliability and can help predict faults before they occur.

You can also integrate wireless communication. Using Bluetooth Low Energy or Wi-Fi, the IMU's classification output can be transmitted to a dashboard or maintenance system. This turns your robot arm into a connected device capable of self-reporting its health status. For industrial setups, alerts could be sent to a supervisory controller or maintenance log automatically.

If you want to go further with embedded intelligence, consider using the ISM330BX's Intelligent Sensor Processing Unit (ISPU). The ISPU lets you run custom C code or neural network models directly on the sensor, enabling more complex pattern recognition or adaptive learning without increasing system load.

On the mechanical side, you could scale the concept to multi-axis robots or conveyor systems. The same process applies: Mount sensors on key

moving parts, collect labeled data, and train small models to detect irregularities. Over time, this creates a network of smart sensors, each of which understands its normal motion pattern.

Finally, you might integrate this project into a larger predictive maintenance system. By logging classified events and their frequency, you can build a long-term health profile for the robot. This data helps schedule maintenance before breakdowns occur, reducing downtime and costs.

This project demonstrated how sensor ML can bring intelligence directly to the data source. By embedding the model inside the IMU, your system detects abnormal motion in real time without relying on a host processor. This approach reduces latency, power use, and complexity while pushing AI to the true edge of the embedded system.

Conclusion

In this chapter, you explored how sensors can do more than just measure: They can interpret, classify, and act. The techniques—collecting labeled sensor data, extracting features, and training a compact model for edge deployment—apply well beyond motion sensing. The next chapter introduces a related but distinct approach to a different sensory domain: audio.

Unwanted noise degrades every audio signal, whether it is a voice on a phone call, a musical note, or a sensor picking up an environmental event. Real-time noise suppression tackles this problem at the source by cleaning up signals before they reach your ears or your model.

In the next chapter, you will build a system that listens, thinks, and improves the input stream. A MEMS microphone captures live audio, which is then processed through a recurrent neural network trained to separate speech from noise. The cleaned audio stream is sent to a display where a spectrogram shows, in real time, how the model suppresses noise and preserves the signal. This project combines hardware, digital signal processing, and machine learning to bring intelligence to hearing.