UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

Evaluating classification models for resource-constrained hardware

Lucas Tsutsui da Silva

Dissertação de Mestrado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

ICMC USP
SÃO CARLOS

Lucas Tsutsui da Silva

Evaluating classification models for resource-constrained hardware

Dissertation submitted to the Institute of Mathematics and Computer Sciences – ICMC-USP – in accordance with the requirements of the Computer and Mathematical Sciences Graduate Program, for the degree of Master in Science. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Gustavo Enrique de Almeida Prado Alves Batista

USP – São Carlos
November 2020

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

da Silva, Lucas Tsutsui
d111e Evaluating classification models for resource-constrained hardware / Lucas Tsutsui da Silva; orientador Gustavo Enrique de Almeida Prado Alves Batista. -- São Carlos, 2020.
141 p.

Dissertação (Mestrado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2020.

1. Machine learning. 2. Classification. 3. Embedded classifier. 4. WEKA. 5. scikit-learn. I. Batista, Gustavo Enrique de Almeida Prado Alves, orient. II. Título.

Bibliotecários responsáveis pela estrutura de catalogação da publicação de acordo com a AACR2:
Gláucia Maria Saia Cristianini - CRB - 8/4938
Juliana de Souza Moraes - CRB - 8/6176

Lucas Tsutsui da Silva

Avaliação de modelos de classificação para hardware com limitação de recursos

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências – Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Gustavo Enrique de Almeida Prado Alves Batista

USP – São Carlos
Novembro de 2020

ACKNOWLEDGEMENTS

Meu primeiro desejo nestes agradecimentos é que atinjam, de maneira geral, todas as pessoas que compartilharam comigo alguns passos da jornada que me fez chegar até este momento. Reconheço o privilégio de ter tido pessoas incríveis ao meu lado durante várias etapas da vida e sou grato a todas elas por isso, uma vez que, como disse BoJack Horseman, *"[...] in this terrifying world, all we have are the connections that we make."*

Em especial, gostaria de dedicar o restante deste espaço para agradecer àqueles que tiveram participação direta em minha vida, principalmente no período que dediquei ao mestrado. Começo, portanto, agradecendo à minha mãe e ao meu pai, que investiram em minha educação, sempre dedicaram todos seus esforços para me oferecer o melhor que podiam e me incentivam a seguir em busca dos meus sonhos. Aos meus irmãos, pela amizade e parceria desde sempre e por me apoiarem em minhas escolhas. À minha namorada, pela companhia diária em todas situações, por auxiliar nas minhas decisões com sua sabedoria e sensatez, e por fazer minha vida mais feliz. Também a todos demais familiares que, de alguma forma, estiveram presentes na minha vida durante esse período.

Agradeço ao meu orientador, que me acolheu em seu grupo de pesquisa, me deu a oportunidade de trabalhar em um relevante projeto e me possibilitou ter experiências que agregaram importantes conquistas e aprendizados à minha carreira profissional. Além disso, sou grato por sempre estar disponível para me atender e ter oferecido sua ajuda em diversos momentos.

Agradeço ao Vinicius, que colaborou com este trabalho desde o início, revisando o primeiro esboço de projeto, dando valiosas sugestões e ajudando na escrita dos artigos publicados. Ao André e à Barbara, por terem me auxiliado na preparação e realização dos experimentos com os mosquitos. Também, a todos integrantes do LABIC, professores e alunos, que se dedicam para fortalecer o ambiente de aprendizado, amizade e colaboração.

Aos amigos Daniela e Raphael, que me fizeram companhia enquanto morei em São Carlos. Também, às demais amizades que mantiveram o contato e o apreço nesse período, apesar da vida ter inevitavelmente colocado muitos de nós em caminhos diferentes.

Ao ICMC-USP, seus docentes e demais servidores, pela estrutura oferecida e empenho em construir um ambiente de excelência e reconhecimento internacional. Aos professores da FACOM-UFMS, que me apresentaram a Computação e foram responsáveis por construir minha base de conhecimento nessa área.

Finalmente, agradeço ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), processo 166919/2017-9, e à *United States Agency for International Development (USAID)*, grant AID-OAA-F-16-00072, pelo suporte financeiro que possibilitou o desenvolvimento deste trabalho.

“Para examinar a verdade é necessário,
pelo menos uma vez na vida, pôr todas as
coisas em dúvida, tanto quanto se puder.”
(René Descartes)

RESUMO

SILVA, L. T. **Avaliação de modelos de classificação para hardware com limitação de recursos**. 2020. 141 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2020.

Aprendizado de Máquina (AM) está se tornando uma tecnologia ubíqua empregada em muitas aplicações do mundo real em diversas áreas, como agricultura, saúde, entomologia e engenharia. Em algumas aplicações, sensores monitoram o ambiente, enquanto algoritmos de AM supervisionado são responsáveis por interpretar os dados para tomar uma decisão automática. Geralmente, esses dispositivos enfrentam três restrições principais: consumo de energia, custo e falta de infraestrutura. A maioria desses desafios pode ser melhor resolvida com a implementação de classificadores de AM no hardware que monitora o ambiente. Portanto, precisamos de classificadores altamente eficientes, adequados para serem executados em hardware com poucos recursos. No entanto, esse cenário entra em conflito com o estado-da-prática de AM, no qual os classificadores são frequentemente implementados em linguagens interpretadas de alto nível (*e.g.*, Java ou Python), fazem uso irrestrito de operações de ponto flutuante e assumem muita disponibilidade de recursos, como memória, processamento e energia. Neste trabalho, apresentamos uma ferramenta de software chamada *Embedded Machine Learning (EmbML)* que implementa um *pipeline* para desenvolver classificadores para microcontroladores de baixa potência. Esse *pipeline* começa com o aprendizado de um classificador em um computador *desktop* ou servidor, utilizando pacotes ou bibliotecas de software populares como WEKA ou scikit-learn. A ferramenta EmbML converte o classificador em um código C++ adaptado com suporte para hardware com recursos limitados, como prevenção do uso desnecessário da memória principal e implementação de operações de ponto fixo para números não-inteiros. Nossa avaliação experimental com conjuntos de dados de *benchmark* e uma variedade de microcontroladores mostra que os classificadores da ferramenta EmbML alcançam resultados competitivos em termos de acurácia, tempo de classificação e custo de memória. Comparados aos classificadores de algumas ferramentas relacionadas existentes, os nossos obtiveram o melhor desempenho de tempo e memória em pelo menos 70% dos casos. Por fim, realizamos experimentos em uma aplicação real para descrever o *pipeline* completo de uso da ferramenta EmbML e avaliar seus classificadores com uma armadilha inteligente para classificar e capturar insetos alados.

Palavras-chave: Aprendizado de máquina, Classificação, Classificador embarcado, WEKA, scikit-learn.

ABSTRACT

SILVA, L. T. **Evaluating classification models for resource-constrained hardware**. 2020. 141 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2020.

Machine Learning (ML) is becoming a ubiquitous technology employed in many real-world applications in diverse areas such as agriculture, human health, entomology, and engineering. In some applications, sensors measure the environment while supervised ML algorithms are responsible for interpreting these data to make an automatic decision. Generally, these devices face three main restrictions: power consumption, cost, and lack of infrastructure. Most of these challenges can be better addressed by embedding ML classifiers in the hardware that senses the environment. Thus, we need highly-efficient classifiers suitable to execute in unresourceful hardware. However, this scenario conflicts with the state-of-practice of ML, in which classifiers are frequently implemented in high-level interpreted languages (*e.g.*, Java or Python), make unrestricted use of floating-point operations and assume plenty of resources such as memory, processing and energy. In this work, we present a software tool named *Embedded Machine Learning (EmbML)* that implements a pipeline to develop classifiers for low-power microcontrollers. This pipeline starts with learning a classifier in a desktop or server computer using popular software packages or libraries as WEKA or scikit-learn. EmbML converts the classifier into a carefully crafted C++ code with support for resource-constrained hardware, such as the avoidance of unnecessary use of main memory and implementation of fixed-point operations for non-integer numbers. Our experimental evaluation on benchmark datasets and a variety of microcontrollers shows that EmbML classifiers achieve competitive results in terms of accuracy, classification time, and memory cost. Compared to classifiers from some existing related tools, ours achieved the best time and memory performances in at least 70% of the cases. Lastly, we conduct experiments in a real-world application to describe the complete pipeline for using EmbML and assessing its classifiers with an intelligent trap to classify and capture flying insects.

Keywords: Machine learning, Classification, Embedded classifier, WEKA, scikit-learn.

LIST OF FIGURES

LIST OF CHARTS

LIST OF ALGORITHMS

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

AI	Artificial Intelligence
API	Application Programming Interface
BSD	Berkeley Software Distribution
CART	Classification And Regression Tree
CMSIS	Cortex Microcontroller Software Interface Standard
CO2	carbon dioxide
DFT	Discrete Fourier Transform
DSP	Digital Signal Processor
EmbML	Embedded Machine Learning
FANN	Fast Artificial Neural Network
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
FPU	Floating-Point Unit
GPU	Graphics Processing Unit
GUI	Graphical User Interface
ID3	Iterative Dichotomiser 3
IDFT	Inverse Discrete Fourier Transform
IEEE	Institute of Electrical and Electronics Engineers
IoT	Internet of Things
JAR	Java Archive
kNN	k-Nearest Neighbors
LED	light-emitting diode
LSB	Least Significant Bit
MATLAB	Matrix Laboratory
ML	Machine Learning
MLP	Multilayer Perceptron
NLLS	Nonlinear Least-Squares
NN	Neural Networks
PWL	Piecewise Linear
RBF	Radial Basis Fuction
ReLU	Rectified Linear Unit

RTC	real-time clock
SD	standard deviation
SRAM	Static Random-Access Memory
SVM	Support Vector Machine
WBF	Wingbeat Frequency
WEKA	Waikato Environment for Knowledge Analysis

CONTENTS

CHAPTER

1

INTRODUCTION

1.1 Background

With the recent technological advancement, computational applications have been developed to solve problems existing in various areas in society. There has also been a growing interest from the scientific community and industry in the development of practical applications that use concepts related to Artificial Intelligence (AI), especially Machine Learning (ML), for tasks involving, for instance, knowledge discovery in large databases or automatic classification. As a result of this effort, ML is becoming a ubiquitous technology with applications in areas as diverse as agriculture, human health, entomology, engineering, transportation, and sociology. Many of these applications use ML algorithms as the primary approach to convert a series of low-level signals sensed from the environment into a higher-level interpretation of the data.

For instance, in human health, classifiers convert accelerometer and gyroscope data into different movements, allowing the detection of falls that constitute a significant problem in the elderly population (CHEN *et al.*, 2006; LUŠTREK; KALUŽA, 2009; CHOI; RALHAN; KO, 2011; CHELLI; PÄTZOLD, 2019). In agriculture, sensors support real-time soil management, allowing a reduction in the use of material resources such as water (COOPERSMITH *et al.*, 2014). In transportation, autonomous vehicles use ML algorithms to interpret data collected from several sensors and produce real-time driving decisions (FERNANDES *et al.*, 2014; BOJARSKI *et al.*, 2016). In entomology, traps can capture and sort mosquitoes by species, providing valuable information to plan control activities (BATISTA *et al.*, 2011a; SOUZA; SILVA; BATISTA, 2013; SILVA *et al.*, 2015).

1.2 Justification and Motivation

In applications which we need to sense, measure, and gather information from the environment, we frequently face three main restrictions (TUBAISHAT; MADRIA, 2003): power

consumption, cost, and lack of infrastructure. For example, sensors often have a battery as its main power source, so efficient use of power allows them to run for more extended periods. Price is a significant factor that hinders scaling in several areas, such as agriculture. Infrastructure assumptions such as reliable internet connection or power supply frequently do not hold when, for instance, surveying mosquitoes in low-income countries.

Most of these challenges can be better addressed by embedding ML classifiers in the hardware that senses the environment, 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 (YICK; MUKHERJEE; GHOSAL, 2008). Since sensor devices have restricted memory capacity and can be deployed in difficult-to-access areas, they often use radio for wireless communication to transfer the data to a base station or to the cloud in case of Internet of Things (IoT) applications (ATZORI; IERA; MORABITO, 2010; BOTTA *et al.*, 2016). Therefore, these smart sensors are more power-efficient since they eliminate the need for communicating all the raw data. Instead, they can only report events of interest, such as a dry soil crop area that needs watering or the capture of a disease-vector mosquito.

However, for this approach to be cost-effective, we need highly-efficient classifiers suitable to execute in unresourceful 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 interpreted languages such as Java or Python, make unrestricted use of floating-point operations and assume plenty of resources such as memory, processing and energy. Moreover, these classifiers usually execute on powerful computer servers that support intensive parallel computing and contain specific hardware such as Graphics Processing Unit (GPU) to speed up processing operations.

The problem of embedding ML models in hardware with severe restrictions in memory, processing, and power also imposes a different perception of the classifier analysis. In this case, metrics such as memory consumption and processing time become as important as the error rate of the ML model. Evaluating those metrics are also extremely necessary to guarantee that the classifier fits in the available memory and returns the results in a viable time, especially when the application requires real-time processing – *i.e.*, when the timing requirement of each task must be individually satisfied (STANKOVIC, 1988). This is a common scenario in data stream mining problems that demand to constantly process a high volume of data in limited time (KREMPL *et al.*, 2014).

In real-world solutions, there are several examples of sensing applications with a highly restricted time between data gathering and acting. For instance, when a flying insect crosses the optical sensor of the intelligent trap proposed by Batista *et al.* (2011a) and Silva *et al.* (2013), the trap must be agile in deciding whether or not to capture the insect before it flies away. In Shi *et al.* (2009), a wearable device must be able to recognize a human fall before it fully completes

in order to activate an airbag system that will protect the individual from impact injuries.

Considering that critical applications may require immediate actions, it is not feasible to rely on transmitting the data to an external system and receiving the response to make a decision. Instead, local processing might be the best solution for these cases, since it decreases the latency of computing a decision. This approach can also help increase the privacy of user data for some applications because it prevents propagating the data to external devices. Therefore, to achieve these benefits, we need to be able to implement the ML models directly in the resource-constrained hardware.

1.3 Proposal

In order to help overcome the problems previously discussed, this work presents a software tool named Embedded Machine Learning (EmbML) (SILVA; SOUZA; BATISTA, 2019a; SILVA; SOUZA; BATISTA, 2019b) that implements a pipeline to develop classifiers for resource-constrained hardware. This pipeline 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) (HALL *et al.*, 2009) and scikit-learn (PEDREGOSA *et al.*, 2011). Next, the proposed tool converts the trained classifier into a carefully crafted C++ code with support for unresourceful hardware, such as the avoidance of unnecessary use of main memory and implementation of fixed-point operations for non-integer numbers.

It is important to highlight that EmbML does not support the learning step in the embedded hardware. We advocate that, for most of the learning algorithms, the search for the model parameters is too expensive to execute in a low-power microcontroller. However, most ML algorithms output classifiers that are highly efficient, including the ones supported in our tool: Logistic Regression, Decision Tree, Multilayer Perceptron (MLP), and Support Vector Machine (SVM). The choice of these approaches is also based on the literature that reports the successful application of these methods with embedded hardware (FAROOQ *et al.*, 2010; RÚA *et al.*, 2012; TOCCHETTO *et al.*, 2014; SAMPAIO *et al.*, 2017).

