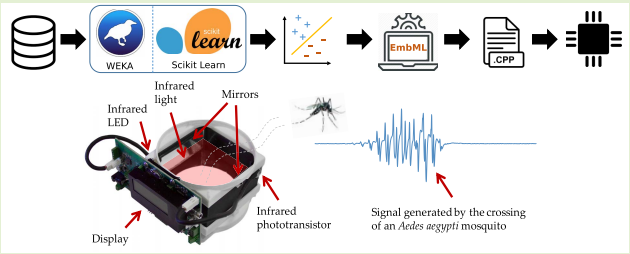
An Open-Source Tool for Classification Models in Resource-Constrained Hardware

Lucas Tsutsui da Silva, Vinicius M. A. Souza, and Gustavo E. A. P. A. Batista

***Abstract*—Sensor applications often face three main restrictions: power consumption, cost, and lack of infrastructure. Most of the challenges imposed by these limitations can be addressed by embedding Machine Learning (ML) classifiers in the sensor hardware, creating smart sensors able to interpret the low-level data stream. However, for this approach to be cost-effective, we need highly efficient classifiers suitable to execute in resource-constrained hardware, such as low-power microcontrollers. In this paper, we present an open-source tool named *EmbML – Embedded Machine Learning* that implements a pipeline to develop classifiers for resource-constrained hardware. We describe EmbML implementation details and comprehensively analyze its classifiers considering accuracy, classification time, and memory usage. Moreover, we compare the performance of EmbML classifiers with classifiers produced by related tools to demonstrate that our tool provides a diverse set of classification algorithms that are both compact and accurate. Finally, we validate EmbML classifiers to recognize disease vector mosquitoes in a smart sensor and trap application.**



***Index Terms*—Classification, edge computing, machine learning, smart sensors.**

I. Introduction

SENSOR applications [1]–[5] often face three main restrictions [6]: power consumption, cost, and lack of infrastructure. For example, sensors often have a battery as their main power source, so efficient power consumption allows them to run for extended periods. Price is a significant factor that hinders scaling in several areas, such as agriculture. Infrastructure assumptions, including the availability of reliable internet connection or power supply, frequently do not hold in remote locations or low-income countries.

Most of these challenges can be addressed by embedding Machine Learning (ML) classifiers in the sensor hardware, creating smart sensors able to interpret the low-level input. These smart sensors are low-powered systems that usually include one or more sensors, a processing unit, memory, a power supply, and a radio [7]. Since sensors have restricted memory capacity and can be deployed in difficult-to-access areas, they often use wireless communication to transfer the data to a base station or the cloud. In this process, smart sensors eliminate the need for communicating all the raw data. Instead, they can only report events of interest periodically when a network connection is available, allowing for efficient use of power and adding tolerance to lack of infrastructure.

However, for this approach to be cost-effective, we need highly efficient classifiers suitable to execute in sensors' resource-constrained hardware, such as low-power microcontrollers. This scenario conflicts with the state-of-practice of ML, in which developers frequently implement classifiers in high-level programming languages such as Java or Python, make unrestricted use of floating-point operations and assume plenty of resources, including memory, processing, and energy.

To overcome these problems, we present an open-source tool named Embedded Machine Learning (EmbML)[1]. Reference [8] that implements a pipeline to develop classifiers for resource-constrained hardware. It starts with learning a classifier in a desktop or server computer using popular software packages or libraries such as Waikato Environment for Knowledge Analysis (WEKA) [9] and scikit-learn [10]. Next, EmbML converts the trained classifier into a carefully crafted code (in C or C++) with support for resource-constrained hardware, such as the avoidance of unnecessary use of static

Manuscript received September 22, 2021; accepted November 9, 2021. Date of publication November 15, 2021; date of current version December 29, 2021. This work was supported in part by the United States Agency for International Development (USAID) under Grant AID-OAA-F-16-00072 and in part by the Brazilian National Council for Scientific and Technological Development (CNPq) under Grant 166919/2017-9. An earlier version of this paper was presented at the 2019 IEEE ICTAI Conference and was published in its proceedings at https://ieeexplore.ieee.org/document/8995408 [DOI: 10.1109/ICTAI.2019.00238]. The associate editor coordinating the review of this article and approving it for publication was Dr. Amitava Chatterjee. *(Corresponding author: Lucas Tsutsui da Silva.)*

Lucas Tsutsui da Silva is with the Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos 13566-590, Brazil (e-mail: lucastsutsui@usp.br).

Vinicius M. A. Souza is with the Graduate Program in Informatics, Pontifícia Universidade Católica do Paraná, Curitiba 80215-901, Brazil (e-mail: vinicius@ppgia.pucpr.br).

Gustavo E. A. P. A. Batista is with the School of Computer Science and Engineering, University of New South Wales, Sydney, NSW 2052, Australia (e-mail: g.batista@unsw.edu.au).

This article has supplementary downloadable material available at https://doi.org/10.1109/JSEN.2021.3128130, provided by the authors.

Digital Object Identifier 10.1109/JSEN.2021.3128130

[1] Available at https://github.com/lucastsutsui/EmbML

random-access memory (SRAM) and implementation of fixed-point operations for real numbers.

EmbML does not support the learning step in the embedded hardware. We advocate that, for most ML algorithms, the search for model parameters is too expensive to be performed on a microcontroller. However, most ML algorithms output highly efficient classifiers, including the ones supported in our tool: Logistic Regression, Decision Tree, Multilayer Perceptron (MLP), and Support Vector Machine (SVM).

Our main contributions and findings are summarized below:

- We provide an open-source tool to convert ML models learned on a desktop/server using popular frameworks to their use on microcontrollers with constrained resources;
- We demonstrate the efficiency of EmbML in a public health case study and carry out a comprehensive experimental evaluation on six real-world benchmark datasets, four classes of ML algorithms, and six resource-constrained microcontrollers with varied characteristics;
- We empirically demonstrate that replacing floating-point with fixed-point improves the classification time on microcontrollers without Floating-Point Unit (FPU) and can reduce the memory consumption of the classifiers;
- For neural network models, we show that the use of approximations for the sigmoid function can reduce its execution time when compared with the original function;
- For tree-based models, we show that employing an if-then-else statement structure reduces the model execution time and does not impact memory consumption, as compared to iterative version.

The remaining of this paper is organized as follows: Section II discusses the related work; Section III presents the EmbML implementation details; Section IV describes the experimental setup; Section V assesses the performance of EmbML; Section VI evaluates the modifications provided by EmbML; Section VII compares the EmbML performance; Section VIII presents a case study; Section IX discusses the limitations of our work and, finally, Section X presents our conclusions and directions for future work.

II. Related Work

EmbML has three main objectives: *i*) having its source code available to the ML community for free usage and improvement; *ii*) generating microcontroller-tailored classifier code with specific modifications to optimize its execution; and *iii*) providing a variety of supported classification models.

Various tools can convert ML models into source code for resource-constrained hardware [11]. However, most of them are from independent or industry developers, which leads to a scarcity of work with rigorous scientific analysis and experimental comparisons, as conducted in this paper. This section summarizes the most popular model conversion tools that can be directly compared to EmbML, allowing us to better establish our contribution to the state-of-the-art. As discussed, none of these works fulfill the three objectives of EmbML.

Sklearn-porter[2] is a tool to convert classification and regression models trained with the scikit-learn library. This tool supports six programming languages (Java, JavaScript, C, Go, PHP, and Ruby) and eleven classifiers such as SVM, Decision Tree, Random Forest (RF), Naive Bayes (NB), k-Nearest Neighbors (kNN), and MLP. Unfortunately, it does not provide any adaptation of classifier code to support resource-constrained hardware.

Weka-porter[3] is a similar but more restricted project focused on the WEKA package. It converts J48 decision tree classifiers into C, Java, and JavaScript codes. Although we can use the C output code to implement embedded classifiers, the support for a single classifier restricts the applicability of the tool.

Several tools can transform decision tree models into C++ source code. For instance, J48toCPP[4] supports *J48* models from WEKA; C4.5 decision tree generator[5] converts *C4.5* models from WEKA; and DecisionTreeToCpp[6] converts *DecisionTreeClassifier* models from scikit-learn. SVM also has various conversion tools to C language. Two examples are: mSVM[7] that supports fixed-point arithmetic; and uLIBSVM[8] without support to fixed-point operations.

M2cgen[9] converts ML models trained with scikit-learn into native code in Python, Java, C, JavaScript, PHP, R, Go, and others. It supports various classification and regression models, including Logistic Regression, SVM, Decision Tree, RF, and XGBoost. Similarly to sklearn-porter, it does not provide any source code adaptation for microcontrollers.

Emlearn[10] generates code in C from decision tree, NB, MLP, and RF models built with scikit-learn or Keras. It includes features to support embedded devices, such as avoiding the usage of dynamic memory allocation and standard C library, as well as fixed-point representation for NB classifiers. It has little diversity of models, not supporting popular algorithms on embedded systems such as SVM. Also, the NB classifier is the only one currently able to use fixed-point arithmetic.

TensorFlow Lite for Microcontrollers[11] is a library designed to execute TensorFlow Neural Network (NN) models on 32-bit hardware such as microcontrollers. To decrease the model size and memory usage, it allows applying post-training quantization, reducing the precision of numbers in the model. Some limitations identified by the authors include: support for a limited subset of TensorFlow operations, support for a limited set of devices, low-level C++ API requiring manual memory management, and lack of support for training models.

EdgeML[12] enables generating code from ML algorithms for resource-scarce devices. The library allows training, evaluation, and deployment of ML models on various devices and platforms. It implements modified and original ML algorithms that focus on time and memory efficiency to execute in

[2]https://github.com/nok/sklearn-porter

[3]https://github.com/nok/weka-porter
[4]https://github.com/mru00/J48toCPP
[5]https://github.com/hatc/C4.5-decision-tree-cpp
[6]https://github.com/papkov/DecisionTreeToCpp
[7]https://github.com/chenguangshen/mSVM
[8]https://github.com/PJayChen/uLIBSVM
[9]https://github.com/BayesWitnesses/m2cgen
[10]https://github.com/emlearn/emlearn
[11]https://www.tensorflow.org/lite/microcontrollers
[12]https://github.com/Microsoft/EdgeML/

resource-constrained devices [12]–[14]. EdgeML also supports generating code that operates with fixed-point format [15]. Besides being a relatively complete solution, a drawback is that it supports ML models generated only by its original algorithms, requiring particular expertise to manipulate them.

CMSIS-NN [16] contains a set of efficient function implementations for layers usually present in NNs to help dump a trained NN in ARM Cortex based microcontrollers. Note that CMSIS-NN does not support training ML models, and the user is responsible for correctly combining function calls and uploading the network weights into the code. Also, this library implements only fixed-point operations and allows building an NN with any of the following layers: fully connected, convolution, pooling, softmax, and others. Disadvantages of using this library include supporting NN models only, and the manual process of producing a classifier code.

FANN-on-MCU [17] is a framework for the deployment of NNs trained with FANN library on ARM Cortex-M cores and parallel ultra-low power RISC-V-based processors. It offers automated code generation targeted to microcontrollers with fixed or floating-point formats and uses some implementations provided by CMSIS to improve performance on ARM Cortex-M cores. Though this is a robust solution, the number of supported ML models and microcontrollers is very restricted.

All the mentioned tools so far are open-source. An example of a proprietary tool is STM32Cube.AI that allows converting NN models from popular tools into code to run on STM32 ARM Cortex-M-based microcontrollers. Besides being a proprietary tool, another drawback is its lack of models diversity.

Commercial tools are abundant, but beyond the scope of this work due to the costs. Considering the popularity in academia, it is valid to mention the MATLAB Coder, a tool that converts a MATLAB program – including ML models – to produce C and C++ codes for a variety of hardware platforms.

Table I summarizes the related work. Although the description of each tool already includes some of its disadvantages compared to EmbML, the differences are more explicit when analyzing this table. For instance, EmbML is a solution that concomitantly offers: support to classifiers from popular ML tools; diversity of efficient models; and adaptations in the output code to improve its performance in resource-constrained hardware. For a detailed survey on related tools, we redirect the reader to the work of Sanchez-Iborra and Skarmeta [11].

TABLE I
COMPARISON BETWEEN RELATED TOOLS

Tool	ML tool	Classifiers	Adaptations	Output code
EmbML	WEKA and scikit-learn	Decision tree, SVM, MLP and logistic regression	fixed-point and sigmoid approximations	C or C++
sklearn-porter	scikit-learn	SVM, kNN, RF, decision tree, MLP and others	–	Java, JavaScript, C, Go, PHP or Ruby
weka-porter	WEKA	Decision tree	–	C, Java or JavaScript
m2cgen	scikit-learn	Logistic regression, RF, decision tree, SVM and others	–	Python, Java, C, JavaScript, Go, R or others
emlearn	scikit-learn and Keras	Decision tree, NB, RF, MLP and others	fixed-point for naive bayes	C
TensorFlow Lite for Microcontrollers	TensorFlow	NN	quantization	C++
EdgeML	–	Decision tree, kNN, and recurrent NN	quantization	C or C++
CMSIS-NN	–	NN	fixed-point	C or C++
FANN-on-MCU	FANN	NN	fixed-point	C

III. PROPOSED TOOL

The main goal of EmbML[13] is to produce classifier code suitable for execution on low-power microcontrollers. The process starts with creating a model using the WEKA package or the scikit-learn library from a dataset at hand. These are popular and open-source tools that provide a wide range of classification algorithms and simplify the training and evaluation of an ML model. After training the model in a desktop or server computer, the user needs to save it as a serialized object file, which is a file type capable of storing the entire contents of an object, such as the classifier parameters and internal data structures.

EmbML receives such a serialized file as input and uses specific methods to deserialize the file, allowing it to retrieve the classifier data and extract relevant parameters. Finally, EmbML fills a template file (for a specific classifier) using the data retrieved in the previous step and generates a file in C or C++ (default) containing the model parameters, their initialization values, and the classification functions. This output file only includes the functions related to the classification step, since the user may later incorporate other functionalities according to the application, such as feature extraction and preprocessing, and further actions on the classifier output.

Fig. 1 illustrates the operation workflow of EmbML. First, the user chooses one of the supported ML tools to process a training dataset and produce a classification model. EmbML is responsible for Step 2 in which it consumes the file containing the serialized model and creates the classifier source code. In this step, the user should decide to apply any of the provided modifications in the generated source code, such as using the fixed-point format to process real number operations. As we will further discuss, such a choice may impact both the accuracy and efficiency of the classifier. After evaluating the classifier in the desired hardware (Step 3), the user may return to Step 2 if they want to assess other modifications or to Step 1 if the classifier does not meet the application's requirements. In Step 3, it is possible to compile the code and deploy it on the microcontroller, for instance, using a combination of cross compilers and firmware upload protocols.

A. Serialization and Model Recovery

In the described pipeline, the user shall use serialization – *i.e.*, to convert an object state into a format so that the object can be stored in a file – to provide trained classifiers as input to EmbML. Therefore, serialization and deserialization allow to save a model trained with WEKA or scikit-learn in a file and later recover it from this same file, respectively.

The WEKA classifier object must be serialized to a file using the *ObjectOutputStream* and *FileOutputStream* classes available in Java. To recover the model content, EmbML uses

[13]https://github.com/lucastsutsui/EmbML

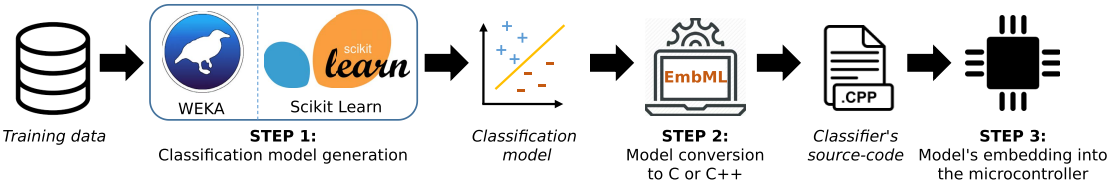


Fig. 1. Workflow for generating classifier source codes using EmbML [8].

the *javaobj*[14] library – available for the Python language. This library allows retrieving a Java serialized object from a file and producing a Python data structure, similar to the original object, that contains all its variables and data with unrestricted access, since Python has no private attributes.