Due to the application purposes, the main focus of EmbML is to produce classifiers to execute in low-power microcontroller-based systems, instead of more robust hardware solutions such as GPU, Digital Signal Processor (DSP), and Field Programmable Gate Array (FPGA). By doing that, we believe that it is possible to provide extensive support for a larger number of embedded applications, once microcontrollers are usually cheaper and easier to program using popular programming languages. Also, in many applications that include sensing, the classifier represents only a piece of a more complex system that may involve more expensive subtasks, such as signal and image processing, and define the requirement for powerful hardware.

This work also includes a case study in which we present and analyze the complete pipeline for producing classifiers for a low-power hardware in an intelligent trap for flying

insects (BATISTA *et al.*, 2011a; SOUZA; SILVA; BATISTA, 2013; SILVA *et al.*, 2015). With this practical case, we focus on explaining each step of this pipeline in detail and illustrating a performance evaluation conducted using EmbML classifiers.

1.4 Objectives and Contributions

The main objective of this work is to provide a simple and open-source tool that can be used by researchers and practitioners in the development of classifiers for low-power microcontrollers. In other words, EmbML objectives include:

- having its source code available to the ML community for free usage and improvement;
- generating microcontroller-tailored classifier code that implements specific modifications to optimize its execution in resource-constrained hardware;
- and providing a variety of supported classification models, supplying options to the end-user, given that no single ML classifier is optimal for all applications (FERNÁNDEZ-DELGADO *et al.*, 2014).

The overall contribution of this dissertation is to present a software tool that automates the process of producing classifiers for low-power microcontroller-based embedded systems. We assume that it is possible to adapt off-board-trained classification models trained in popular ML tools for use on these systems.

A detailed presentation concerning implementation issues of the proposed tool, the analysis of results with benchmark datasets and different microcontrollers, as well as the comparison against other similar software, also constitute an important contribution of this work since most of the previously proposed and related tools are available without such scientific rigor. Thus, this work aims to support both researchers and practitioners interested in embedding their supervised learning models into resource-constrained hardware using open-source software.

The specific contributions are as follows:

- Assist the process of using classifiers in unresourceful embedded systems;
- Support the development of technological solutions that employ classification models in low-power applications;
- Provide a range of options of classification models produced by different learning algorithms to execute in microcontrollers;
- Enable the production of modified classifier codes that can decrease memory consumption and processing time;

- Produce a comparative analysis using accuracy, classification time, and memory usage to evaluate the performance of EmbML classifiers against those produced by some similar tools;
- In a practical application, present the detailed pipeline of using EmbML to embed a classifier into resource-constrained hardware and evaluate its performance.

1.5 Dissertation Organization

The next chapters of this dissertation are organized as follows:

- Chapter 2 presents a literature review of related tools and possible applications;
- Chapter 3 discusses the relevant ML concepts, focusing on details of the supported classification algorithms;
- Chapter 4 explains the pipeline process for using the implemented tool and describes its features;
- Chapter 5 presents an in-depth study of the supported classifiers from WEKA and scikit-learn;
- Chapter 6 shows a comparative analysis of the classifier performances using different benchmark datasets and microcontrollers;
- Chapter 7 considers a real-world application to explain the pipeline for producing classifiers using EmbML;
- Finally, Chapter 8 presents our conclusions as well as this work's limitations. We also discuss possible pathways for future work.

1.6 Publications

The intermediate results from this work produced two papers published at the *14º Simpósio Brasileiro de Automação Inteligente* (SILVA; SOUZA; BATISTA, 2019b) and the *2019 IEEE 31st International Conference on Tools with Artificial Intelligence* (SILVA; SOUZA; BATISTA, 2019a).

CHAPTER

2

RELATED WORK

2.1 Initial Considerations

In this chapter, we review the most related research in the literature to the proposal of this work. First, we present existing tools that allow obtaining classifiers for microcontrollers using models trained in popular ML tools. The main goals are to present the features of these related tools – such as supported models, optimizations for unresourceful hardware, and programming languages of the generated classifier source codes – and how they differ from EmbML. Then, we examine works involving successful applications of ML models in resource-constrained embedded systems. We later explain how similar projects can take advantage of using EmbML in their analysis.

2.2 Related Tools

There is a good range of tools in the literature that converts classifiers into source code. However, all these tools fail in, at least, one of the objectives of **EmbML**, presented in Section 1.4. This section briefly summarizes the most relevant related tools in the literature.

Sklearn-porter[1] is a popular tool to convert classification and regression models built off-board using the Python ML library scikit-learn. This tool supports a wide range of programming languages, including Java, JavaScript, C, Go, PHP, and Ruby, as well as several classifiers such as SVM, Decision Trees, Random Forest, Naive Bayes, k-Nearest Neighbors (kNN), and MLP. Chart 1 presents all classifiers and programming languages supported by this tool. Unfortunately, the **sklearn-porter** does not provide any modification in the output classifier codes to support unresourceful hardware. For instance, they do not offer an efficient usage of data memory or any option to optimize operations with real numbers. Therefore, we assume this type of hardware is

[1] <https://github.com/nok/sklearn-porter>

not the focus of the tool.

Chart 1 – Classifiers and programming languages supported by **sklearn-porter** (version 0.7.3).

Scikit-learn classifier	***Java***	***JavaScript***	***C***	***Go***	***PHP***	***Ruby***
svm.SVC	✓	✓	✓		✓	✓
svm.NuSVC	✓	✓	✓		✓	✓
svm.LinearSVC	✓	✓	✓	✓	✓	✓
tree.DecisionTreeClassifier	✓	✓	✓	✓	✓	✓
ensemble.RandomForestClassifier	✓	✓	✓	✓	✓	✓
ensemble.ExtraTreesClassifier	✓	✓	✓		✓	✓
ensemble.AdaBoostClassifier	✓	✓	✓			
neighbors.KNeighborsClassifier	✓	✓				
naive_bayes.GaussianNB	✓	✓				
naive_bayes.BernoulliNB	✓	✓				
neural_network.MLPClassifier	✓	✓				
neural_network.MLPRegressor		✓				

Weka-porter[2] is a similar, but a more restricted project focused on the popular WEKA software package. This tool converts J48 decision tree classifiers into C, Java and JavaScript codes. Although the author indicates that this tool can be used to implement embedded classifiers, the lack of options for classification algorithms – such as SVM and Neural Networks (NN) – restricts its applicability.

Similarly to **weka-porter**, there are a considerable number of tools that are specialized in transforming decision tree models into C++ source code. Although the reasons for such prevalence are not clear, it is reasonable to presume that it is due to the direct mapping of these models into if-then-else statements. Some examples are:

- **J48toCPP**[3] that supports *J48* classifiers from WEKA;
- **C4.5 decision tree generator**[4] that converts *C4.5* models from WEKA;
- and **DecisionTreeToCpp**[5] converts *DecisionTreeClassifier* models from scikit-learn.

SVM is another model that has a good number of conversion tools to C++. Two tools based on the LIBSVM library (CHANG; LIN, 2011) and developed for microcontrollers are:

- **mSVM**[6] that includes support to fixed-point arithmetic;

[2] <https://github.com/nok/weka-porter>
[3] <https://github.com/mru00/J48toCPP>
[4] <https://github.com/hatc/C4.5-decision-tree-cpp>
[5] <https://github.com/papkov/DecisionTreeToCpp>
[6] <https://github.com/chenguangshen/mSVM>

- and **uLIBSVM**[7] that provides a simplified version of SVM-predict function from LIBSVM; however, without support for fixed-point representation.

Although both tools allow the conversion of SVM classifiers to run in microcontrollers, an explicit limitation is the lack of support for a more diverse set of algorithms.

M2cgen[8] is another option that can convert ML models, trained with scikit-learn, into native code in Python, Java, C, JavaScript, PHP, R, Go, and others. It supports a variety of classification and regression models such as Logistic Regression, SVM, Decision Tree, Random Forest, XGBoost, and others. But, similar to **sklearn-porter**, this tool does not provide any source code adaptation specific to run in a microcontroller environment.

Emlearn[9] is one of the most similar tool to the one proposed in this work. It generates source code in the C programming language from models built with scikit-learn or Keras. The tool presents support to the following algorithms: Decision Trees, Naive Bayes, MLP, and Random Forest. This tool is specifically designed to support embedded devices and has features such as avoiding the usage of dynamic memory allocation and standard libraries such as stdlib, as well as fixed-point representation for Naive Bayes classifiers. Despite those advantages, it has little diversity of classification models, not supporting popular algorithms on embedded systems such as SVM. Also, the Naive Bayes classifier is the only one currently able to use fixed-point arithmetic.

The popular TensorFlow[10] platform also provides support to run its models in embedded microcontrollers. TensorFlow is an ML system that operates on a large scale and in heterogeneous environments and supports a variety of applications, with a focus on training and inference on deep neural networks (ABADI *et al.*, 2016). A set of its tools, called TensorFlow Lite, enables applying its models on mobile, embedded, and IoT devices. Particularly, **TensorFlow Lite for Microcontrollers** is a library written in C++ programming language and designed to execute TensorFlow ML models on 32-bit hardware such as microcontrollers and other devices with memory constraints. As an option to decrease the model size and memory usage, it allows applying a post-training quantization technique that reduces the precision of the 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++ Application Programming Interface (API) requiring manual memory management, and training is not supported. It is also possible to recognize a restricted set of supported classifiers – only NNs – when compared to **EmbML**.

EdgeML[11] is a library written in Python using Tensorflow and PyTorch that generates

[7] <https://github.com/PJayChen/uLIBSVM>
[8] <https://github.com/BayesWitnesses/m2cgen>
[9] <https://github.com/emlearn/emlearn>
[10] <https://www.tensorflow.org/>
[11] <https://github.com/Microsoft/EdgeML/>

code from ML algorithms for resource-scarce devices, such as Arduino and Raspberry Pi. The **EdgeML** library provides a set of efficient ML algorithms designed to work off the grid on severely resource-constrained scenarios. This library allows the training, evaluation, and deployment of these algorithms onto various target devices and platforms. It contains implementations of the following algorithms: Bonsai (KUMAR; GOYAL; VARMA, 2017), a shallow and strong non-linear tree-based classifier; ProtoNN (GUPTA *et al.*, 2017), a prototype based kNN classifier; and EMI-RNN (DENNIS *et al.*, 2018), FastRNN, and FastGRNN (KUSUPATI *et al.*, 2018), techniques for training recurrent NN cells. These modified and original ML algorithms focus on improving processing time and memory consumption of the models to execute them in resource-constrained devices. **EdgeML** also supports generating fixed-point code for these ML models that can run on microcontrollers through SeeDot (GOPINATH *et al.*, 2019), an original automatic quantization tool. Besides being a relatively complete solution, a possible drawback of using this tool is the limitation of supporting ML models generated only by its original algorithms, which demands particular expertise to manipulate them but is also a unique characteristic compared to other related tools – including **EmbML**.

Cortex Microcontroller Software Interface Standard (CMSIS)[12] offers a different approach to executing NN models in microcontrollers. This library is a hardware abstraction layer that defines generic tool interfaces and enables device support for microcontrollers based on Arm Cortex processors. One of its components is the **CMSIS-NN** which is a set of efficient kernels developed to maximize the performance and minimize the memory footprint of NN models on Arm Cortex-M processors targeted for IoT devices (LAI; SUDA; CHANDRA, 2018). Note that **CMSIS-NN** does not support training the model, it consists of a collection of function implementations for layers usually presented in NNs to help dump a trained NN in a microcontroller. It is the user's responsibility to correctly combine function calls and upload the network weights in the code. Also, this library only implements fixed-point operations and allows building an NN with any of the following layers: fully connected, convolution, pooling, softmax, and others. Some disadvantages of using this library include supporting only NN models and the non-automatic process of producing a classifier code.

Fast Artificial Neural Network (FANN)[13] is a free open-source NN library, which implements multilayer artificial NN in C and supports execution in both fixed and floating point formats. Derived from this library, **FANN-on-MCU**[14] is an open-source framework for easy 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 optimized functions provided by CMSIS to improve performance on ARM Cortex-M cores. Wang *et al.* (2020) present this framework in detail and evaluate runtime and power consumption of its NNs in three different applications for

[12] <https://github.com/ARM-software/CMSIS_5>
[13] <https://github.com/libfann/fann>
[14] <https://github.com/pulp-platform/fann-on-mcu>

a wearable multi-sensor bracelet. Though this is a robust solution, the numbers of supported ML models (only NNs) and microcontrollers are very restricted.

All the tools mentioned so far are open-source. An example of a proprietary tool is the **STM32Cube.AI** which allows fast and automatic conversion of NN models into optimized code that can run on STM32 ARM Cortex-M-based microcontrollers. It offers interoperability with popular deep learning training tools, permitting to import their output models directly into the **STM32Cube.AI**. Besides being a proprietary tool, another drawback is its lack of diversity in ML models.

Furthermore, commercial tools are abundant, but beyond the scope of this work due to the costs involved. However, considering the popularity in academia, it is valid to mention the **Matrix Laboratory (MATLAB) Coder**, a tool that allows converting a MATLAB program – including ML models – to produce C and C++ codes for a variety of hardware platforms, from desktops to embedded hardware.

2.3 Related Applications

The following works report the use of ML models in resource-constrained systems. They are some examples of applications that motivate the development of the tool presented in this dissertation. The objective of showing them is to identify: successful experiences of embedding ML models in low-power systems, different application scenarios that could take advantage of using **EmbML**, characteristics of these problems, and the most commonly employed ML models.

2.3.1 Eye gesture classification

O'Bard and George (2018) propose a low-cost assistive system for eye gesture classification. Its main goal is to provide the ability to control speech devices using eye movements for patients with quadriplegia, amyotrophic lateral sclerosis, or other neurodegenerative diseases. The proposed system uses four electrodes placed around the user's eyes to collect electrooculography signals generated by eye movements. The authors evaluated kNN, SVM, and decision tree models to predict gesture signals, which were recorded and divided into looking up, down, left, and right as well as blink and idle. From this analysis, they selected the decision tree model to implement on an ATMega328p microcontroller since it produced the smallest footprint of MATLAB generated C code. In order to test on the embedded device in a real-time setup, they decided to predict only blink and idle classes. Results show that the device was able to classify blinks with 97.33% accuracy and filter out unintentional blinks with 100% accuracy.

2.3.2 Fall detection

A wearable device containing gyroscope sensors and accelerometers, proposed by Shi *et al.* (2009), aims to reduce the force of the impact caused by falls in elderly people. The system, containing an TMS3206713 DSP, preprocesses the signals gathered by the sensors, then a classifier produced with the SVM algorithm is responsible for detecting a fall. Therefore, this system must be able to identify falls before they complete in order to have sufficient time to produce an intervention. The experiments performed in this work demonstrated that the developed system successfully classified falls in real-time, enabling it to activate an airbag system to reduce the impact. It is also important to note that, when analyzing the SVM classifier with a linear kernel, the authors reported that despite the costly training process of the algorithm, this model requires little computational effort to execute.