For scikit-learn models, it is possible to use the *pickle*[15] module that allows serializing and deserializing a Python object. The user shall serialize the classification model into a file by applying the *dump* function. After that, EmbML can recover the classifier object from this file using the *load* method and access the object content without restriction.

B. Algorithms and Classes

The algorithms supported by EmbML are those suitable to execute in resource-constrained hardware since they train simple models that generally require little processing time and produce a small memory footprint compared to larger and slower methods such as ensemble and deep learning models. EmbML supports representative models of different learning paradigms: MLP networks, logistic regression, decision tree, and SVM classifiers.

EmbML accepts models from the following WEKA classes: *J48* generates decision tree classifiers; *Logistic* trains logistic regression classifiers; *MultilayerPerceptron* produces MLP classifiers; and *SMO* creates SVM classifiers – with linear, polynomial, and Radial Basis Function (RBF) kernels.

It also supports the models from the following scikit-learn classes: *DecisionTreeClassifier* produces decision tree models; *LinearSVC* builds SVM classifiers with linear kernel; *LogisticRegression* creates logistic regression classifiers; *MLPClassifier* generates MLP classifiers; and *SVC* trains SVM classifiers – with polynomial and RBF kernels.

C. General Modifications

To improve the classifiers' performance in low-power microcontrollers, EmbML implements modifications in all produced source codes. It is worth noting that EmbML never interferes with the training process, it only provides adjustments that affect the execution of the classification process.

One modification is based on the idea that the model parameters – *e.g.*, the weights of an NN – do not change during execution, according to our pipeline. Consequently, these data can be stored in the microcontroller's flash memory, once it is usually larger than its SRAM memory. Therefore, EmbML generates classifier source codes that employ the *const* keyword, which expresses to the compiler that these data are read-only and should be stored in the flash memory.

Another modification lies in the fact that most microcontrollers lack an FPU. This hardware is specifically designed to perform floating-point computations efficiently. There are two options to tackle the absence of FPU [15]: emulating floating-point operations via software or converting real numbers to a fixed-point format. Software emulation usually is processing-expensive. However, the second approach may reduce the range of representable values and cause a loss of precision.

EmbML produces classifier source codes that use both floating-point and fixed-point formats to store real numbers. For the first option, the generated code can directly proceed to the microcontroller's compilers since most of them already provide configuration options to emulate floating-point operations or use FPU instructions. For fixed-point representation, the scenario is quite different since the compilers of different microcontrollers do not offer a universal solution.

Therefore, EmbML implements a library of fixed-point operations based on: *fixedptc*,[16] *libfixmath*[17] and *AVRfix*.[18] It includes the basic arithmetic operations as well as other functions required by some classifiers – *e.g.*, exponential, power, and square root. Our library supports storing real numbers in integer variables with 32, 16, or 8 bits and implements the *Qn.m* format in which *n* and *m* are the numbers of bits in the integer and fractional parts, respectively [18].

D. Modifications for MLP Models

EmbML supports three different approximations for the sigmoid function in MLP classifiers since it requires the expensive processing of the exponential function. The solution lies in replacing it, in the inference step, for functions that have similar behavior but perform simpler operations. However, the model training still employs the original sigmoid function because scikit-learn and WEKA do not support defining custom activation functions.

Fig. 2 illustrates the three options alongside with the original sigmoid curve and reveals that they are in fact comparatively similar. Intuitively, the 4-point Piecewise Linear (PWL) version seems to be the most well-fitted version. The 2-point PWL produces a simple but relatively precise replica of the original pattern. The third approximation generates a smooth curve with strong correspondence with the sigmoid.

Furthermore, to use processing and memory resources more efficiently, EmbML codes implement MLP models that reuse the output buffer of one layer as input to the next layer.

[14] https://pypi.org/project/javaobj-py3/
[15] https://docs.python.org/3/library/pickle.html
[16] https://sourceforge.net/projects/fixedptc/
[17] https://code.google.com/archive/p/libfixmath/
[18] https://sourceforge.net/projects/avrfix/

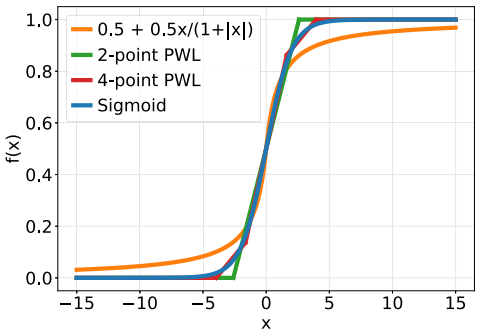


Fig. 2. Approximations available in EmbML for the sigmoid function.

TABLE II
CODE MODIFICATIONS SUPPORTED BY EMBML

Classifier class	*Const* variables	Fixed-point representation	Sigmoid approximations	If-then-else statements
J48	✔	✔		✔
Logistic	✔	✔		
MultilayerPerceptron	✔	✔	✔	
SMO	✔	✔		
DecisionTreeClassifier	✔	✔		✔
LinearSVC	✔	✔		
LogisticRegression	✔	✔		
MLPClassifier	✔	✔	✔	
SVC	✔	✔		

E. If-Then-Else Statements for Decision Trees

For the decision tree models, the default option of EmbML is to produce an output code that represents the binary tree with data structures and traverse the tree from the root to a leaf node iteratively. This iterative approach consists of a loop containing one if-then-else statement that uses the data from the data structures to check the conditional statement of the current node and decide which child the traversal will move to for the next iteration. Alternatively, EmbML can also generate a code produced by unrolling the loop of the iterative method into several nested if-then-else statements – one for each non-leaf node – which embed the data from the entire binary tree structure instead of storing them in data structures. This approach intends to optimize the classifier time performance by eliminating the loop overhead of the iterative method – *e.g.*, instructions to modify the iteration variable and test the end of the loop – at the expense of creating a larger code.

Finally, all the modifications employed by EmbML to optimize the classifiers' processing times and memory costs are summarized in Table II.

IV. EXPERIMENTAL SETUP

Our experiments consider the C++ versions of EmbML classifier codes and three primary metrics to assess their performance: i) accuracy rate, ii) classification time, and iii) memory usage. Ideally, we aim to obtain a high accuracy rate and low values for classification time and memory usage, since it may allow opting for simpler hardware – reducing cost and power requirements for the system.

Accuracy and classification time are estimated using examples from a test set. Memory usage is measured from the compiled classifier code using the GNU *size* utility. As for the classification time, we collected the mean value per instance using the *micros* function from the Arduino library. To better estimate these values, we executed each classifier ten times in the test set. The microcontrollers always read the test set examples from a microSD memory card, but the results consider only the time spent in the classification process.

TABLE III
CHARACTERISTICS OF THE EVALUATED DATASETS

Identifier	Dataset	Features	Classes	Instances
D1	Aedes aegypti-sex	42	2	42,000
D2	Asfault-roads	64	4	4,688
D3	Asfault-streets	64	5	3,878
D4	GasSensorArray	128	6	13,910
D5	PenDigits	8	10	10,992
D6	HAR	561	6	10,299

The analysis of EmbML classifiers includes the performance comparison of three representations for signed real numbers:

1) Floating-point with 32 bits (referred as FLT), defined by the IEEE 754 standard and provided by the compiler;
2) Fixed-point with 32 bits (referred as FXP32), using the $Q22.10$ format provided by EmbML;
3) Fixed-point with 16 bits (referred as FXP16), using the $Q12.4$ format provided by EmbML.

A. Datasets

We consider six benchmark datasets from real-world sensor applications, representing typical use-case scenarios for the EmbML classifiers. Follows a brief description of each dataset:

- *Aedes aegypti*-sex [19]. An optical sensor measures the wingbeat frequency and other audio-related features of flying insects that cross a sensor light. The classification task is to identify the sex of *Aedes aegypti* mosquitoes;
- Asfault-streets [2]. This dataset contains data collected from an accelerometer sensor installed in a car to evaluate pavement conditions of urban streets. The instances have the following categories related to the pavement quality: good, average, fair, and poor, as well as the occurrence of obstacles such as potholes or speed bumps;
- Asfault-roads [2]. This dataset represents the same previously presented problem of pavement conditions evaluation but performed on highways instead of urban streets. The main differences are the lack of the poor class and the car speed during data collection;
- GasSensorArray [20]. It includes the data from a gas delivery platform with 16 chemical sensors that measure six distinct pure gaseous substances in the air: ammonia, acetaldehyde, acetone, ethylene, ethanol, and toluene.
- PenDigits [21]. It comprises the problem of classifying handwritten digits (from 0 to 9) according to the coordinates (x, y) of a pen writing them on a digital screen;
- HAR [3]. This dataset contains data from accelerometer and gyroscope sensors of a waist-mounted smartphone for the following human activities: walking, climbing stairs, downstairs, sitting, standing, and lying down.

Table III shows the datasets main characteristics. The experimental evaluation references these data using the identifiers.

To evaluate the classifiers, we applied a 70/30 holdout validation. This method splits the data into two stratified and mutually exclusive subsets: the training part takes 70% of the instances, and the test set incorporates the 30% remaining.

B. Classifiers

We consider the default hyperparameter values provided by WEKA and scikit-learn. Therefore, the accuracy rates reported in our analysis may not represent the best possible values. Also, the results obtained by the same model learned from different libraries cannot be directly compared since the hyperparameter values are not the same. However, our primary concerns are to study the viability of embedded implementation of classifiers, determine if their execution in microcontrollers achieves the same accuracy as the desktop version, and optimize classification time and memory costs.

Given that we wanted to explore all possibilities supported by EmbML, there were a few cases in which we had to set some hyperparameters manually:

- Since the *SMO* class employs a linear kernel by default, we modified it to train the classifiers with polynomial (using $degree = 2$) and RBF kernels;
- In the case of *SVC* class, we adjusted it to produce SVM models with a polynomial kernel (using $degree = 2$), since the RBF kernel is its default choice;
- We changed the *MLPClassifier* also to create MLP networks that apply the sigmoid activation function, considering that the ReLU function is its default option.

C. Microcontrollers

Given the availability of a large number of microcontrollers suitable for low-power hardware, we selected six microcontrollers used in popular platforms for prototype projects. These platforms are representative examples that can be easily sourced, such as Arduino and Teensy,[19] which also improves the reproducibility of the experiments. However, EmbML is not restricted to maker platforms by any means. Any microcontroller for which a C or C++ compiler is available can use the classifiers generated with the aid of EmbML.

The following microcontrollers were evaluated:

i) ATmega328/P [22] available in the Arduino Uno. It is a low-power 8-bit microcontroller with the simplest features among the chosen models;
ii) ATmega2560 [23] available in the Arduino Mega 2560. It is an 8-bit microcontroller similar to the previous one, with some improvements in memory storage;
iii) AT91SAM3 × 8E [24] available in the Arduino Due. It is a high-performance 32-bit microcontroller and represents one of the most robust Arduino platforms;
iv) MK20DX256VLH7 [25] available in the Teensy 3.1 and Teensy 3.2. It is a 32-bit microcontroller with intermediate processing and memory power;
v) MK64FX512VMD12 [26] available in the Teensy 3.5. It has a single-precision FPU, and better clock frequency and memory storage compared to the previous version;
vi) MK66FX1M0VMD18 [27] available in the Teensy 3.6. It is the most powerful processor in these experiments, operates with 32 bits and includes a single-precision FPU.

[19]Teensy platforms are Arduino-compatible USB-based development boards with various microcontroller configurations.

TABLE IV
CHARACTERISTICS OF THE EVALUATED EMBEDDED PLATFORMS

Platform	Microcontroller	Clock (MHz)	SRAM (kB)	Flash (kB)	FPU
Arduino Uno	ATmega328/P	20	2	32	✘
Arduino Mega 2560	ATmega2560	16	8	256	✘
Arduino Due	AT91SAM3X8E	84	96	512	✘
Teensy 3.2	MK20DX256VLH7	72	64	256	✘
Teensy 3.5	MK64FX512VMD12	120	256	512	✔
Teensy 3.6	MK66FX1M0VMD18	180	256	1,024	✔

The Teensy platforms have an ARM Cortex-M4 core and the Arduino Due platform has an ARM Cortex-M3 core. The Arduino Uno and Arduino Mega 2560 have a low-power microcontroller from the AVR family. Table IV shows some of the main specifications of the chosen embedded platforms.

In the following, we present our experimental evaluation considering three main aspects: accuracy, time, and memory; in Section VI, we assess the code modifications provided by EmbML; and Section VII presents a comparison of classifiers produced by EmbML and related tools. These sections contain an overview of the outcomes of our experiments. We provide detailed results in the supplementary material[20] of this paper.

V. EVALUATION OF EMBML CLASSIFIERS

Next, we describe the experiments involving a sanity check and an analysis of classification time and memory consumption. First, we compare the accuracy rates obtained by EmbML classifiers running on the microcontrollers with the values obtained by scikit-learn or WEKA on a desktop computer, using the same test sets and corresponding trained models. Then, we estimate the time and memory results of the classifiers supported by EmbML as well as the impact of different representations for real numbers.

A. Accuracy

Table V shows the accuracy rates for the test examples of each dataset running the models in a desktop and in a microcontroller with a classifier code produced by EmbML. It does not mention the microcontroller model since all results are the same, independent of the hardware. In this table, the symbol "-" means that the produced code did not fit in any microcontroller's memory, and the accuracies of the EmbML versions appear as differences from their desktop versions.

As expected, the classifiers using FLT obtain the same accuracy rates than their desktop counterparts. These results imply that the EmbML classifiers correctly implement the trained models. There are only some minor exceptions:

- For D2 with the *MultilayerPerceptron*, the accuracy improves from 89.19% (desktop) to 89.26% (EmbML/FLT);
- For D1 with the *DecisionTreeClassifier*, the accuracy reduces from 98.54% (desktop) to 98.53% (EmbML/FLT);
- For D1, D4, and D5 with *SVC* (polynomial kernel), the accuracies from EmbML classifiers (using FLT) are lower than those obtained in desktop. In this specific case, the decrease happens because the *SVC* employs double-precision (64 bits) floating-point operations, and EmbML