2.3.3 Low-cost autonomous vehicle

Farooq *et al.* (2010) present the development of a low-cost autonomous vehicle for navigation and transportation of lightweight equipment. The system uses ultrasonic sensors, whose data work as input to an NN model responsible for controlling the vehicle's engines to avoid obstacles. Also, the system includes a compass sensor, GPS receiver, and wheel encoder so that it can determine and reach the desired destination. In order to reduce memory consumption and processing time of the classifier implementation in the microcontroller (an AT89C52), they used a piecewise linear approximation for the activation function of the NN neurons and converted the network weights into integer format. The performed experiments obtained a success rate of 80% for transporting computer accessories inside a university campus. In the end, the authors suggest, as a possible application, the use of this system in wheelchair navigation for individuals with physical disabilities.

2.3.4 Lung signals classification

A system for real-time monitoring and classification of lung signals, proposed by Tocchetto *et al.* (2014), aims to develop a portable device for clinical support. In this proposal, the device captures lung sounds through a microphone placed in the patient's thorax region, processes the audio signals, and sends the information to a computer server that is available to the attending doctor. Next, the authors trained an NN model with a dataset of lung sounds recorded and classified into three classes, one normal and two pathological. Then, they implemented the trained NN in a low-cost Atmel microcontroller with an Arm Cortex-M3 core, which obtained classification results similar to those obtained with MATLAB in a desktop. Finally, they reported that the classifier implementation in a low-cost microcontroller was a successful experience.

2.3.5 Ventricular tachycardia and fibrillation detection

Rúa *et al.* (2012) evaluate the performance of two classifiers when running on a Kinetis K60 microcontroller – with an Arm Cortex-M4 core – for the task of real-time detecting ventricular tachycardia and fibrillation. In this application, the authors focus on the importance of analyzing the computational cost of the classifiers since they will run on ambulatory monitoring devices. Thus, they trained SVM and NN classifiers with signals from an outpatient electrocardiogram dataset, implemented and tested it on a microcontroller using a dataset partition. Finally, the authors report that they have achieved, in experiments, results close to studies that employ off-line signal processing and conclude that the evaluated classifiers are suitable for detecting tachycardia and ventricular fibrillation in microcontrollers.

2.3.6 White cane gesture classification

Patil *et al.* (2019) present a solution for people with visual impairment who use a white cane for navigation. This solution is named GesturePod and consists of a plug-and-play device that can be attached to any white cane to perform real-time gesture-based interactions to access smartphones. According to the authors, the main technical challenge of this work is to develop a solution that satisfies all the following restrictions: low-cost, lightweight, day-long operation, and robust gesture recognition. Therefore, the proposal concentrates on an Arduino MKR1000 board (with an Arm Cortex-M0+ core) that reads and processes the data from accelerometer and gyroscope sensors. Due to battery limitations, the ML gesture recognition classifier executes in the microcontroller, so it communicates only recognized gestures to the smartphone through a Bluetooth low-energy module.

For the classification algorithm, the authors decided to use the multiclass formulation of ProtoNN (GUPTA *et al.*, 2017), provided by **EdgeML**, to identify five gesture classes and negative examples that represent the regular use of the cane. This approach was able to produce a model with $6\,KB$ in size and achieve an accuracy of 99.9% on the testing set. The authors also conducted in-lab and in-wild user experiments, recruiting people with visual impairment to evaluate the device. The results from these experiments confirmed that GesturePod is a robust solution that can promote better access to specific smartphone tasks.

2.3.7 Other applications

In addition, other studies present ML applications that demonstrate high potential of using ML models in low-power embedded systems with different objectives:

- face detection and recognition (HWANG *et al.*, 2007);
- human movement classification (KARANTONIS *et al.*, 2006);
- toxic gas detection (ALIPPI; PELOSI; ROVERI, 2006).

Since these works have a deficiency in reporting practical experiments evaluating ML models in low-power hardware, this section does not deeply explore them.

2.4 Discussion

Chart 2 exhibits a summary of the main open-source tools described in this chapter as examples related to **EmbML**. Although the description of each tool already includes some of their disadvantages compared to **EmbML**, the differences are more explicit when analyzing this table. For instance, **EmbML** is a solution that concomitantly offers:

- support to classification models trained with WEKA and scikit-learn, which are two of the most popular ML tools;
- a range of efficient classifiers that includes decision tree, SVM, MLP, and logistic regression, and explores relatively diverse learning paradigms;
- fixed-point representation (for real number operations) and sigmoid approximations (for MLP activation functions) to use as adaptations in the output classifier code and improve its performance, especially in resource-constrained hardware.

Chart 2 – Comparison between related tools.

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

In this comparison, the only possible drawback of using **EmbML** is its limitation to produce only C++ source code. However, this is intentional since C++ is a commonly used

language for programming microcontrollers. **EmbML** aims to combine in one solution the best features of the existing tools for producing microcontroller-tailored classifier code. Consequently, these observations justify the need and relevance of developing a tool such as the one proposed in this work to facilitate and popularize its use.

Chart 3 shows a comparison between the works examined in this chapter that employ classifiers to execute in embedded hardware such as sensors and wearable devices. It also includes the case study considered in this dissertation to evaluate **EmbML** classifiers in a practical application, and thoroughly investigated in Chapter 7.

Chart 3 – Comparison between related applications.

Work	Objective	Processing unit	Evaluated classifiers
In this dissertation, proposed by Batista *et al.* (2011a)	Flying insect classification	MK20DX256VLH7	MLP, SVM, logistic regression, and decision tree
O'Bard and George (2018)	Eye gesture classification	ATMega328p	kNN, SVM, and decision tree
Shi *et al.* (2009)	Fall detection	TMS3206713	SVM
Farooq *et al.* (2010)	Low-cost autonomous vehicle	AT89C52	NN
Tocchetto *et al.* (2014)	Lung signals classification	Atmel Arm Cortex-M3	NN
Rúa *et al.* (2012)	Ventricular tachycardia and fibrillation detection	Kinetis K60	SVM and NN
Patil *et al.* (2019)	White cane gesture classification	MKR1000	modified kNN

Confronting these examples makes it possible to show different successful applications that implement classifiers in unresourceful hardware, but also to establish a chronic lack of variety in classification models employed in most of these works – except the one explored in this dissertation. A point of view could argue that the authors preferred to analyze only high-efficient classifiers since microcontrollers are the main choice for their processing unit. Nevertheless, logistic regression and decision tree are also efficient classifiers – as we shall see in Chapter 6 – but not properly investigated in these works, for example. In conclusion, **EmbML** offers different alternatives for classifiers and adaptations, so developers of future low-power smart devices can take advantage of them and produce a more extensive comparative analysis to choose the most suitable option.

2.5 Final Considerations

This chapter presented an overview of some existing tools related to the one proposed in this work. It described their characteristics and specific drawbacks that make EmbML an attractive solution compared to them. We also examined some applications that relate to both the case study later explored in this dissertation and experiences of using ML models in unresourceful

hardware. The main focus was to verify how EmbML features would be beneficial to produce a comprehensive analysis of the classification models. In the next chapter, we will discuss the essential points of the ML models supported by EmbML.

CHAPTER

3

CLASSIFICATION ALGORITHMS AND MODELS

3.1 Initial Considerations

Before presenting the implementation details of EmbML, we review in this chapter the theoretical concepts of ML with a focus on classification algorithms. Then, we study each of the models supported by EmbML and describe their learning and inference processes as well as other singularities. At the final, we provide a brief overview of some popular tools in ML for running experiments, their main features, and the reasons to select WEKA and scikit-learn to support in EmbML.

3.2 Machine Learning Overview

Machine Learning (ML) consists of developing programs that automatically improve their performance from previous experiences (MITCHELL, 1997). An ML algorithm seeks to extract knowledge from examples – which can also be called instances. Each example contains a set of values for different attributes that describe its characteristics or aspects (MONARD; BARANAUSKAS, 2003). For instance, words can be attributes of a text, the frequency spectrum can represent a song, and the values from red, green and blue components of all pixels can be attributes of an image.

Typically, the inductive learning process, accomplished by reasoning about available instances, can be divided into supervised and unsupervised (MONARD; BARANAUSKAS, 2003). In supervised learning, each instance of the training set, used in the learning process, has an associated class. Such class can be continuous, in regression problems, or discrete, in classification problems. Also, since the class of each dataset example is known beforehand, it is possible to perform tests that estimate the classification performance for the model produced

by the algorithm. In the unsupervised case, however, the learning algorithm is responsible for analyzing the given instances and identifying clusters (MONARD; BARANAUSKAS, 2003), *i.e.*, groups of similar examples. For this task, it is common to apply measures that define the similarity between the examples (IRANI; PISE; PHATAK, 2016).

3.3 Classification

In classification problems, we aim to use the examples available in a training set to produce a model that can correctly predict the discrete classes of unlabeled instances, using only their attribute values. This model (or classifier) can also be understood as a function learned during the training step that is responsible for mapping elements from the input space (space of instances) into values from the output space (label space), producing as few error as possible (LUXBURG; SCHÖLKOPF, 2011). In the training process, the classification algorithm takes a set of instances $(\mathbf{x_i}, y_i)$ as input, in which $\mathbf{x_i} = (x_{i,1}, ..., x_{i,n})$ is an observed example and y_i is its corresponding class, and produces a classification model. During the test step, we evaluate whether this classifier correctly predicts the y_t classes for unseen $\mathbf{x_t}$ inputs, *i.e.*, examples that do not belong to the training set (DOMINGOS, 2012).

Classification algorithms can present considerable distinctions, in part by how they represent the candidate models (*e.g.*, decision trees, hyperplanes, and NNs) and also by how they search through the hypothesis space – the set of classifiers that it can learn (*e.g.*, optimization algorithms and evolutionary search methods) (JORDAN; MITCHELL, 2015; DOMINGOS, 2012). Therefore, different algorithms can produce classifiers with distinct biases and characteristics, such as structures, size (in terms of memory usage to store its parameters), and the number of operations required to classify an instance. These peculiarities are some of the reasons why we should explore a wide range of classification algorithms, with different learning paradigms, in order to determine the most appropriate ones for an addressed problem, depending on the objectives and limitations previously defined.

Among the variety of ML algorithms commonly employed in classification problems, the remaining of this chapter focus on describing aspects from the four different classifiers supported by the EmbML tool: Decision Tree, Multilayer Perceptron, Logistic Regression, and Support Vector Machine. The selected algorithms are those whose classifiers tend to be simpler, commonly used in low-power embedded applications, and that represent different learning paradigms. Furthermore, while some techniques such as the ensemble of classifiers (DIETTERICH, 2000), deep learning (SCHMIDHUBER, 2014), and instance-based algorithms (MARTIN, 1995) usually achieve competitive classification performance in many real-world applications, their classifiers are often computationally intensive and/or memory-consuming. For these reasons, they are not suitable to execute in a computational environment with severe memory and processing constraints.

3.3.1 Decision Tree

The decision tree is a classification algorithm that induces a set of *if-then-else* rules organized into a tree structure. Therefore, it is a process that produces a rule-based classifier and belongs to the symbolic learning paradigm.

A decision tree model contains zero or more internal nodes and one or more leaf nodes. The classification process consists of traversing the trained tree, starting from the root, and continuing until it reaches a leaf. Each leaf node assigns a class to an instance. Each internal decision node has two or more child nodes and tests the value of an instance attribute, which determines the edge – or branch – to follow to one of its children (MURTHY, 1998).

Learning a decision tree model involves two main problems: selecting the attribute to test on each internal node of the tree, and determining when a node shall be a leaf. For instance, one of the most common ways to solve the first task is by using the information gain measure that indicates the expected entropy reduction after partitioning the remaining examples for a given node using one of the attributes (MITCHELL, 1997). Also, we can consider a node as a leaf when all the remaining examples for it belong to the same class, which becomes the class assigned to this leaf. Some popular examples of decision tree induction algorithms are Iterative Dichotomiser 3 (ID3) (QUINLAN, 1986), C4.5 (QUINLAN, 1993), and Classification And Regression Tree (CART) (BREIMAN *et al.*, 1984).

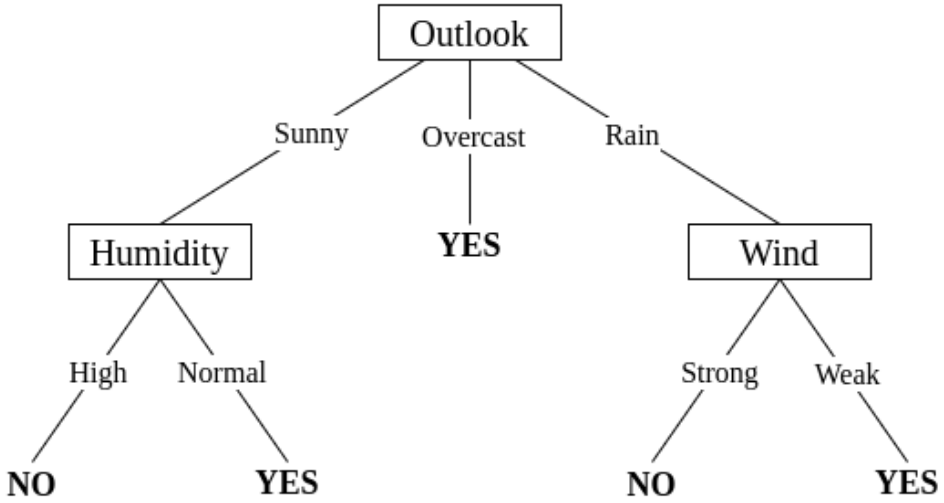
After the training process of a decision tree, it is common to occur overfitting, *i.e.*, when the produced model is very adjusted to the training set instances and does not perform well with other examples. Some of the typical solutions to this problem involve pruning the tree, turning internal nodes into leaf nodes, and stopping the training step before it creates a tree model that perfectly fits the examples from the training set (KOTSIANTIS; ZAHARAKIS; PINTELAS, 2007).

Figure 1 illustrates an example of a decision tree model built for the problem of deciding to play tennis, in which the possible classes are *YES* and *NO*, and the attributes are *outlook*, *humidity*, and *wind*. By observing the tree structure, it is possible to note, for instance, that there are three possible values for the attribute *outlook*: *sunny*, *overcast* and *rain*. Moreover, this decision tree would classify the examples $\{outlook = sunny, humidity = normal, wind = strong\}$ and $\{outlook = rain, humidity = normal, wind = strong\}$ as *YES* and *NO*, respectively.

3.3.2 Multilayer Perceptron

Multilayer Perceptron (MLP) is an NN model that represents a connectionist learning paradigm and derives from the idea of the perceptron, a linear and binary classifier proposed by Rosenblatt (1958). An MLP network consists of multiples of layers of neurons (or processing elements) that interact using a weighted connection (PAL; MITRA, 1992). There are usually three types of layers: input, hidden, and output. A common MLP design for classification problems

Figure 1 – An example of a decision tree model for playing tennis.



Source: Adapted from Mitchell (1997).

is to set the number of input layer neurons as the number of attributes of the instances, and the number of output layer neurons as the number of possible classes.