[20]https://github.com/lucastsutsui/ieee-sensors-2021

TABLE V
ACCURACY (%) FOR THE EMBML CLASSIFIERS

Classifier	Version	D1	D2	D3	D4	D5	D6
J48	Desktop	99.00	88.48	84.28	97.41	84.71	94.34
	EmbML/FLT	0.00	0.00	0.00	0.00	0.00	0.00
	EmbML/FXP32	-0.03	-0.07	+0.26	0.00	0.00	-0.33
	EmbML/FXP16	-1.75	-1.42	-15.72	-38.76	0.00	-14.73
Logistic	Desktop	97.71	91.61	89.00	98.97	73.00	97.35
	EmbML/FLT	0.00	0.00	0.00	0.00	0.00	0.00
	EmbML/FXP32	-0.06	-0.07	-1.03	-0.62	-0.28	0.00
	EmbML/FXP16	-47.65	-24.04	-71.04	-64.11	-32.19	-2.95
Multilayer-Perceptron	Desktop	98.67	89.19	90.29	92.84	80.46	93.62
	EmbML/FLT	0.00	+0.07	0.00	0.00	0.00	0.00
	EmbML/FXP32	-0.02	+1.14	+0.17	+0.02	+0.12	+0.04
	EmbML/FXP16	-44.27	-0.57	-1.80	-74.46	-0.58	-0.90
SMO (linear kernel)	Desktop	98.39	91.96	91.75	97.13	80.67	98.38
	EmbML/FLT	0.00	0.00	0.00	0.00	0.00	0.00
	EmbML/FXP32	+0.01	+0.36	+0.17	0.00	-0.06	+0.10
	EmbML/FXP16	-8.42	-10.52	-20.36	-74.78	-2.33	**-81.65**
SMO (polynomial kernel)	Desktop	98.76	92.39	91.15	99.40	89.11	98.96
	EmbML/FLT	0.00	0.00	0.00	0.00	0.00	-
	EmbML/FXP32	-0.05	-1.35	-1.63	-62.46	0.00	-
	EmbML/FXP16	-39.73	-65.08	-51.37	**-85.36**	-44.63	-
SMO (RBF kernel)	Desktop	98.08	87.62	83.59	75.59	67.63	95.99
	EmbML/FLT	0.00	0.00	-	-	-	-
	EmbML/FXP32	-0.09	+0.15	-	-	-	-
	EmbML/FXP16	-48.08	-66.92	-48.37	-	-58.04	-
DecisionTree-Classifier	Desktop	98.54	86.13	84.02	97.03	83.83	93.20
	EmbML/FLT	-0.01	0.00	0.00	0.00	0.00	0.00
	EmbML/FXP32	-0.05	-0.35	+0.26	0.00	0.00	-0.35
	EmbML/FXP16	-28.08	-4.76	-20.96	-36.03	0.00	-18.02
LinearSVC	Desktop	90.51	92.11	88.83	80.02	36.74	98.58
	EmbML/FLT	0.00	0.00	0.00	0.00	0.00	0.00
	EmbML/FXP32	-3.87	+0.07	+0.09	-44.75	-0.33	0.00
	EmbML/FXP16	-40.51	**-0.29**	-5.75	-61.57	-27.15	-50.23
LogisticRegression	Desktop	98.18	90.97	84.19	98.06	71.51	98.25
	EmbML/FLT	0.00	0.00	0.00	0.00	0.00	0.00
	EmbML/FXP32	-0.03	-0.07	-0.08	-51.89	+0.24	+0.03
	EmbML/FXP16	-48.18	**-0.07**	-0.77	-79.61	-31.13	**-0.13**
MLPClassifier	Desktop	95.96	92.46	91.41	96.43	89.96	98.54
	EmbML/FLT	0.00	0.00	0.00	0.00	0.00	0.00
	EmbML/FXP32	+0.16	+0.14	+0.43	-0.17	-0.09	-0.16
	EmbML/FXP16	-39.52	**-87.34**	-27.32	-79.76	-32.19	-60.22
SVC (polynomial kernel)	Desktop	98.47	77.17	64.78	98.87	90.75	93.95
	EmbML/FLT	-46.91	0.00	0.00	-1.84	-19.24	-
	EmbML/FXP32	-48.25	0.00	0.00	-80.42	-81.56	-
	EmbML/FXP16	-48.47	-0.36	-29.56	-80.42	-80.62	-
SVC (RBF kernel)	Desktop	58.53	88.62	86.51	21.63	18.69	95.28
	EmbML/FLT	-	0.00	0.00	-	0.00	-
	EmbML/FXP32	-	+0.14	-0.43	-	-0.36	-
	EmbML/FXP16	-8.53	-67.92	-51.29	0.00	-8.80	-76.41

only supports single-precision (32 bits). We confirmed this fact by executing, on a desktop, the EmbML output codes for these cases using both representations.

The reduction in precision for *SVC* models affects intermediate calculations depending on various factors intrinsic to each dataset (*e.g.*, the ranges of attribute values and model weights) that also explain unstable results for a given model on different datasets. Moreover, in most cases, there is not a significant change in accuracy when using FXP32 compared to FLT. On the other hand, the use of FXP16 can cause a notable reduction in accuracy for most classifiers due to several reasons. For instance, underflow[21] and overflow occur regularly (from 26.64% to 38.71%) in the arithmetic operations of the cases highlighted in red in Table V, associated with high accuracy losses. Differently, they happen less frequently (from 14.78% to 19.07%) in the operations of the cases highlighted in green, associated with low accuracy losses. Also, the results imply that FXP16 may be more sensitive to the choices of n and m, which are not optimal in our experiments and can negatively affect accuracy.

[21]We consider underflow when it rounds a non-zero real number to zero, possibly canceling out results of subsequent multiplications.

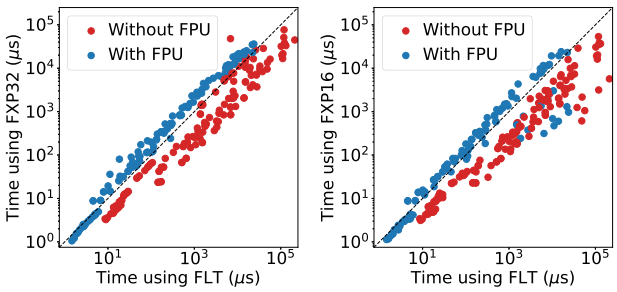


Fig. 3. Run-time comparison for floating-point and fixed-point formats for FXP32 (left) and FXP16 (right).

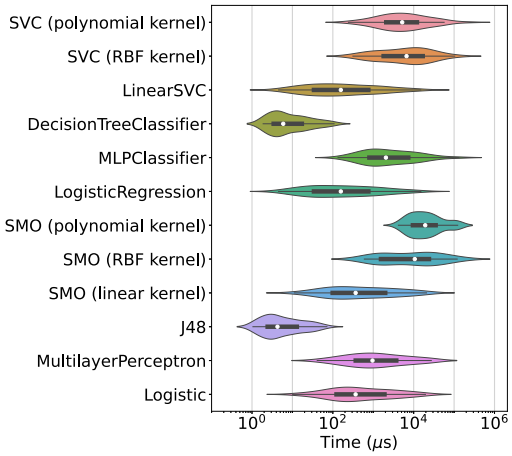


Fig. 4. Run-time comparison for all classifiers.

B. Classification Time

Next, we compare the average time that each model spent to classify an instance. Fig. 3 shows the executions of the EmbML codes using different representations for real numbers. In these graphs, the coordinates of each point represent the results of average classification time of FLT and FXP32 (*left*); and FLT and FXP16 (*right*) – for the same classifier, microcontroller, and dataset. In the microcontrollers that lack an FPU, we can observe that fixed-point versions achieve lower classification time than FLT. However, we do not see such an improvement in microcontrollers with FPU.

In Fig. 4, we present the average classification time per classifier class. This graph combines all results of each classifier obtained from executing it with every microcontroller and dataset and presents them in a way to contrast their behavior. From it, we can notice that the decision tree models usually deliver the lowest classification time. The logistic regression and SVM (linear kernel) models have similar performances, reaching the overall second-best results. The MLP models achieve an intermediate performance: faster than the SVM with polynomial and RBF kernels, but usually slower than the others. The SVM models with the polynomial and RBF kernels perform the worst results.

C. Memory Usage

The last metric evaluated from classifier performance is memory consumption. We separately compare data (SRAM) and program (flash) memories among the supported classifiers.

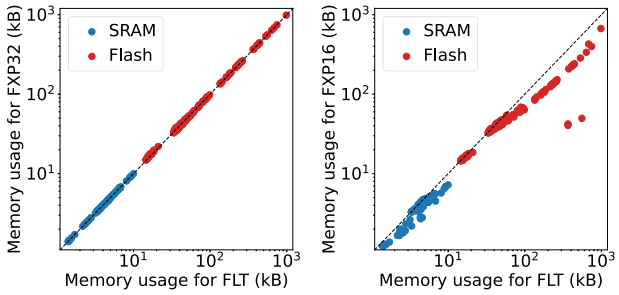


Fig. 5. Memory usage comparison for floating-point and fixed-point formats for FXP32 (left) and FXP16 (right).

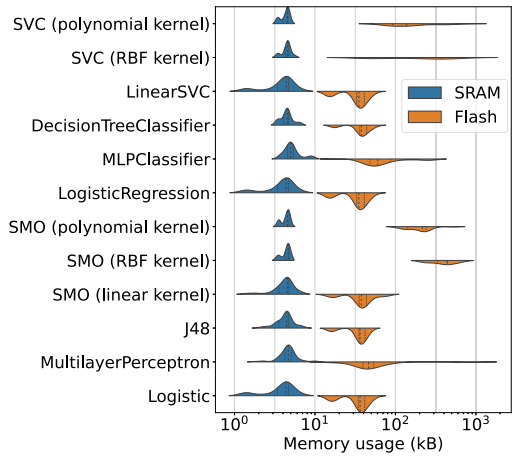


Fig. 6. Memory usage comparison for all classifier.