The input layer neurons usually do not perform any kind of processing, they just propagate forward the input signal that represents the value of an instance attribute. A neuron k, belonging to the hidden (or output) layer, applies an activation function f to the weighted sum v_k of the received signals y_i from the previous layer neurons. Then, it produces the output y_k presented in Equation 3.1, in which m is the number of neurons in the layer before the neuron k layer, w_{ik} is the weight associated with the connection between the neurons i and k, and θ_k is the bias associated with the neuron k.

$$y_k = f(v_k) = f\left(\left(\sum_{i=1}^{m} w_{ik} \times y_i\right) + \theta_k\right) \tag{3.1}$$

The most common activation functions to use in an MLP model are: the sigmoid, presented in Equation 3.2; the Rectified Linear Unit (ReLU), presented in Equation 3.3; the linear function, presented in Equation 3.4; and the hyperbolic tangent function, presented in Equation 3.5 (BALDI; HORNIK, 1995; MAAS; HANNUN; NG, 2013).

$$f(x) = \frac{1}{1+\exp(-x)} \tag{3.2}$$

$$f(x) = \max(0,x) \tag{3.3}$$

$$f(x) = x \tag{3.4}$$

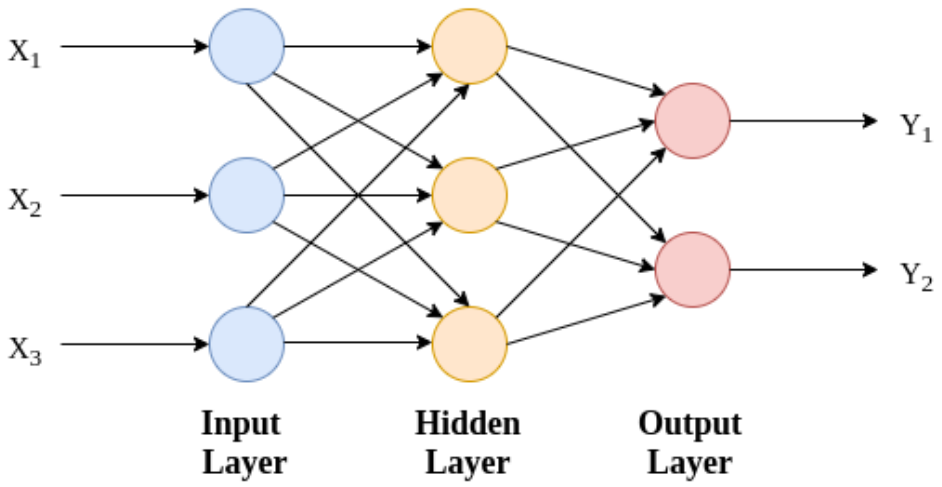
$$f(x) = \tanh(x) \tag{3.5}$$

The process of classifying an example in a trained MLP network starts with applying the attribute values as input to the input layer neurons. Then, the signals propagate forward through every layer until they reach the output layer neurons that will indicate the classification result. Since each neuron in the output layer generally represents a possible class, the predicted class is the one corresponding to the neuron that has the highest activation value at the end of this process.

The learning step of an MLP network occurs by adjusting the connection weights between neurons based on the training set knowledge so that the network produces the expected output for a given input. The backpropagation is a widely known method to train an NN model (HECHT-NIELSEN, 1992). It essentially consists of the following steps: the input signal propagates through the network until it reaches the output layer; the comparison between the output result and the expected one produces an error signal; and, finally, this error propagates in the opposite direction (backward) to update the weights for every connection in the entire network (HAYKIN *et al.*, 2009). The algorithm performs several iterations of these steps, using the examples from the training set, until the error decreases to a value below a defined threshold, when convergence occurs, or until it reaches the maximum limit of iterations. Since training an MLP model involves searching in large parameter space, it is usually a slow process (PAL; MITRA, 1992).

Figure 2 presents an MLP network for a problem with three attributes $\{X_1, X_2, X_3\}$ and two classes represented by $\{Y_1, Y_2\}$. This model also has three neurons in a single hidden layer and it is fully-connected – *i.e.*, each neuron from a given layer is connected to every neuron from the next layer.

Figure 2 – An example of an MLP network.



Source: Elaborated by the author.

3.3.3 Logistic Regression

Logistic regression is a generalized linear model used for classification problems (PREGIBON *et al.*, 1981; PENG; LEE; INGERSOLL, 2002). Considering binary classification problems, the output produced by this model is an estimation of the probability for a class ($+y$, for instance) and it is calculated using the sigmoid function defined in Equation 3.6, in which $\mathbf{w}$ is a vector with the model parameters and $\mathbf{x}$ corresponds to an example of the problem, represented as an attribute vector. The probability estimation for the complementary class of the problem ($-y$, for instance) is calculated using Equation 3.7.

$$p(+y \,|\, \mathbf{x}) = f(\mathbf{w}^T \mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{w}^T \mathbf{x})} \tag{3.6}$$

$$p(-y \,|\, \mathbf{x}) = 1 - p(+y \,|\, \mathbf{x}) \tag{3.7}$$

Therefore, the training process of a logistic regression model consists of determining values for the parameter vector $\mathbf{w}$ that produces the expected outputs. Using the maximum likelihood estimation method combined with an iterative numerical method, such as Gradient descent or Newton-Raphson, is a common way to estimate the values for $\mathbf{w}$ (BISHOP, 2006). Finally, the classification step in a logistic regression model only requires applying the input $\mathbf{x}$ in the trained model and analyzing the output produced by the sigmoid function: the label associated with the highest probability usually defines the predicted class.

In a multiclass classification problem, it is possible to use the one-against-all method, which involves training m binary classifiers – as the one described above – for a problem with m possible classes and each classifier separates one class from all the rest (POLAT; GÜNEŞ, 2009). Another approach comprises estimating the posterior probability for each class using the generalization of the logistic function, *i.e.*, the softmax function, given by the Equation 3.8, in which m is the number of possible classes, y_k represents one possible class, $\mathbf{w}$ is a matrix with parameter values, and $\mathbf{w}_i$ is the i-th column of $\mathbf{w}$ (BISHOP, 2006).

$$p(y_k \,|\, \mathbf{x}) = \frac{\exp(\mathbf{w}_k^T \mathbf{x})}{\sum_{j=1}^{m} \exp(\mathbf{w}_j^T \mathbf{x})} \tag{3.8}$$

3.3.4 Support Vector Machine

Support Vector Machine (SVM) is an ML algorithm proposed by Cortes and Vapnik (1995) to solve binary classification problems. In order to deal with multiclass problems, an SVM model can combine a set of classifiers using one-against-one or one-against-all strategies, for instance. The SVM algorithm aims to construct an optimal hyperplane that separates examples from different classes, which allows the model to have a high generalization potential (CORTES; VAPNIK, 1995). A common approach for SVM algorithms is to apply a nonlinear transformation

– called kernel function – in the input data in order to map the examples into a higher dimensional space that may facilitate the class separation. Chart 4 presents examples of popular kernel functions. In this chart, $\mathbf{u}$ and $\mathbf{v}$ are vectors from the input space, $\mathbf{u} \cdot \mathbf{v}$ represents the dot product between vectors $\mathbf{u}$ and $\mathbf{v}$, $\gamma > 0$ is a free kernel coefficient, $d > 0$ is the degree of the polynomial function, $\|\mathbf{u} - \mathbf{v}\|^2$ represents the squared Euclidean distance between vectors $\mathbf{u}$ and $\mathbf{v}$, and $r \in \mathbb{R}$ is an independent term.

Chart 4 – Examples of kernel functions.

Kernel functions	
Linear kernel	$K(\mathbf{u}, \mathbf{v}) = \mathbf{u} \cdot \mathbf{v}$
Homogeneous polynomial kernel	$K(\mathbf{u}, \mathbf{v}) = (\gamma(\mathbf{u} \cdot \mathbf{v}))^d$
Inhomogeneous polynomial kernel	$K(\mathbf{u}, \mathbf{v}) = (\gamma(\mathbf{u} \cdot \mathbf{v}) + 1)^d$
Radial Basis Function kernel	$K(\mathbf{u}, \mathbf{v}) = \exp(-\gamma\|\mathbf{u} - \mathbf{v}\|^2)$
Sigmoid kernel	$K(\mathbf{u}, \mathbf{v}) = \tanh(\gamma(\mathbf{u} \cdot \mathbf{v}) + r)$

Learning an SVM classifier means solving a quadratic programming problem that determines the hyperplane with a maximum margin of separation between examples from different classes, *i.e.*, this hyperplane must be as far as possible from the support vectors – examples that limit the margins. Equation 3.9 shows the formulation provided by Vapnik (1999) for this problem, in which m is the number of examples in the training set, x_i is the i-th example in the training set, y_i is the label of x_i, α_i is the Lagrange multiplier for x_i, and K is the kernel function. The examples x_i for which $\alpha_i > 0$ are the support vectors of the model (SCHÖLKOPF *et al.*, 2002).

$$\begin{aligned} \max_{\alpha \in \mathbb{R}^m} \quad & \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \, \alpha_j \, y_i \, y_j \, K(x_i, x_j) \\ \text{subject to} \quad & \sum_{i=1}^{m} \alpha_i \, y_i = 0, \\ & \alpha_i \geq 0, \text{ for all } i = 1, \ldots, m \end{aligned} \tag{3.9}$$

Cortes and Vapnik (1995) present two ways to formulate this optimization problem. The hard-margin approach assumes that the problem is linearly separable, and it is possible to separate the training examples from each class without error. The soft-margin approach tolerates some noisy examples or class overlap. In this last case, it is not possible to perfectly separate the training data using a hyperplane, thus the method aims to find the hyperplane that performs the minimal amount of error. Figure 3 and Figure 4 respectively illustrate examples of hard-margin and soft-margin approaches, in which the dashed lines represent the margins, the separating

hyperplanes are the lines between the margins, and the support vectors for each class are the highlighted examples – over the margins.

Figure 3 – An example of hyperplane with hard-margin for a linearly separable problem.

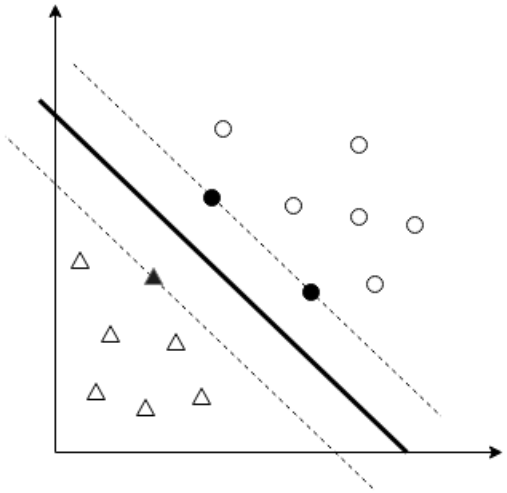
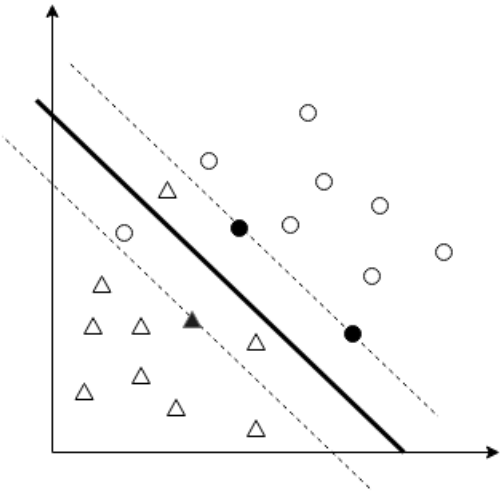


Figure 4 – An example of hyperplane with soft-margin for a problem with noisy data.



Source: Elaborated by the author.

The decision function presented in Equation 3.10 (VAPNIK, 1999) determines the predicted class – positive or negative – of an input $\mathbf{x}$, which corresponds to its position in relation to the separating hyperplane. In this equation, n is the number of support vector of the model, y_i is the class associated with the support vector x_i, α_i is the Lagrange multiplier for x_i, K is the kernel function, and b is the hyperplane threshold that can be calculated, for instance, by averaging the possible values from Equation 3.11 over all support vectors x_j (SCHÖLKOPF *et al.*, 2002).

$$f(\mathbf{x}) = sign\left(\left(\sum_{i=1}^{n} y_i\,\alpha_i\,K(\mathbf{x_i},\,\mathbf{x})\right) - b\right) \tag{3.10}$$

$$b = y_j - \left(\sum_{i=1}^{n} y_i\,\alpha_i\,K(\mathbf{x_i},\,\mathbf{x_j})\right) \tag{3.11}$$

3.4 Machine Learning Tools

Software tools, libraries, and frameworks are quite popular and important in the ML community for many reasons. For instance, they provide implementations for several traditional ML algorithms and make it easier to run experiments. In addition, they can allow some features to the users that help in most steps of the ML pipeline, such as:

- saving the trained model;

- using data preprocessing techniques;
- visualizing classes and attributes distributions for the dataset;
- applying and analyzing different attribute selection techniques;
- employing performance evaluation techniques such as k-fold cross-validation and holdout;
- choosing the set of hyperparameters for the algorithms;
- making use of small datasets provided by the tools;
- and treating different types of ML problems, such as classification, regression, and clustering.

Particularly due to the usability and popularity of these tools, the idea of using classifiers trained with them represents a way to simplify their use in microcontroller-based embedded systems.

Among all the existing tools, it is important to highlight some of the most popular that are capable of performing most of the features mentioned: WEKA (HALL *et al.*, 2009), scikit-learn (PEDREGOSA *et al.*, 2011), RapidMiner (HOFMANN; KLINKENBERG, 2013), KNIME (BERTHOLD *et al.*, 2007), and MOA (BIFET *et al.*, 2010). Although all of these tools have enough interesting features to explore in this work, this study focus on developing a tool that supports only models generated with WEKA and scikit-learn. The main factors that justify these choices are: the time limit for the development of this work; the availability of access to the source code of these tools; and the fact that both provide easy-to-use implementations of the main classification algorithms.

Scikit-learn provides state-of-the-art implementations of several popular ML algorithms, including those intended for unsupervised and supervised problems. It has a friendly interface using Python – a high-level and general-purpose programming language – which enables easy access to non-experts. According to Pedregosa *et al.* (2011), the main benefits of scikit-learn comparing to other ML toolboxes in Python – *e.g.*, MDP (ZITO *et al.*, 2009), PyBrain (SCHAUL *et al.*, 2010), and PyMVPA (HANKE *et al.*, 2009) – are: the distribution under the simplified Berkeley Software Distribution (BSD) license; the incorporation of compiled code for efficiency; the dependency of only NumPy (OLIPHANT, 2006) and SciPy (Virtanen *et al.*, 2020) modules which facilitates its distribution; and the focus on imperative programming.

In the case of WEKA, it consists of an open-source workbench written in Java with strong popularity in academia and business (HALL *et al.*, 2009). It intends to be flexible for examining different methods and facilitate the experimental process in data mining. To achieve these objectives, WEKA incorporates, in a uniform interface, a diverse set of state-of-the-art ML algorithms and solutions to several data mining tasks, including classification,

regression, clustering, association rule mining, and attribute selection (WITTEN *et al.*, 2016). Data exploratory is also supported using preprocessing and visualization mechanisms. Moreover, it provides easy access to all its features through a Graphical User Interface (GUI) that allows using it without writing any line of code.

3.5 Final Considerations

In this chapter, we showed an overview of the ML theory for classification problems. We then focused on studying the details for each of the classification models that EmbML supports. The reasons to choose these models consider their efficiency and the restrictions imposed for executing them in unresourceful hardware. At last, we introduced some ML tools and explained the features of WEKA and scikit-learn that made them attractive to this work. Therefore, this chapter constitutes an important step of this work since it allows establishing the foundations to develop the examination of each WEKA and scikit-learn models in Chapter 5.

CHAPTER

4

EMBML – EMBEDDED MACHINE LEARNING

4.1 Initial Considerations

This chapter, based on Silva, Souza and Batista (2019a) and Silva, Souza and Batista (2019b), presents EmbML[1] – a tool written in Python to automatically convert off-board-trained models into C++ source code files that can be compiled and executed in microcontrollers. We start by describing each step involving the pipeline process of using EmbML to produce a classifier code. After that, we explain the input format and the classes of WEKA and scikit-learn models supported by EmbML. Finally, we focus on explaining the implemented modifications that enable the classifiers to reduce processing time and memory consumption for running in resource-constrained hardware.

4.2 Pipeline Overview

The main goal of using EmbML is to produce classifier source codes for executing in low-power microcontrollers. This process starts with creating a model using the WEKA package or the scikit-learn library from a dataset at hand. As presented in the previous chapter, these are popular and open-source tools that provide a wide range of classification algorithms and simplify training and evaluating an ML model. After training the model using these tools in a desktop or server computer, the user needs to save it as an object serialized file, which is capable of saving all the object content – including the classifier parameters and data structures – for future use.

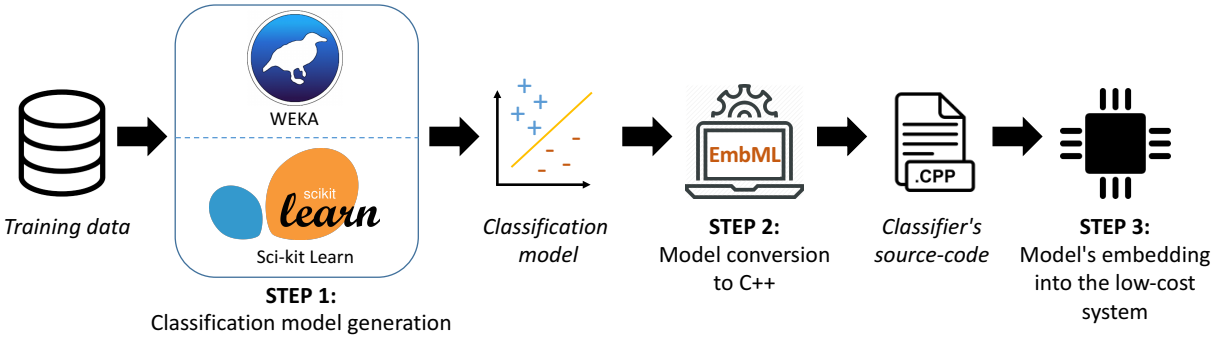
EmbML receives such a serialized file as input and uses specific libraries that implement methods to deserialize the file. The deserialization process allows the tool to recover the classifier data and extract the relevant information, such as the model parameters. Finally, EmbML fills a

[1] Available online at: <https://github.com/lucastsutsui/EmbML> and <https://pypi.org/project/embml/>

template file – for a specific classifier – using the data retrieved in the previous step and generates a C++ programming language file containing the model parameters, their initialization values, and implementations of functions that use them for classifying an instance. This output file only contains the functions related to the classification step, since it will later become part of a more complex program for running on an embedded platform and may include other functionalities according to the application such as feature extractor and preprocessing, and further actions on the classifier output.

Figure 5 illustrates the operation workflow. The user employs 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 fixed-point or floating-point representations for processing real number operations. As we will see in Chapter 6, such choice may impact both the accuracy and efficiency of the classifier. Therefore, after evaluating the classifier in the desired hardware, the user may return to Step 2 if they want to assess other numerical representations or even to Step 1 if the classifier does not meet the time or memory requirements of the application. Thus, the workflow is likely to have these feedback loops not illustrated in the figure. In Step 3, it is possible to compile the code and deploy it on the microcontroller, for instance, using a combination of cross compilers – *e.g.*, *avr-gcc*[2] and *gcc-arm-none-eabi*[3] – and firmware upload protocols – *e.g.*, *avrdude*[4] and *jlink*[5].

Figure 5 – Workflow for generating classifier source code using EmbML.



Source: Silva, Souza and Batista (2019a).

[2] <https://gcc.gnu.org/wiki/avr-gcc>
[3] <https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm>
[4] <https://www.nongnu.org/avrdude/>
[5] <https://www.segger.com/products/debug-probes/j-link/>

4.3 Serialization and Model Recovery

A significant piece of the described pipeline lies in understanding the format used to provide off-board-trained classifiers as input to EmbML. This step is achievable by using the method called serialization – *i.e.*, the process of converting an object state into a format so that the object can be stored in a file. Therefore, after training a classification model with WEKA or scikit-learn, it is possible to save it in a file and later recover it from this same file through serialization and deserialization processes, respectively.

The WEKA classifier object can be serialized to a file using the *ObjectOutputStream* and *FileOutputStream* classes available in Java. This object can be retrieved from this file through the *ObjectInputStream* and *FileInputStream* classes also available in Java. When working with the WEKA GUI, it provides the options to save a trained model and load a serialized model that use these classes in the underlying code. Otherwise, if the user writes a Java code to train a model, using the WEKA Java Archive (JAR) package, they have to employ these classes manually.

However, the classes that implement the WEKA classifiers have most of their main variables declared as private, which prevents obtaining their values from an external program. The alternative explored in this work to access the contents of WEKA classifier objects is by using the *javaobj*[6] 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. Since Python has no private attributes, we have access to all variables of the trained classifier and can select the ones that are relevant to the classification step.

When dealing with scikit-learn models, it is possible to use the *pickle*[7] module that allows serializing and deserializing a Python object. The approach accepted by EmbML for saving a classification model consists of applying the function *dump* to serialize the model into a file. After that, EmbML can recover the classifier object from this file using the *load* method and access the object content without restriction.

After retrieving the variables and data structures of the trained model, EmbML produces a C++ file that contains the initialized variables and structures, including function implementations for the classification step – based on WEKA and scikit-learn source codes, with adjustments to improve their performance on low-power microcontrollers such as replacing recursive functions by their iterative implementations.

[6] <https://pypi.org/project/javaobj-py3/>
[7] <https://docs.python.org/3/library/pickle.html>

4.4 Algorithms and Classes

The algorithms supported by EmbML are those suitable to execute in unresourceful hardware. Therefore, they are simple models that generally require little processing time and produce a small memory footprint. Thus, EmbML does not support some popular approaches such as:

- kNN, due to its requirement of storing the training set in memory as well as its necessity of searching such a set in classification time;
- ensemble methods since they require storing multiple classifiers in memory;
- and deep learning algorithms, considering that they often create large NN models with several number of layers.

As discussed in the previous chapter, the EmbML supports representative models of different learning paradigms: MLP networks, logistic regression, decision tree, and SVM – with linear, polynomial, and Radial Basis Fuction (RBF) kernels. For WEKA, EmbML accepts the models from the following classes:

- ***J48*** generates decision tree classifier;
- ***Logistic*** trains logistic regression classifiers;
- ***MultilayerPerceptron*** produces MLP classifiers;
- and ***SMO*** creates SVM classifiers – with linear, polynomial, and RBF kernels.

As for the scikit-learn models, it allows the models from these 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.

4.5 General Modifications

EmbML implements some modifications in all produced source code as a way to improve performance when running it in low-power microcontrollers. Yet, in all of them, EmbML does not interfere with the training process, it only provides adjustments that affect the execution of the classifier source code. The idea to develop one of these available modifications is that the classifier parameters do not change during execution in a microcontroller, according to our pipeline. Consequently, it is possible to store the model parameters – *e.g.*, the weights of an NN network – in the microcontroller's program memory, once it is usually larger than its data memory. To accomplish that, EmbML generates classifier source codes that declare these data in *const* variables. This keyword expresses to the compiler that these data are read-only, and it is reasonable to store them in the microcontroller's flash memory.

Another modification lies in the fact that most microcontrollers have a limited processing capacity such that they even lack a Floating-Point Unit (FPU). This unit consists of a hardware system specifically designed to efficiently perform floating-point computations such as addition, subtraction, division, and multiplication. When it is missing, executing operations with real numbers becomes a challenging task. As described in Gopinath *et al.* (2019), there are two alternatives to tackle the absence of floating-point support: emulating the floating-point operations via software or converting real numbers to fixed-point representation. Software emulation usually is processing-expensive and results in a loss of efficiency. On the other hand, the second approach may reduce the range of represented values and also cause a loss of precision.

EmbML enables producing classifier source codes that use both floating-point and fixed-point formats to store real number data. For the first option, the generated code can directly proceed to the microcontroller's compilers since most of them – *e.g.*, the *gcc-arm-none-eabi* compiler for 32-bit Arm Cortex processors – already provide configuration options that allow emulating floating-point operations or using FPU instructions. For fixed-point representation, the scenario is quite different since the compilers of different microcontrollers do not offer a universal solution. Therefore, developers need to seek existing open-source libraries that implement fixed-point operations or create their own.

4.5.1 Fixed-point Representation

As EmbML supports that its classifiers manipulate fixed-point representations instead of floating-point, we implemented a library of fixed-point operations based on existing ones: *fixedptc*[8], *libfixmath*[9] and *AVRfix*[10]. It includes the basic arithmetic operations – addition, subtraction, multiplication, and division – as well as other functions required by some classifiers – such as exponential, power, and square root functions. This library supports storing real numbers

[8] <https://sourceforge.net/projects/fixedptc/>
[9] <https://code.google.com/archive/p/libfixmath/>
[10] <https://sourceforge.net/projects/avrfix/>

in integer variables with 32, 16, or 8 bits and implements the *Qn.m* format in which n is the number of bits in the integer part, and m is the number of bits in the fractional part (VLĂDUŢIU, 2012). For instance, let $(x_{n-1},...,x_0,...,x_{-m+1},x_{-m})$ be the fixed-point binary representation of an unsigned number X and x_{-m} is the Least Significant Bit (LSB), then its value is given by:

$$X = \sum_{i=-m}^{n-1} x_i \, 2^i \tag{4.1}$$

Using the two's complement format, a signed number Y with fixed-point binary representation $(y_{n-1},...,y_0,...,y_{-m+1},y_{-m})$ has the value obtained by:

$$Y = -y_{n-1} \, 2^{n-1} + \sum_{i=-m}^{n-2} y_i \, 2^i \tag{4.2}$$

Now, consider the numbers $A \in \mathbb{R}$ and $B \in \mathbb{R}$. Also, let $FXP(X,n,m)$ be the function, shown in Equation 4.3, that transforms the number $X \in \mathbb{R}$ to its *Qn.m* fixed-point representation and $round(X)$ maps a real number X to the closest integer value. Thus, Chart 5 presents how to perform the basic arithmetic operations in the fixed-point format[11].

$$FXP(X,n,m) = round(X \times 2^m) \mod 2^{n+m} \tag{4.3}$$

Chart 5 – Arithmetic operations in *Qn.m* fixed-point representation.

Operation	**Formula**
Addition	$FXP(A+B,n,m) = FXP(A,n,m) + FXP(B,n,m)$
Subtraction	$FXP(A-B,n,m) = FXP(A,n,m) - FXP(B,n,m)$
Multiplication	$FXP(A \times B,n,m) = FXP(A,n,m) \times FXP(B,n,m) \,/\, FXP(1,n,m)$
Division	$FXP(A \,/\, B,n,m) = FXP(A,n,m) \times FXP(1,n,m) \,/\, FXP(B,n,m)$

In order to calculate the exponential function with base e, used in MLP and logistic regression classifiers, it is easier to first implement its version with base 2. To improve efficiency, we implemented this function using the recurrent formula from Equation 4.4. This method reduces (or increases) the exponent value k until it belongs to the interval $[0,1]$ in which it is possible to apply a two-degree polynomial approximation of the function to obtain the corresponding value. We used an implementation[12] of the Nonlinear Least-Squares (NLLS) Marquardt-Levenberg algorithm (LEVENBERG, 1944; MARQUARDT, 1963) to find the values

[11] From now on, consider that, for numbers in a fixed-point format, the operator / represents the integer division.

[12] Available in gnuplot <http://www.gnuplot.info/>

for the coefficients: $c_2 = 0.342656$, $c_1 = 0.649427$, and $c_0 = 1.00376$. Finally, Equation 4.5 demonstrates how we obtain the exponential function with base e.

$$FXP(2^k,n,m) = \begin{cases} FXP(2^{k-1},n,m) \times FXP(2,n,m) \,/\, FXP(1,n,m) & k > 1 \\ FXP(2^{k+1},n,m) \times FXP(1,n,m) \,/\, FXP(2,n,m) & k < 0 \\ FXP(c_2k^2 + c_1k + c_0,n,m) & 0 \leq k \leq 1 \end{cases} \tag{4.4}$$

$$FXP(e^k,n,m) = FXP(2^{k \times log_2(e)},n,m) \tag{4.5}$$

Since EmbML supports SVM classifiers with the polynomial kernel, it also has to include an implementation of the power function for the fixed-point format. In order to compute this function, we use the idea presented in Equation 4.6 that consists of splitting the exponent number into two parts: integer and decimal. Equation 4.7 shows how to calculate the power function with an integer exponent applying the exponentiation by squaring method. For the other part, with a decimal exponent, we compute it using the recurrent method presented in Equation 4.8. This approach requires an implementation of the square root function, which is possible to conceive using the recurrent formula from Equation 4.9. In this method, the value of x is divided (or multiplied) by 4, until it belongs to the interval $[1,4]$. Then, we use a two-degree polynomial approximation to calculate the resulting value of the function in this interval. We applied the same NLLS method as before to obtain the coefficient values: $d_2 = -0.0352734$, $d_1 = 0.502293$, and $d_0 = 0.546737$.

$$FXP(x^y,n,m) = FXP(x^{\lfloor y \rfloor},n,m) \times FXP(x^{y-\lfloor y \rfloor},n,m) \,/\, FXP(1,n,m) \tag{4.6}$$

$$FXP(x^i,n,m) = \begin{cases} \dfrac{FXP(x^{\frac{i-1}{2}},n,m) \times FXP(x^{\frac{i-1}{2}},n,m) \times FXP(x,n,m)}{FXP(1,n,m) \times FXP(1,n,m)} & i \text{ is odd} \\ \dfrac{FXP(x^{\frac{i}{2}},n,m) \times FXP(x^{\frac{i}{2}},n,m)}{FXP(1,n,m)} & i \text{ is even} \\ FXP(1,n,m) & i = 0 \end{cases} \tag{4.7}$$

$$FXP(x^j,n,m) = \begin{cases} FXP(\sqrt{x^{2j}},n,m) & 0 < j < 1 \\ FXP(x,n,m) \times FXP(\sqrt{x^{2(j-1)}},n,m) \,/\, FXP(1,n,m) & j \geq 1 \\ FXP(1,n,m) & j = 0 \end{cases} \tag{4.8}$$

$$FXP(\sqrt{x},n,m) = \begin{cases} FXP(\sqrt{4x},n,m) \times FXP(1,n,m) \,/\, FXP(2,n,m) & x < 1 \\ FXP(\sqrt{x/4},n,m) \times FXP(2,n,m) \,/\, FXP(1,n,m) & x > 4 \\ FXP(d_2x^2 + d_1x + d_0,n,m) & 1 \leq x \leq 4 \end{cases} \tag{4.9}$$