In Fig. 5, we show the memory usage of classifiers using FXP32 (*left*) and FXP16 (*right*) compared to FLT. These graphs reveal that there is no advantage of employing FXP32 in terms of this analysis. However, FXP16 representation can decrease memory consumption.

Fig. 6 displays the analysis of memory usage per classifier class, including results from all combinations of microcontrollers and datasets. We can observe that the decision tree, logistic regression, and SVM (linear kernel) models achieve the best memory performance. On the other hand, the SVM classifiers with polynomial and RBF kernels obtain the highest memory consumption. The MLP models again have an intermediate performance in most cases compared to the two previously defined groups.

VI. Evaluation of Code Modifications

To understand if the proposed modifications positively affect the time performance and do not cause a negative effect on the accuracy rates, we experimentally evaluate the impact of each one. It includes the analysis of approximations for sigmoid function in MLP models and transforming decision trees from an iterative structure to if-then-else statements. Since the previous section already assessed the impact of using fixed-point, this modification is not individually analyzed.

A. Approximations for Sigmoid Function in MLP

Table VI and Table VII present the accuracy rates estimated in each test set through applying the approximation functions

TABLE VI
ACCURACY (%) FOR THE *MultilayerPerceptron* MODELS

Classifier	Version	D1	D2	D3	D4	D5	D6
Original sigmoid	Desktop	98.67	89.19	90.29	92.84	80.46	93.62
	EmbML/FLT	0.00	+0.07	0.00	0.00	0.00	0.00
	EmbML/FXP32	-0.02	+1.14	+0.17	+0.02	+0.12	+0.04
	EmbML/FXP16	-44.27	-0.57	-1.80	-74.46	-0.58	-0.90
$0.5 + 0.5x/(1 + \|x\|)$ function	EmbML/FLT	0.00	0.00	+0.09	+0.07	0.00	+0.07
	EmbML/FXP32	-0.02	0.00	+0.17	+0.14	+0.03	+0.10
	EmbML/FXP16	-44.32	-1.92	-1.80	-74.18	-0.94	-0.19
2-point PWL	EmbML/FLT	0.00	+1.71	-0.08	-0.12	-0.27	+0.07
	EmbML/FXP32	-0.02	+1.85	-0.17	-0.08	-0.24	+0.07
	EmbML/FXP16	-44.28	-0.50	-2.15	-74.32	-0.48	-0.93
4-point PWL	EmbML/FLT	0.00	+1.78	+0.26	+0.02	-0.06	+0.07
	EmbML/FXP32	-0.02	+1.78	+0.09	+0.02	-0.09	+0.04
	EmbML/FXP16	-44.28	-0.78	-1.63	-74.56	-0.30	-0.93

TABLE VII
ACCURACY (%) FOR THE *MLPClassifier* MODELS WITH SIGMOID

Classifier	Version	D1	D2	D3	D4	D5	D6
Original sigmoid	Desktop	98.39	92.25	90.72	74.58	91.78	98.64
	EmbML/FLT	0.00	0.00	0.00	0.00	0.00	0.00
	EmbML/FXP32	-0.17	-0.07	0.00	-0.19	+0.06	+0.07
	EmbML/FXP16	-17.18	-2.42	-10.99	-56.97	-3.98	-57.83
$0.5 + 0.5x/(1 + \|x\|)$ function	EmbML/FLT	0.00	-0.93	-0.43	-0.07	-0.18	+0.16
	EmbML/FXP32	-0.13	-0.93	-0.51	-0.19	-0.24	+0.07
	EmbML/FXP16	-11.57	-2.07	-8.16	-57.26	-5.28	-39.55
2-point PWL	EmbML/FLT	-0.03	+0.28	-0.34	0.00	-0.03	-0.06
	EmbML/FXP32	-0.21	+0.21	-0.08	-0.19	+0.09	-0.06
	EmbML/FXP16	-16.50	-2.63	-9.71	-57.02	-3.86	-59.51
4-point PWL	EmbML/FLT	+0.01	-0.22	+0.26	0.00	+0.03	-0.06
	EmbML/FXP32	-0.20	-0.14	+0.52	-0.19	+0.09	0.00
	EmbML/FXP16	-16.78	-4.13	-14.09	-56.97	-3.58	-56.92

provided by EmbML for substituting the sigmoid in MLP models of WEKA and scikit-learn, respectively. These tables contain the accuracy values for the different real numbers representations and also the accuracy obtained by MLP models using the original sigmoid function for comparison. The values from EmbML versions are shown as the relative difference to the desktop version with the original sigmoid.

These tables must be analyzed independently because WEKA and scikit-learn use different hyperparameter values to train a model. Contrasting the alternative modifications with the original sigmoid, the highest difference in accuracy for the *MultilayerPerceptron* models occurred in D2 using the 4-point PWL approximation and FLT. In this case, the accuracy rate increased from 89.26% (original sigmoid) to 90.97% (4-point PWL). As for the *MLPClassifier*, the maximum difference happened in D6 using the $0.5 + \frac{0.5x}{(1+|x|)}$ function and FXP16. The accuracy increased from 40.81% (original sigmoid) to 59.09% (approximation function) in this situation. As a general rule, the accuracy values from the modified models are relatively close to the original versions and can be acceptable in practice.

In Fig. 7, we exhibit the average classification time comparison for MLP models using the provided options for the sigmoid function. We can identify that these options produce similar time results in many cases. However, the use of PWL approximations can frequently decrease the classification time of MLP models, whereas not causing expressive changes in the accuracy rates. Consequently, these versions are attractive options to help improve the performance of MLP classifiers.

We do not include memory usage analysis, since the difference in this metric for using or not the modified classifiers is

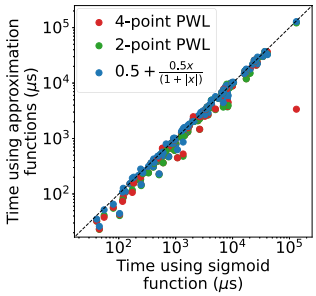


Fig. 7. Time comparison for approximation functions in MLP models.

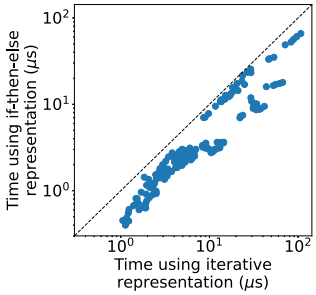


Fig. 8. Time comparison for iterative and if-then-else representations.

relatively inexpressive, considering that the sigmoid approximations do not affect the size of the classifier variables.

B. If-Then-Else Statements and Iterative Decision Trees

For decision trees, EmbML provides the options of using an iterative structure or if-then-else statements. In this analysis, the only difference between these options is structural and does not influence accuracy. As for memory usage, the amount of data to store is not affected, but code size may increase using if-then-else statements. However, both options achieved quite similar values for memory comparison: in the worst case, a classifier using if-then-else statements consumed only $2.55\,kB$ more memory than its iterative version – a maximum increase of 6.04%. Therefore, we exclusively focus on comparing mean classification time, as displayed in Fig. 8.

Fig. 8 helps to recognize the lower time results of the if-then-else representations of the decision tree models. For this reason, we recommend choosing this version when generating a decision tree classifier with EmbML.

VII. Comparison With Related Tools

To show that EmbML produces competitive classifiers, we evaluate its performance against classifiers generated by the following related tools: emlearn (version 0.10.1), m2cgen (version 0.5.0), sklearn-porter (version 0.7.4) and weka-porter (version 0.1.0). To make a consistent comparison, we selected only models from tools that have a direct correspondent in EmbML. For example, we consider the *MLPClassifier* model since both EmbML and emlearn support it. The chosen models and tools that support them are listed below.

- *J48* is supported by EmbML and weka-porter;
- *SVC* (polynomial and RBF kernels) is supported by EmbML, m2cgen, and sklearn-porter;
- *LinearSVC* is supported by EmbML, m2cgen, and sklearn-porter;
- *DecisionTreeClassifier* is supported by EmbML, emlearn, m2cgen, and sklearn-porter;
- *MLPClassifier* is supported by EmbML and emlearn;
- *LogisticRegression* is supported by EmbML and m2cgen.