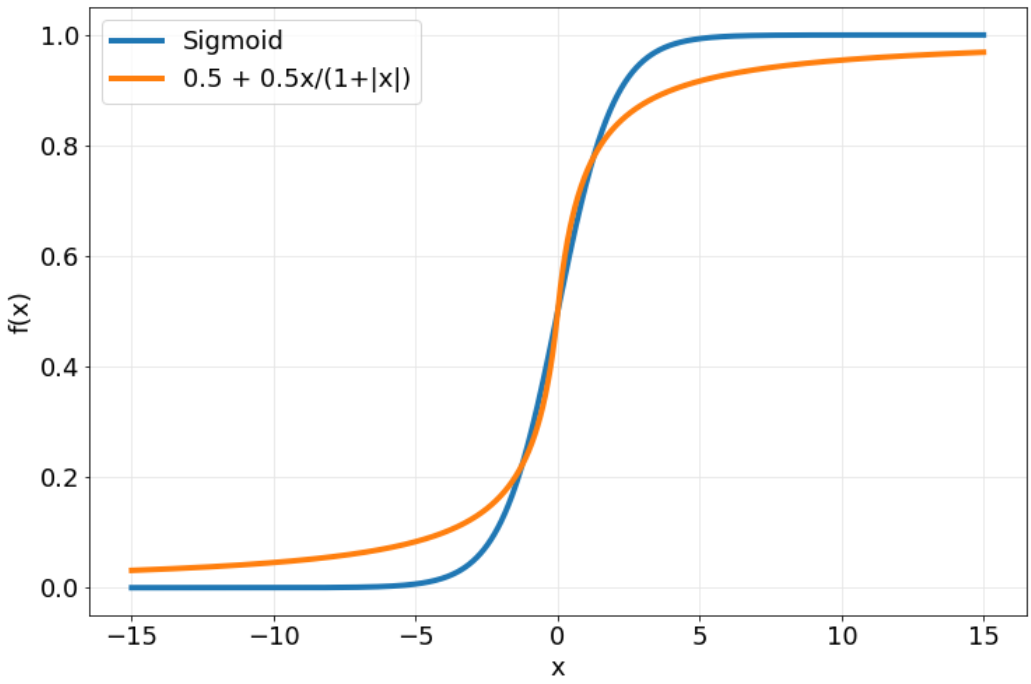
4.6 Sigmoid Function Approximations

Another modification, supported by EmbML but only for MLP classifiers, involves providing different approximations for the sigmoid function since it requires the expensive processing of an exponential function. The solution lies in examinating functions that have similar behavior but perform simpler operations.

One of the alternatives consists of using the function given by Equation 4.10 that contains only basic arithmetic operations. Figure 6 plots this function alongside with the sigmoid. The analysis of these curves intuitively reveals that they have a strong correspondence which indeed is accurate since the maximum difference between them is approximately 0.0822893 and occurs at the points $x \approx 3.77422$ and $x \approx -3.77422$.

$$f(x) = 0.5 + \frac{0.5x}{1+|x|} \tag{4.10}$$

Figure 6 – Sigmoid function and its approximation.



Source: Elaborated by the author.

4.6.1 Piecewise Linear Approximation

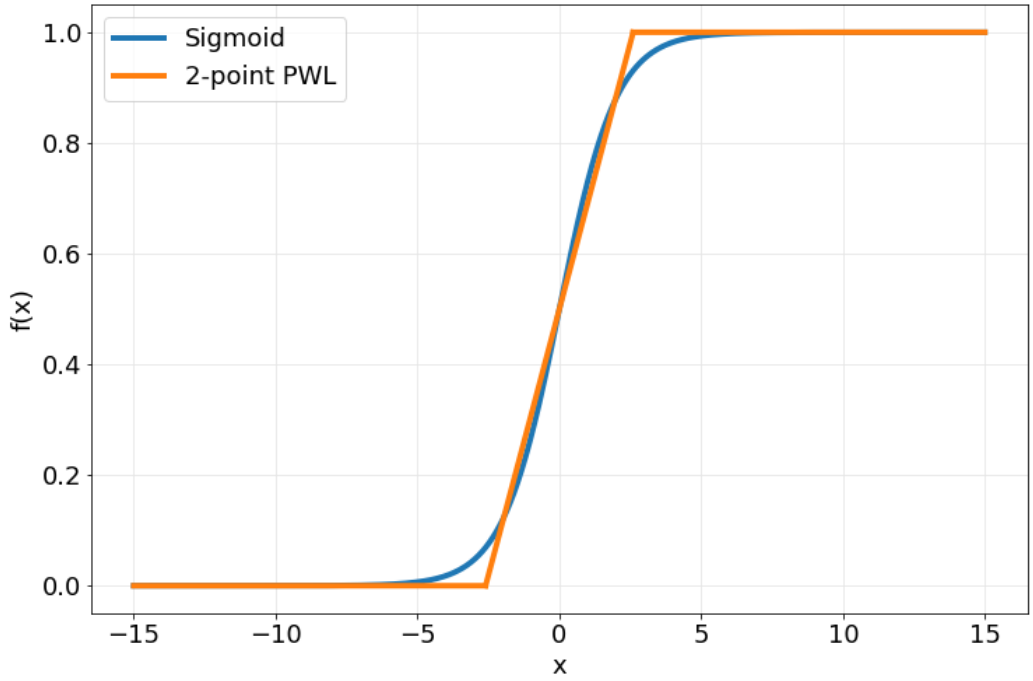
The other approach implemented in EmbML to substitute the sigmoid function is the Piecewise Linear (PWL) approximation. This method creates a series of linear segments to approximate a nonlinear function. As described by Bradley, Hax and Magnanti (1977), we obtain a PWL approximation by linearly connecting a set of selected points. The resulting set

of segments can replace the original curve and produce better results as much as we add more points to it. Since this technique reduces the sigmoid function to a group of linear functions, it can consequently improve the effort necessary to compute a value.

In the case of the PWL method, EmbML allows employing it with two or four points to replace the sigmoid function. Equation 4.11 presents the 2-point PWL approximation. In this option, we used the previously presented NLLS algorithm to determine the interval points and the function coefficients. Figure 7 compares the graphs of this approximation and the sigmoid functions and reveals that it produces a simple but relatively precise replica of the original pattern, even with just two points. The highest difference between them has a value close to 0.0690992 at the points $x \approx -2.60061$ and $x \approx 2.60061$.

$$f(x) = \begin{cases} 0 & x \leq -2.60061 \\ 0.19226x + 0.5 & -2.60061 < x \leq 2.60061 \\ 1 & x > 2.60061 \end{cases} \tag{4.11}$$

Figure 7 – Sigmoid function and a 2-point PWL approximation.



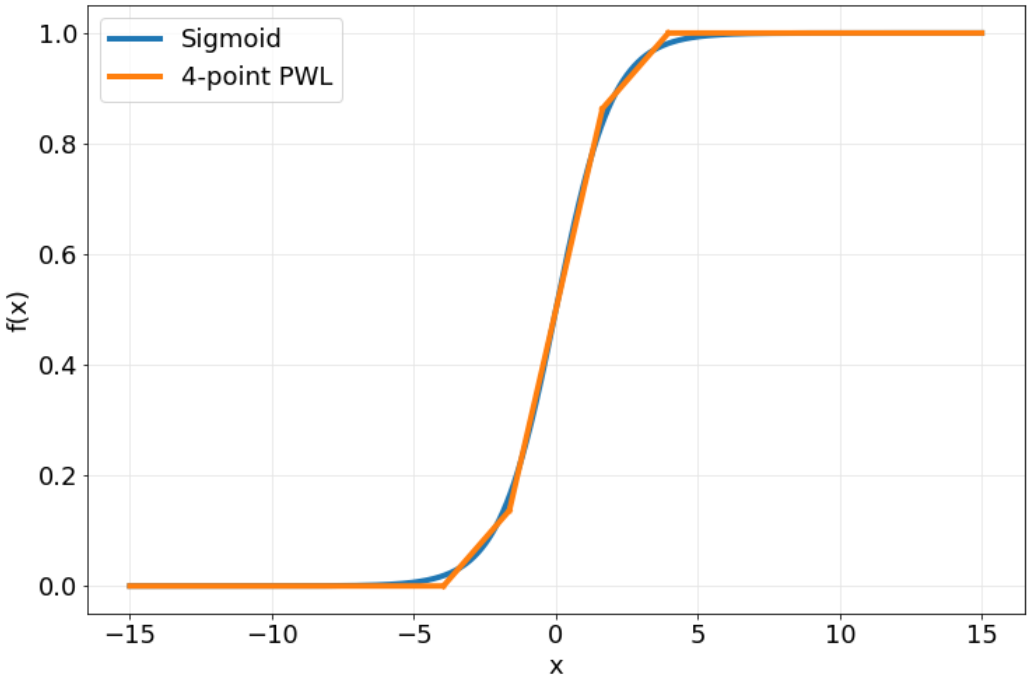
Source: Elaborated by the author.

For the 4-point PWL version of the sigmoid function, Equation 4.12 shows how EmbML implements this approximation. Again, we applied the NLLS method to optimize the values for the interval edges and the function coefficients. Figure 8 presents the graphs of the 4-point PWL approximation and the sigmoid function. Comparing with the other options, it seems to be the most well-fitted version. Also, to support this idea, the maximum difference between it and the

sigmoid achieves the lowest value among the examined approaches. It is around 0.0260915 and locates at the points $x \approx -1.63796$ and $x \approx 1.63796$.

$$f(x) = \begin{cases} 0 & x \leq -3.96049 \\ 0.05884x + 0.23303 & -3.96049 < x \leq -1.63796 \\ 0.22183x + 0.5 & -1.63796 < x \leq 1.63796 \\ 0.05884x + 0.76697 & 1.63796 < x \leq 3.96049 \\ 1 & x > 3.96049 \end{cases} \tag{4.12}$$

Figure 8 – Sigmoid function and a 4-point PWL approximation.



Source: Elaborated by the author.

4.7 If-Then-Else Statements For Decision Trees

Both WEKA and scikit-learn source codes implement the classification steps of their decision tree models using iterative or recursive methods to traverse the tree. When EmbML processes a decision tree classifier, its default option is to produce an output code that contains the iterative version of tree traversal (recursive methods are converted to their corresponding iterative form). For instance, an iterative decision tree model performs operations similar to those illustrated in the generic example of Algorithm 1.

In addition, EmbML provides an alternative representation for this process, which involves converting the binary tree structure into nested if-then-else statements. This approach

Algorithm 1 – Iterative version of a decision tree classification algorithm.

```
1:  procedure ITERATIVE_DECISION_TREE
2:      currentNode ← root
3:      while currentNode ≠ leaf do                    ▷ Tests the end of loop
4:          if conditions[i] then
5:              i ← rightChild[i]                      ▷ Proceeds to the left child
6:          else
7:              i ← leftChild[i]                       ▷ Proceeds to the right child
8:          end if
9:      end while
10:     ...                                            ▷ Processes the leaf node
11: end procedure
```

intends to optimize the classifier execution time at the expense of increasing its code size. As a result, it shall achieve better time performance since it eliminates the loop overhead – *e.g.*, instructions to increment the loop counter and test the end of the loop – of the iterative method. But, it can consume more memory resources given that its machine code will possibly contain more instructions – to reproduce the multiple comparisons from the if-then-else statements. Algorithm 2 shows the steps for a generic decision tree classifier represented as nested if-then-else statements.

Algorithm 2 – If-then-else statements for a decision tree classification algorithm.

```
1:  procedure IF_THEN_ELSE_DECISION_TREE
2:      if condition_0 then                            ▷ Right child of node 0 (root)
3:          if condition_1 then                        ▷ Right child of node 1
4:              ...          ▷ Performs other comparisons and processes the leaf nodes
5:          else                                       ▷ Left child of node 1
6:              ...          ▷ Performs other comparisons and processes the leaf nodes
7:          end if
8:      else                                           ▷ Left child of node 0 (root)
9:          ...              ▷ Performs other comparisons and processes the leaf nodes
10:     end if
11: end procedure
```

4.8 Final Considerations

This chapter presented the details of the EmbML implementation. We described the pipeline to employ this tool in the process of generating a C++ classifier source code to run in low-power microcontrollers. Then, we examined some aspects of this pipeline: the format used to accept off-board-trained classifiers as input to EmbML, and which classes of the training tools EmbML supports. Finally, we discussed the options that enable EmbML to modify classifier codes to optimize their performance in unresourceful hardware. It covered the analysis of how EmbML incorporates the fixed-point format to process real numbers, approximation functions

to substitute the sigmoid in MLP models, and different representations of decision tree models. Chart 6 summarizes the available modifications for each classifier. In the next chapter, we focus on reviewing the implementation of each supported classification model.

Chart 6 – Code modifications supported by EmbML for each classifier class.

Classifier classes	**Usage of *const* variables**	**Fixed-point representation**	**Sigmoid approximations**	**If-then-else statements**
J48 WEKA	✓	✓		✓
Logistic WEKA	✓	✓		
MultilayerPerceptron WEKA	✓	✓	✓	
SMO WEKA	✓	✓		
DecisionTreeClassifier scikit-learn	✓	✓		✓
LinearSVC scikit-learn	✓	✓		
LogisticRegression scikit-learn	✓	✓		
MLPClassifier scikit-learn	✓	✓	✓	
SVC scikit-learn	✓	✓		

CHAPTER

5

WEKA AND SCIKIT-LEARN MODELS

5.1 Initial Considerations

In this chapter, we perform a more extensive examination of the WEKA and scikit-learn classes that implement the classification models supported by EmbML. For all models, we focus on describing their working process and the specific features – variables and algorithms – from them that are necessary for classifying an instance, according to their source code available in WEKA or scikit-learn. We also provide an analysis of the computational complexity of these models, concentrating on the time and memory resources needed only for the classification process. This evaluation considers the worst-case scenario to motivate the discussion of their practical performance.

5.2 WEKA classes

The next subsections address the WEKA classes that implement the ML models provided by EmbML.

5.2.1 J48

The ***J48*** class from WEKA implements the C4.5 decision tree learning algorithm. Some hyperparameters of this class consist of deciding for a pruned or unpruned tree (the default choice is for a pruned tree), setting the confidence threshold for pruning (the default value is 0.25), and the minimum number of instances per leaf (the default value is 2).

In the WEKA original implementation of the ***J48*** decision tree, each leaf node stores its probabilities for all classes, and the model computes the probabilities for each class given an input instance. In the EmbML output code for a ***J48*** model, each leaf node saves only the class with the highest probability, and the classification function only has to return the predicted class,

which simplifies the process and reduces the amount of data to store. Therefore, the EmbML output classifier for a ***J48*** input model incorporates the following main variables:

- ***LEN_TREE*** is an integer value that represents the total number of nodes in the decision tree.