TABLE VIII
Overall Time and Memory Comparison of Classifiers From EmbML and Related Tools

Dataset	Cases which EmbML classifiers achieve the lowest time results	Cases which EmbML classifiers achieve the smallest memory results	Total number of cases
D1	25 (71.43%)	27 (77.14%)	35
D2	27 (75.00%)	30 (83.33%)	36
D3	27 (77.14%)	30 (85.71%)	35
D4	22 (70.97%)	27 (87.10%)	31
D5	28 (77.78%)	35 (97.22%)	36
D6	23 (85.19%)	21 (77.78%)	27
Total	152 (76.00%)	170 (85.00%)	200

For decision trees, we used their if-then-else representations provided by EmbML. Also, we used the same trained model and test set to generate and evaluate the classifier versions. We executed each classifier with the datasets and microcontrollers to compare average classification time and memory usage.

Our analysis incorporates only the time or memory values associated with a high accuracy to prevent including poor solutions – *e.g.*, the FXP16 versions of EmbML classifiers are faster but usually have lower accuracy than their corresponding FLT version. Therefore, after combining all outcomes from the same microcontroller, dataset, and classifier, we determined the average accuracy of these results, and eliminated those associated with an accuracy lower than the average.

With the remaining results from this process, we show in Table VIII, for each dataset, the number of cases that EmbML classifiers accomplished the best time and memory performances, and the total number of cases – *i.e.*, combinations of microcontrollers and classifiers – that at least one classifier code was able to execute. Accordingly, the EmbML classifiers produced the best average classification time in at least 70.97% of the cases and the smallest memory consumption in at least 77.14%. These results reveal that EmbML classifiers often perform better than the other solutions, indicating that EmbML is an advantageous alternative.

VIII. Case Study: Intelligent Trap

In this section, we assess the EmbML classifiers using a practical application: an intelligent trap developed by our research group to classify and capture mosquitoes [1], [28]–[32]. The trap contains a low-cost optical sensor composed by a structure of parallel mirrors, an infrared LED, and an infrared phototransistor [32]. When a flying insect crosses this structure, the movements of its wings partially occlude the light, producing small variations captured by the phototransistor as an input signal [1]. Fig. 9 show the sensor and a signal generated by an *Aedes aegypti* mosquito. The trap's hardware processes the input signal to obtain features from its frequency spectrum [30], [31] and uses them in the classification process to determine the species and gender of flying insects.

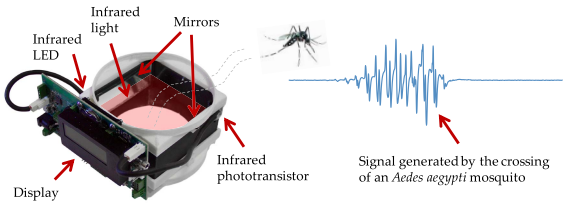


Fig. 9. Intelligent trap sensor and a signal example.

(a) Outside view of the cage (b) Intelligent trap inside the cage

Fig. 10. Arrangement of the cage and the intelligent trap for mosquitoes employed in the experiments.

We used the *Aedes aegypti*-sex dataset [19] to train and evaluate the classifiers because its data was obtained with the same optical sensor of the trap. We performed a grid search to define the best set of hyperparameters for each classifier class supported by EmbML. After that, we used the test set to evaluate the outcomes and selected the *J48* model with FXP32 format since it achieved the best results executing in the trap's microcontroller (MK20DX256VLH7): 98.92% of accuracy, 1.26 μs of average classification time, 4.2 kB of SRAM and 32.6 kB of flash. On average, the trap consumes 435.6 mW while waiting for insect crossings. When it processes and classifies an event, it consumes around 514.8 mW. Also, communication usually requires additional 36 mW, using Bluetooth Low Energy (BLE).

For each insect crossing event, we programmed the trap's microcontroller to perform the following tasks: read the input signal; extract the predictive features from it; classify it using the code produced by EmbML for the *J48* model; and activate the trap's fan to capture female *Aedes aegypti* and expel males.

To analyze the trap performance to capture mosquitoes selectively, we used a cage that allowed releasing only *Aedes aegypti* mosquitoes inside it, as illustrated in Fig. 10. It has dimensions of approximately $1.8m \times 1.8m \times 1.8m$ and includes double protection of mosquito netting fabric connected to a plastic pipe structure to create an isolated internal space. CO_2 was used to attract mosquitoes to the trap. We placed the cage in a room with controlled temperature and humidity.

This experiment was conducted in three rounds of approximately 24 hours each. At the beginning of a round, we inserted 30 *Aedes aegypti* mosquitoes (15 females and 15 males) inside the cage. At the end of every round, we manually captured, counted, and classified the insects outside and inside the trap, copied the produced event data, and reset the trap's microcontroller. Table IX exhibits the results obtained at the end of each round of this experiment.

The results demonstrate that the trap was effective in capturing female mosquitoes. It caught all the released female mosquitoes, but it also wrongly captured at least 20% of the males. This value is higher than expected from previous results, in which it was possible to classify correctly over 98% of the examples. One explanation is that while capturing a female mosquito, the trap could have caught some nearby males attracted by the female mosquitoes [33].

TABLE IX
RESULTS FROM THE INTELLIGENT TRAP EXPERIMENT

Day	Inside		Outside		Classified as Female	Total Captured	Total Events
	Female	Male	Female	Male			
1	15 (100%)	3 (20%)	0 (0%)	12 (80%)	17	18	56
2	15 (100%)	5 (33%)	0 (0%)	10 (67%)	17	20	34
3	15 (100%)	7 (47%)	0 (0%)	8 (53%)	23	22	73

IX. LIMITATIONS

EmbML is an easy-to-use tool that automatically converts models into C or C++ code for microcontrollers. Such simplicity comes with a few limitations. One of them is to employ fixed values for n and m in FXP16 and FXP32 representations. Although the user can choose the values for these two parameters, they must remain constant during the entire classification process. This limitation is because EmbML, as a general and multi-model conversion tool, does not impose assumptions about the range of attributes values or require pre-processing steps such as data normalization.

In our experiments, we use the same values of n and m for all datasets. Such a decision simplifies the experiments but impacts the model accuracy since these values do not provide the best representation for each attribute. Alternatively, we could assume an initial representation for all model parameters and input values by enforcing data normalization and change the number of fractional bits as we operate over the model. Such an approach is prevalent in the literature (see, for instance, [34]).

Our literature review is limited to similar tools that support embedded classifiers. However, there exists a large body of literature that implements application-specific classifiers. In some cases, these classifiers may be more efficient than our solution because they are well-tuned for a given problem. However, the process of building such classifiers is usually laborious and may require in-depth technical knowledge.

Finally, EmbML does not implement approximation functions to RBF kernel as it does for the sigmoid function. In general, our experiments showed that models with such kernels present high demands for memory and processing.

X. CONCLUSION AND FUTURE WORK

We presented the implementation details of EmbML tool, which automatically produces classifier codes from trained models to execute in resource-constrained hardware. It includes specific support for this type of hardware: avoidance of unnecessary use of SRAM, fixed-point operations for real numbers, approximations for the sigmoid function in MLP models, and if-then-else statements for representing decision trees. Our experimental analysis empirically demonstrates that EmbML classifiers maintain the accuracy rates obtained with the training tools, improve performance by using available modifications, and achieve competitive results compared to related tools. Moreover, we successfully implemented an EmbML classifier in a practical application that allows a trap

to capture selectively disease-vector mosquitoes. Future work will investigate improvements for cumbersome classifiers, such as SVM (polynomial and RBF) and ensemble models. For instance, [35] has proposed an approximation of the RBF kernel that can reduce the SVM model complexity as well.

REFERENCES

[1] G. E. A. P. A. Batista, E. J. Keogh, A. Mafra-Neto, and E. Rowton, "SIGKDD demo: Sensors and software to allow computational entomology, an emerging application of data mining," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2011, pp. 761–764.

[2] V. M. A. Souza, R. Giusti, and A. J. L. Batista, "Asfault: A low-cost system to evaluate pavement conditions in real-time using smartphones and machine learning," *Pervasive Mobile Comput.*, vol. 51, pp. 121–137, Dec. 2018.

[3] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. Reyes-Ortiz, "A public domain dataset for human activity recognition using smartphones," in *Proc. Eur. Symp. Artif. Neural Netw., Comput. Intell. Mach. Learn.*, 2013, pp. 437–442.

[4] L. Zhang and D. Zhang, "Domain adaptation extreme learning machines for drift compensation in E-nose systems," *IEEE Trans. Instrum. Meas.*, vol. 64, no. 7, pp. 1790–1801, Jul. 2015.

[5] L. Zhang, D. Zhang, X. Yin, and Y. Liu, "A novel semi-supervised learning approach in artificial olfaction for E-nose application," *IEEE Sensors J.*, vol. 16, no. 12, pp. 4919–4931, Jun. 2016.

[6] M. Tubaishat and S. Madria, "Sensor networks: An overview," *IEEE Potentials*, vol. 22, no. 2, pp. 20–23, Aug. 2003.

[7] J. Yick, B. Mukherjee, and D. Ghosal, "Wireless sensor network survey," *Comput. Netw.*, vol. 52, no. 12, pp. 2292–2330, Aug. 2008.

[8] L. Tsutsui da Silva, V. M. A. Souza, and G. E. A. P. A. Batista, "EmbML tool: Supporting the use of supervised learning algorithms in low-cost embedded systems," in *Proc. IEEE 31st Int. Conf. Tools Artif. Intell. (ICTAI)*, Nov. 2019, pp. 1633–1637.

[9] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: An update," *ACM SIGKDD Explor. Newslett.*, vol. 11, no. 1, pp. 10–18, 2009.

[10] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Oct. 2011.

[11] R. Sanchez-Iborra and A. F. Skarmeta, "TinyML-enabled frugal smart objects: Challenges and opportunities," *IEEE Circuits Syst. Mag.*, vol. 20, no. 3, pp. 4–18, 3rd Quart., 2020.

[12] C. Gupta *et al.*, "ProtoNN: Compressed and accurate knn for resource-scarce devices," in *Proc. Int. Conf. Mach. Learn.*, 2017, pp. 1331–1340.

[13] D. Dennis, C. Pabbaraju, H. V. Simhadri, and P. Jain, "Multiple instance learning for efficient sequential data classification on resource-constrained devices," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2018, pp. 10953–10964.

[14] A. Kusupati, M. Singh, K. Bhatia, A. Kumar, P. Jain, and M. Varma, "FastGRNN: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network," in *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2018, pp. 9017–9028.

[15] S. Gopinath, N. Ghanathe, V. Seshadri, and R. Sharma, "Compiling KB-sized machine learning models to tiny IoT devices," in *Proc. 40th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2019, pp. 79–95.

[16] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: Efficient neural network kernels for arm Cortex-M CPUs," 2018, *arXiv:1801.06601*.

[17] X. Wang, M. Magno, L. Cavigelli, and L. Benini, "FANN-on-MCU: An open-source toolkit for energy-efficient neural network inference at the edge of the Internet of Things," *IEEE Internet Things J.*, vol. 7, no. 5, pp. 4403–4417, May 2020.

[18] M. Vlăduţiu, *Computer Arithmetic: Algorithms Hardware Implementations*. Berlin, Germany: Springer, 2012.

[19] D. M. Dos Reis, A. G. Maletzke, and G. E. A. P. A. Batista, "Unsupervised context switch for classification tasks on data streams with recurrent concepts," in *Proc. 33rd Annu. ACM Symp. Appl. Comput.*, Apr. 2018, pp. 518–524.

[20] A. Vergara *et al.*, "Chemical gas sensor drift compensation using classifier ensembles," *Sens. Actuators B, Chem.*, vol. 166, pp. 320–329, May 2012.

[21] F. Alimoglu and E. Alpaydin, "Methods of combining multiple classifiers based on different representations for pen-based handwritten digit recognition," in *Proc. Turkish Artif. Intell. Artif. Neural Netw. Symp. (TAINN)*, 1996, pp. 1–5.

[22] *Atmel ATmega328/P DATASHEET*, Atmel Corporation, San Jose, CA, USA, Nov. 2016.

[23] *Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V DATASHEET*, Atmel Corporation, San Jose, CA, USA, Feb. 2014.

[24] *Atmel SAM3X/SAM3A Series DATASHEET*, Atmel Corporation, San Jose, CA, USA, Mar. 2014.

[25] *K20 Sub-Family Data Sheet*, Freescale Semiconductor, Austin, TX, USA, Nov. 2012.

[26] *Kinetis K64F Sub-Family Data Sheet*, NXP Semiconductors, Eindhoven, The Netherlands, Nov. 2016.

[27] *Kinetis K66 Sub-Family*, NXP Semiconductors, Eindhoven, The Netherlands, Apr. 2017.

[28] V. M. A. D. Souza, D. F. Silva, and G. E. A. P. A. Batista, "Classification of data streams applied to insect recognition: Initial results," in *Proc. Brazilian Conf. Intell. Syst.*, Oct. 2013, pp. 76–81.

[29] Y. Chen, A. Why, G. Batista, A. Mafra-Neto, and E. Keogh, "Flying insect detection and classification with inexpensive sensors," *J. Vis. Exp.*, no. 92, Oct. 2014, Art. no. e52111.

[30] Y. Qi, G. T. Cinar, V. M. A. Souza, G. E. A. P. A. Batista, Y. Wang, and J. C. Principe, "Effective insect recognition using a stacked autoencoder with maximum correntropy criterion," in *Proc. Int. Joint Conf. Neural Netw.*, 2015, pp. 1–7.

[31] D. F. Silva, V. M. A. Souza, D. P. W. Ellis, E. J. Keogh, and G. E. A. P. A. Batista, "Exploring low cost laser sensors to identify flying insect species," *J. Intell. Robot. Syst.*, vol. 80, no. S1, pp. 313–330, Dec. 2015.

[32] V. M. A. Souza, D. M. Dos Reis, A. G. Maletzke, and G. E. A. P. A. Batista, "Challenges in benchmarking stream learning algorithms with real-world data," *Data Mining Knowl. Discovery*, vol. 34, no. 6, pp. 1805–1858, Nov. 2020.

[33] P. Belton and R. A. Costello, "Flight sounds of the females of some mosquitoes of Western Canada," *Entomol. Exp. Appl.*, vol. 26, no. 1, pp. 105–114, Jul. 1979.

[34] G. Cerutti, R. Prasad, A. Brutti, and E. Farella, "Compact recurrent neural networks for acoustic event detection on low-energy low-complexity platforms," *IEEE J. Sel. Topics Signal Process.*, vol. 14, no. 4, pp. 654–664, May 2020.

[35] M. Claesen, F. De Smet, J. A. K. Suykens, and B. De Moor, "Fast prediction with SVM models containing RBF kernels," 2014, *arXiv:1403.0736*.

Lucas Tsutsui da Silva received the B.Sc. degree in computer engineering from the Federal University of Mato Grosso do Sul, Brazil, in 2016, and the M.Sc. degree in computer science and computational mathematics from the University of São Paulo, Brazil, in 2020. His research interests include machine learning, data mining, and edge computing.

Vinicius M. A. Souza received the Ph.D. degree in computer science from the University of São Paulo, Brazil, in 2016. He is an Associate Professor with the Graduate Program in Informatics, Pontifical Catholic University of Paraná, Brazil. Earlier, he held a postdoctoral position at The University of New Mexico, USA. Dr. Souza has authored articles in top-tier peer-reviewed conferences and journals, including *Data Mining and Knowledge Discovery*, *Information Sciences*, *Knowledge and Information Systems*, and ICDM. His research interests include data mining, data streams, and time series.

Gustavo E. A. P. A. Batista received the Ph.D. degree in computer science from the University of São Paulo, Brazil, in 2003. In 2007, he joined the University of São Paulo as an Assistant Professor and became an Associate Professor in 2016. In 2018, he joined the University of New South Wales, Australia, as an Associate Professor. Dr. Batista has more than 100 papers in conferences and journals. He has served as a Program Committee Member for conferences, such as ACM-KDD, IEEE-ICDM, and IJCAI, and an Editorial Board Member for the *Machine Learning* journal.