- ***ATT_OFFSET*** is an integer value equals to the number of attributes in the model. This variable indicates whether a node i is internal ($m\_attIndex[i] < ATT\_OFFSET$) or leaf ($m\_attIndex[i] \geq ATT\_OFFSET$).

- ***m_attIndex*** is a one-dimensional integer array with $O(LEN\_TREE)$ space complexity. This variable stores the attribute indices tested in each internal node or the corresponding class of a leaf. For instance, the internal node i checks the attribute value at index $m\_attIndex[i]$. For leaf nodes, $m\_attIndex[i]$ has a value greater or equal to ATT_OFFSET, and $(m\_attIndex[i] - ATT\_OFFSET)$ is its corresponding class number.

- ***m_splitPoint*** is a one-dimensional array with real numbers and $O(LEN\_TREE)$ space complexity. It saves the split point of each internal node. When executing a tree traversal during classification, this value decides which child – left or right – to proceed.

- ***tree*** is a two-dimensional integer array with $O(LEN\_TREE)$ space complexity. This array stores the tree structure such that the values $tree[i][0]$ and $tree[i][1]$ respectively specify the left and right children of node i.

Algorithm 3 shows how to use these data structures in the classification step implemented by EmbML for ***J48*** decision trees. The process consists of traversing the binary tree, starting at the root node. At each iteration of the loop, it checks if the current node is a leaf and proceeds to the child nodes according to the attribute values. When it reaches a leaf node, it returns a value that corresponds to the predicted class of the input instance. Thus, the time complexity of this algorithm is $O(LEN\_TREE)$[1] since the maximum path from the root to a leaf can be in this order of complexity.

Algorithm 4 presents the recursive method that EmbML uses to produce if-then-else statements, in string format, for a ***J48*** WEKA classifier. In this procedure, the method $Create\_Statement$ creates a string statement using its argument. Given that the classifier produced by this approach still executes the same amount of operations of the previous method, it has the same time complexity.

[1] This complexity is also proportional to the height of the tree.

Algorithm 3 – Classification algorithm for the ***J48*** WEKA model.

```
1:  procedure J48_CLASSIFY(attributes)
2:      i ← root
3:      while m_attIndex[i] < ATT_OFFSET do                ▷ Ends when it finds a leaf
4:          if attributes[m_attIndex[i]] ≤ m_splitPoint[i] then
5:              i ← tree[i][0]                             ▷ Proceeds to the left child
6:          else
7:              i ← tree[i][1]                             ▷ Proceeds to the right child
8:          end if
9:      end while
10:     return (m_attIndex[i] − ATT_OFFSET)                ▷ Returns the class number
11: end procedure
```

Algorithm 4 – Converting a ***J48*** WEKA model into if-then-else statements.

```
1: procedure GENERATE_CONDITIONALS(node)
2:     if m_attIndex[node] ≥ ATT_OFFSET then      ▷ leaf node: creates return statement
3:         return Create_Statement("return"
                   +string(m_attIndex[node] − ATT_OFFSET))
4:     else                                       ▷ internal node: creates if-then-else statement
5:         return Create_Statement("if(attributes[" + string(m_attIndex[node])
                   +"] ≤ m_splitPoint[" + string(m_attIndex[node]) + "])")
                   + Generate_Conditionals(tree[node][0])  ▷ recursion to left child
                   + Create_Statement("else")
                   + Generate_Conditionals(tree[node][1])▷ recursion to right child
6:     end if
7: end procedure
```

5.2.2 Logistic

To create multinomial logistic regression models in WEKA, we use the ***Logistic*** class. Before training, this class allows the user to set the ridge in the log-likelihood (the default value is 10^{-8}) and the maximum number of iterations (by default, it waits until convergence) of the model. EmbML uses the following parameters to represent a classifier trained with the ***Logistic*** class:

- ***NUM_PREDICTORS*** is an integer variable that stores the number of attributes used in the classification process.
- ***NUM_CLASSES*** is an integer value that describes the number of class labels.
- ***m_selectedAttributes*** is a one-dimensional integer array with space complexity in the order of $O(NUM\_PREDICTORS)$. It saves the indices of attributes selected for the classification.
- ***m_Par*** is a two-dimensional array with real numbers and memory complexity in the order of $O(NUM\_CLASSES \times NUM\_PREDICTORS)$. This array stores the optimized

coefficients of the model.

The steps described in Algorithm 5 correspond to the classification procedure used by an EmbML output code for ***Logistic*** WEKA models. First, it selects the attributes used in the classification. Then, it preprocesses the dot product between the attribute array and the coefficients of each class to use in the softmax function. Finally, it uses these values to calculate the posterior probability of the classes and returns the label associated with the maximum value. Overall, the time complexity of this classification process is $O(NUM\_CLASSES \times NUM\_PREDICTORS)$, intuitively defined by the pair of nested loops that determines the dot products.

Algorithm 5 – Classification algorithm for the ***Logistic*** WEKA model.

```
1:  procedure LOGISTIC_CLASSIFY(attributes)
2:      newInstance[0] ← 1
3:      for i ← 1 to NUM_PREDICTORS do                ▷ Filters the used attributes
4:          if m_SelectedAttributes[i] ≤ CLASS_INDEX then
5:              newInstance[i] ← attributes[m_SelectedAttributes[i − 1]]
6:          else
7:              newInstance[i] ← attributes[m_SelectedAttributes[i]]
8:          end if
9:      end for
10:     v[NUM_CLASSES] ← {0}
11:     for i ← 0 to (NUM_CLASSES − 2) do             ▷ Preprocess the dot products
12:         for j ← 0 to NUM_PREDICTORS do
13:             v[i] ← v[i] + (m_Par[i][j] × newInstance[i])
14:         end for
15:     end for
16:     v[NUM_CLASSES − 1] ← 0
17:     for i ← 0 to (NUM_CLASSES − 1) do   ▷ Calculates the posterior probability of each
    class
18:         acc ← 0
19:         for j ← 0 to (NUM_CLASSES − 2) do
20:             acc ← acc + exp(v[j] − v[i])
21:         end for
22:         prob[i] ← 1/(acc + exp(−v[i]))
23:     end for
24:     return argmax(prob)
25: end procedure
```

5.2.3 MultilayerPerceptron

The ***MultilayerPerceptron*** is the WEKA class that trains fully-connected MLP networks for classification and regression problems. In this class, we can define some hyperparameters of the model, such as the learning rate for the backpropagation algorithm (the default value is 0.3), the number of epochs to train (the default value is 500), and the number of hidden layers

in the network and their sizes (by default, it uses one hidden layer with $\lfloor (a+c)/2 \rfloor$ neurons[2]). EmbML adopts the following variables from a model built with this class:

- ***INPUT_SIZE*** is an integer value equals to the input size – *i. e.*, the number of attributes in the model.
- ***OUTPUT_SIZE*** is an integer value equals to the output size – *i. e.*, the number of class labels.
- ***FIRST_OUTPUT*** is an integer variable that stores the index of the first output neuron.
- ***NUMBER_OF_NEURONS*** is an integer number that represents the total number of neurons in the MLP network.
- ***m_attributeBases*** is a one-dimensional array that contains real numbers with the ranges for all the attributes. It occupies $O(INPUT\_SIZE)$ memory space and is used in the normalization process.
- ***m_attributeRanges*** is a one-dimensional array that stores real numbers with the base values for all the attributes. It consumes $O(INPUT\_SIZE)$ memory space and is used in the normalization process.
- ***m_numberOfConnections*** is a one-dimensional integer array with space complexity in the order of $O(NUMBER\_OF\_NEURONS)$. It saves, for each neuron i, the number of neurons that connect to i.
- ***m_connections*** is a two-dimensional array with integers numbers and space complexity in the order of $O(NUMBER\_OF\_NEURONS^2)$. This array stores, for each neuron i, the list of indices of other neurons that connect to i.
- ***m_weights*** is a two-dimensional array with real numbers and space complexity in the order of $O(NUMBER\_OF\_NEURONS^2)$. If $j \geq 1$, the value of $m\_weights[i][j]$ represents the weight of the edge between neuron i and its j-th connection. The content of $m\_weights[i][0]$ is the bias associated with neuron i.

Algorithm 6 describes the working process used by EmbML output codes for a ***MultilayerPerceptron*** WEKA classifier. Compared with the WEKA original source code, the main differences are that the EmbML version uses iterative methods (instead of recursive) and processes neurons according to the order of the layers, from input to output (instead of depth-first search), which can reduce classification time and memory usage. This process includes normalizing the input attributes, propagating the values through the neurons, and returning the class label related to the output neuron with the highest value.

[2] In this context, a is the number of attributes and c is the number of classes.

Algorithm 6 – Classification algorithm for the ***MultilayerPerceptron*** WEKA model.

```
1:  procedure MULTILAYERPERCEPTRON_CLASSIFY(attributes)
2:      for i ← 0 to (INPUT_SIZE − 1) do                    ▷ Normalizes the input
3:          if m_attributeRanges[i] ≠ 0 then
4:              attributes[i] ← (attributes[i] − m_attributeBases[i])/(m_attributeRanges[i])
5:          else
6:              attributes[i] ← (attributes[i] − m_attributeBases[i])
7:          end if
8:      end for
9:      m_value ← forward(attributes)                       ▷ Forwards the input through the network
10:     for i ← 0 to (OUTPUT_SIZE − 1) do                   ▷ Gets the output values
11:         theArray[i] ← m_value[i + FIRST_OUTPUT]
12:     end for
13:     return argmax(theArray)
14: end procedure
```

In Algorithm 7, we present the procedure that forwards the input values until they reach an output. Following a topological order of the layers, this algorithm computes the weighted sum of all signals from the input connections of a neuron and produces its output value using an activation function. In ***MultilayerPerceptron*** WEKA networks, the neurons can use sigmoid or linear activation functions, but EmbML only supports the first one. At the final, the algorithm returns an array with the output produced for each neuron.

Algorithm 7 – Forward function for the ***MultilayerPerceptron*** WEKA model.

```
1:  procedure FORWARD(attributes)
2:      for i ← 0 to (INPUT_SIZE − 1) do        ▷ Copies the input values to the first neurons
3:          m_value[i] ← attributes[i]
4:      end for
5:      for i ← INPUT_SIZE to (NUMBER_OF_NEURONS − 1) do
6:          value ← m_weights[i][0]                          ▷ Adds the bias value
7:          for j ← 0 to (m_numberOfConnections[i] − 1) do   ▷ Calculates the weighted sum
8:              value ← value + (m_weights[i][j + 1] × m_value[m_connections[i][j]])
9:          end for
10:         m_value[i] ← activationFunction(value)           ▷ Generates the output value
11:     end for
12:     return m_value
13: end procedure
```

Hence, the task of classifying an instance using these procedures has a time complexity of $O(NUMBER\_OF\_NEURONS^2)$, because the maximum total number of neuron connections lies in this order, regardless of the number of layers in the model.

5.2.4 SMO

The ***SMO*** is a class from WEKA that produces binary SVM classifiers using the sequential minimal optimization algorithm (PLATT, 1998). For dealing with multiclass problems, this class automatically builds a collection of binary classifiers using the one-versus-one approach. The training configuration of an ***SMO*** model includes setting the complexity constant (the default value is 1) and the kernel function to use (the polynomial is the default choice). In the polynomial kernel, we can define the exponent value (the default is 1.0, which represents a linear kernel) and choose to apply, or not, the inhomogeneous version (by default, the kernel does not use it). In the RBF kernel, we can determine the gamma coefficient value (the default is 0.01). An EmbML classifier code produced for the ***SMO*** class includes the following parameters:

- ***INPUT_SIZE*** is an integer value equals to the number of attributes in the input.
- ***NUM_CLASSES*** is an integer variable that contains the number of class labels.
- ***CLASS_INDEX*** is an integer number that stores the index of the class in the input array.
- ***M_EXPONENT*** is a variable with a real number that represents the exponent used in the polynomial kernel.
- ***GAMMA*** is a variable with a real number that saves the gamma parameter used in the RBF kernel.
- ***M_LOWERORDER*** is a boolean variable that indicates whether the model uses the inhomogeneous version of the polynomial kernel.
- ***TOTAL_SV*** is an integer value equals to the total number of support vectors in an ***SMO*** model with polynomial or RBF kernel, considering all binary classifiers.
- ***minArray*** is a one-dimensional array with real numbers and $O(INPUT\_SIZE)$ space complexity. This variable contains the minimum values for each attribute used in the normalization step.
- ***maxArray*** is a one-dimensional array with real numbers and $O(INPUT\_SIZE)$ space complexity. This variable saves the maximum values for each attribute used in the normalization step.
- ***m_b*** is a two-dimensional array with real numbers that denote the thresholds for each binary classifier. The space complexity of this variable is in the order of $O(NUM\_CLASSES^2)$.
- ***m_sparseWeights*** is a three-dimensional array with real numbers and demands a memory space of $O(NUM\_CLASSES^2 \times INPUT\_SIZE)$. This variable is only available in models with a linear kernel and stores, for each binary classifier, the weight associated with the attributes.

- ***m_size*** is a two-dimensional integer array that contains the number of support vectors of each binary classifier. This array consumes $O(NUM\_CLASSES^2)$ memory space and is available for all models, except those with a linear kernel.
- ***m_class_alpha*** is a three-dimensional array with real values and $O(TOTAL\_SV)$ memory complexity. For each support vector, this array stores the precalculated product of its class value (transformed to 1 or -1) and Lagrange multiplier.
- ***m_AttValues*** is a four-dimensional array consisting of real values that represent the attributes of each support vector. The space complexity of this variable is in the order of $O(TOTAL\_SV \times INPUT\_SIZE)$.

The procedure presented in Algorithm 8 describes how an EmbML output code implements the classification process for ***SMO*** WEKA classifiers. The first step consists of normalizing the input data. Then, it iterates over each binary classifier to determine the prediction results. Since this model applies the one-versus-one voting strategy, it produces $k(k-1)/2$ binary classifiers for a problem with k labels. After collecting all intermediate results, it returns the index of the class with more votes.

Algorithm 8 – Classification algorithm for the ***SMO*** WEKA model.

```
1:  procedure SMO_CLASSIFY(attributes)
2:      for i ← 0 to INPUT_SIZE do                          ▷ Normilizes the input
3:          if maxArray[i] = minArray[i] or minArray[i] = NAN then
4:              attributes[i] ← 0
5:          else
6:              attributes[i] ← (attributes[i] − minArray[i])/(maxArray[i] − minArray[i])
7:          end if
8:      end for
9:      result[NUM_CLASSES] ← {0}
10:     for i ← 1 to (NUM_CLASSES − 1) do ▷ Calculates the output of each binary classifier
11:         for j ← 0 to (i − 1) do
12:             output ← SVMOutput(i, j, attributes)          ▷ Produces one vote
13:             if output > 0 then
14:                 result[i] ← result[i] + 1
15:             else
16:                 result[j] ← result[j] + 1
17:             end if
18:         end for
19:     end for
20:     return argmax(result)          ▷ Returns the index of the class with more votes
21: end procedure
```

The binary classification steps vary depending on the kernel function chosen for the model. When using the linear kernel, Algorithm 9 defines this process, which consists of comparing the input with the separating hyperplane.

Algorithm 9 – SVMOutput function for the linear kernel.

```
1: procedure SVMOUTPUT(i, j, attributes)
2:     result ← 0
3:     for p1 ← 0 to (INPUT_SIZE − 1) do                ▷ Processes all attributes
4:         if p1 ≠ CLASS_INDEX then
5:             result ← result + (attributes[p1] × m_sparseWeights[i][j][p1])  ▷ Compares the input with the hyperplane
6:         end if
7:     end for
8:     result ← result − m_b[i][j]
9:     return result
10: end procedure
```

For the polynomial kernel, Algorithm 10 presents this procedure, which has to calculate the dot product between all support vectors and the input, apply the kernel function to the result, and multiply by the Lagrange and class coefficients.

Algorithm 10 – SVMOutput function for the polynomial kernel.

```
1: procedure SVMOUTPUT(i, j, attributes)
2:     result ← 0
3:     for k ← 0 to (m_size[i][j] − 1) do                ▷ Processes all support vectors
4:         resultAux ← 0
5:         for p1 ← 0 to INPUT_SIZE do
6:             if p1 ≠ CLASS_INDEX then
7:                 resultAux ← resultAux + (attributes[p1] × m_AttValues[i][j][k][p1])  ▷ Compares the input with each support vector
8:             end if
9:         end for
10:        if M_LOWERORDER = TRUE then                ▷ For the inhomogeneous version
11:            resultAux ← resultAux + 1.0
12:        end if
13:        resultAux ← pow(resultAux, M_EXPONENT)  ▷ Applies the polynomial function
14:        result ← result + (m_class_alpha[i][j][k] × resultAux)  ▷ Multiplies by the coefficients
15:    end for
16:    result ← result − m_b[i][j]
17:    return result
18: end procedure
```

The process for the RBF kernel is relatively similar to the polynomial kernel, except that it applies a different method to determine the distance between the input and the support vectors, and uses another kernel function, as described in Algorithm 11.

In contrast to the WEKA source code, the EmbML implementation for this model only stores the precalculated multiplication of the class value and the Lagrange multiplier from each support vector, instead of having both values separately, eliminates support vectors associated

Algorithm 11 – SVMOutput function for the RBF kernel.

```
1: procedure SVMOUTPUT(i, j, attributes)
2:     result ← 0
3:     for k ← 0 to (m_size[i][j] − 1) do                ▷ Processes all support vectors
4:         resultAux ← 0
5:         p1 ← 0
6:         for p1 ← 0 to INPUT_SIZE do
7:             if p1 ≠ CLASS_INDEX then
8:                 resultAux ← resultAux + (attributes[p1] − m_AttValues[i][j][k][p1])^2   ▷ Compares the input with each support vector
9:             end if
10:        end for
11:        resultAux ← exp(−GAMMA × resultAux)           ▷ Applies the RBF function
12:        result ← result + (m_class_alpha[i][j][k] × resultAux)   ▷ Multiplies by the coefficients
13:    end for
14:    result ← result − m_b[i][j]
15:    return result
16: end procedure
```

with null coefficients, and iterates sequentially over the support vectors.

Finally, the time complexity for the entire classification step using the linear kernel is $O(NUM\_CLASSES^2 \times INPUT\_SIZE)$, considering the loops involved in the procedures. As for the models with the polynomial and RBF kernels, this complexity is in the order of $O(TOTAL\_SV \times INPUT\_SIZE)$, since they compare the input with all support vectors from every binary classifier.

5.3 Scikit-learn Classes

The subsequent classes correspond to the scikit-learn implementations of the classifiers available in EmbML.

5.3.1 DecisionTreeClassifier

The ***DecisionTreeClassifier*** is a scikit-learn class that implements an optimized version of the CART method to train a decision tree model. In the constructor method of this class, the developer can set the function to measure the quality of a split (the default choice is the Gini impurity), the strategy used to determine the split at each node (by default, it chooses the best split), the maximum depth of the tree (by default, it is unlimited), the minimum number of samples required to split an internal node (the default value is 2), and other hyperparameters. Then, the EmbML classifier for this class contains the following variables:

- ***LEN_TREE*** is an integer number equals to the size (*i.e.*, number of nodes) of the decision tree.
- ***NUM_CLASSES*** is an integer value that represents the number of class labels.
- ***attributeIndex*** is a one-dimensional integer array that occupies $O(LEN\_TREE)$ memory space. This array stores the index of the attribute tested in each internal node of the tree. For leaves, it saves its corresponding class index.
- ***threshold*** is a one-dimensional array with real numbers and $O(LEN\_TREE)$ space complexity. For each internal node, this array saves its division point, determining the interval values for the right and left children.
- ***children_left*** is a one-dimensional integer array that saves the left child indices of every internal node. This array has a memory complexity in the order of $O(LEN\_TREE)$.
- ***children_right*** is a one-dimensional integer array with $O(LEN\_TREE)$ space complexity. It contains the right child indices of each internal node.
- ***classes*** is a one-dimensional array with integer values that denote the class labels of the model. It consumes $O(NUM\_CLASSES)$ memory space.

Algorithm 12 indicates the steps that EmbML classifier codes perform to process the input and produce a classification result for a ***DecisionTreeClassifier*** model. The procedure is similar to the ***J48*** WEKA version: it starts at the root and traversals the tree until it reaches a leaf node. The internal nodes test the attribute values and decide the path to follow next (right or left children), and the leaf nodes designate the label for the input. The complete process has a time complexity of $O(LEN\_TREE)$[3], justified by the same idea presented for the ***J48*** WEKA models in Subsection 5.2.1.

Algorithm 12 – Classification algorithm for the ***DecisionTreeClassifier*** scikit-learn model.

```
1:  procedure DecisionTreeClassify_classify(attributes)
2:      i ← root
3:      while i ≠ −1 and children_left[i] ≠ −1
          and children_right[i] ≠ −1 do          ▷ Ends when it finds a leaf
4:          if attributes[attributeIndex[i]] ≤ threshold[i] then
5:              i ← children_left[i]             ▷ Proceeds to the left child
6:          else
7:              i ← children_right[i]            ▷ Proceeds to the right child
8:          end if
9:      end while
10:     return classes[attributeIndex[i]]        ▷ Returns the corresponding class
11: end procedure
```

[3] This complexity is also proportional to the height of the tree.

It is also possible to transform the binary tree format into if-then-else statements, as indicated in Algorithm 13. Compared to the previous strategy, time and memory complexities remain the same.

Algorithm 13 – Converting a ***DecisionTreeClassifier*** scikit-learn model into if-then-else statements.

```
1: procedure GENERATE_CONDITIONALS(node)
2:     if node = −1 or children_left[node] = −1
         or children_right[node] = −1 then          ▷ leaf node: creates return statement
3:         return Create_Statement("return"
                    +string(classes[attributeIndex[node]]))
4:     else                                  ▷ internal node: creates if-then-else statement
5:         return Create_Statement("if(attributes[" + string(attributeIndex[node])
                    +"] ≤ threshold[" + string(node) + "])")
                    + Generate_Conditionals(children_left[node])   ▷ recursion to left child
                    + Create_Statement("else")
                    + Generate_Conditionals(children_right[node])▷ recursion to right child
6:     end if
7: end procedure
```

5.3.2 LinearSVC and LogisticRegression

The scikit-learn implements the logistic regression and SVM (with linear kernel) models in the classes ***LogisticRegression*** and ***LinearSVC***, respectively. Considering that they both represent linear models, these classes have identical classification methods and variables. Also, in multiclass problems, they automatically apply the one-versus-rest scheme. The set of hyperparameters of these models includes the regularization parameter (the default value is 1.0) and, for the ***LogisticRegression***, the algorithm to use in the optimization problem (by default, it applies an approximation of the Broyden–Fletcher–Goldfarb–Shanno algorithm (LIU; NOCEDAL, 1989)). The EmbML output codes for these models incorporate the following parameters:

- ***INPUT_SIZE*** is an integer value that corresponds to the number of attributes in the input.
- ***NUM_CLASSES*** is an integer value equals to the number of class labels.
- ***intercept*** is a one-dimensional array with real numbers representing the independent terms of the linear model. This variable has $O(NUM\_CLASSES)$ memory complexity.
- ***coef*** is a two-dimensional array with real numbers and a space complexity in the order of $O(NUM\_CLASSES \times INPUT\_SIZE)$. It saves the estimated coefficients of the linear model.
- ***classes*** is a one-dimensional integer array with $O(NUM\_CLASSES)$ memory complexity and stores the class labels.

In Algorithm 14, we show the procedure employed by EmbML classifiers for the ***LinearSVC*** and the ***LogisticRegression*** models to perform the classification of an instance. It examines each class by calculating the linear combination of the model coefficients and the input values. Finally, the predicted label is the one associated with the highest value resulting from the previous step. As this model iterates over the complete input for each possible class, it spends $O(NUM\_CLASSES \times INPUT\_SIZE)$ operations to finish the task.

Algorithm 14 – Classification algorithm for the ***LinearSVC*** and the ***LogisticRegression*** scikit-learn models.

```
1:  procedure LinearSVC_LogisticRegression_classify(attributes)
2:      for i ← 0 to (NUM_CLASSES − 1) do                ▷ Iterates over each class
3:          scores[i] ← intercept[i]
4:          for j ← 0 to (INPUT_SIZE − 1) do
5:              scores[i] ← scores[i] + (coef[i][j] × attributes[j])   ▷ Linear combination
6:          end for
7:          if NUM_CLASSES = 2 then                      ▷ Binary classifier
8:              if scores[i] > 0 then
9:                  return classes[1]
10:             else
11:                 return classes[0]
12:             end if
13:         end if
14:     end for
15:     return classes[argmax(scores)]                   ▷ Returns the class label
16: end procedure
```

5.3.3 MLPClassifier

The ***MLPClassifier*** class implements fully-connected MLP models in scikit-learn using the backpropagation technique. This class enables to set some model configurations, such as the number of hidden layers and their sizes (by default, it builds one hidden layer with 100 neurons), the activation function for the hidden layer neurons (ReLU is the default choice), the solver for weight optimization (the default method is the Adam, proposed by Kingma and Ba (2014)), and the maximum number of iterations (the default value is 200). An EmbML output classifier for this class has the parameters described below:

- ***INPUT_SIZE*** is an integer variable that has the number of attributes in the input.
- ***N_LAYERS*** is an integer value equals to the number of layers in the trained network.
- ***N_NEURONS*** is an integer value corresponding to the total number of neurons in the model.
- ***sizes*** is a one-dimensional integer array that contains the number of neurons in each layer. This array occupies $O(N\_LAYERS)$ memory space.

- ***intercept*** is a two-dimensional array with real numbers and space complexity in the order of $O(N\_LAYERS \times N\_NEURONS)$. It stores the bias value of each neuron in the network.

- ***coef*** is a three-dimensional array with real values representing the weights of the network neurons. The memory complexity of this variable is $O(N\_NEURONS^2)$.

- ***classes*** is a one-dimensional integer array that saves the class labels and consumes $O(NUM\_CLASSES)$ memory.

Algorithm 15 presents the strategy that EmbML adopts in its output codes from ***MLPClassifier*** models to classify an example. The whole process consists of dealing with two variables: one array that saves the input of the current layer, and another array to store the produced output. In the beginning, we copy the attribute values to the output array. Then, we examine each layer by swapping the input and output arrays, forwarding the input to produce the layer output, and applying the activation function to these values. In the end, the output array stores the signals from the output layer neurons, so it just returns the label associated with the neuron holding the highest value.

Algorithm 15 – Classification algorithm for the ***MLPClassifier*** scikit-learn model.

```
1:  procedure MLPClassifier_classify(attributes)
2:      for i ← 0 to (INPUT_SIZE − 1) do                    ▷ Copies the input values
3:          output[i] ← attributes[i]
4:      end for
5:      for i ← 0 to (N_LAYERS − 2) do                      ▷ Goes over the layers
6:          swap(output, input)                             ▷ Last output becomes current input
7:          output ← forward(attributes, input, i, sizes[i], sizes[i + 1])   ▷ Forwards the input
8:          for j ← 0 to (sizes[i + 1] − 1) do
9:              if i + 1 ≠ N_LAYERS − 1 then                ▷ Hidden layer neuron
10:                 output[j] ← activation_hidden(output[j])
11:             else                                        ▷ Output layer neuron
12:                 output[j] ← activation_output(output[j])
13:             end if
14:         end for
15:     end for
16:     return classes[argmax(output)]                      ▷ Returns class label
17: end procedure
```

In Algorithm 16, we describe the procedure that takes an input array and generates the output array for a given network layer. It simply iterates over the neurons of that layer, calculating the weighted sum of their connection weights and the input array. In conclusion, the overall cost of classification time using these procedures is $O(N\_NEURONS^2)$, which is explained by the argument previously presented for the ***MultilayerPerceptron*** WEKA model in Subsection 5.2.3.

Algorithm 16 – Forward function for the ***MLPClassifier*** scikit-learn model.

```
1: procedure FORWARD(attributes, input, layer_index, input_size, output_size)
2:     for i ← 0 to (output_size − 1) do          ▷ Process each neuron of the current layer
3:         acc_sum ← 0
4:         for j ← 0 to (input_size − 1) do
5:             acc_sum ← acc_sum + (coef[layer_index][i][j] × input[j])  ▷ Weighted sum of the input signals
6:         end for
7:         output[i] ← acc_sum + intercept[layer_index][i]                ▷ Adds bias
8:     end for
9:     return output                ▷ Returns the output for each neuron of the current layer
10: end procedure
```

5.3.4 SVC

The ***SVC*** class provides an implementation of the SVM classifier based on the LIBSVM library (CHANG; LIN, 2011). When creating a model with this class, we can select some different hyperparameters: the regularization parameter (the default value is 1.0), the kernel function (RBF is the default choice), the degree of the polynomial kernel function (the default value is 3), gamma coefficient (the default value is $1/(a \times v)$)[4], and the limit of iterations (by default, it is unlimited). An ***SVC*** object handles multiclass problems employing the one-versus-one scheme, and the EmbML output for this class includes the following parameters:

- ***INPUT_SIZE*** is an integer value that denotes the number of attributes in the input.
- ***MODEL_L*** is an integer value equals to the total number of support vectors of all binary classifiers in the model.
- ***NR_CLASS*** is an integer variable containing the number of possible classes.
- ***GAMMA*** is a real number equals to the gamma coefficient of the polynomial and RBF kernels.
- ***COEF0*** is a real variable that represents the independent term of the polynomial kernel.
- ***DEGREE*** is a real number that saves the degree of the polynomial kernel function.
- ***support_vectors*** is a two-dimensional array with real values. This array contains the support vectors of the model and requires an $O(MODEL\_L \times INPUT\_SIZE)$ memory space.
- ***dual_coef*** is a two-dimensional array with real numbers and $O(MODEL\_L)$ space complexity. It stores the trained coefficients of each support vector.

[4] In this context, a is the number of attributes and v is the variance of all training data.

- ***intercept*** is a one-dimensional array with real values that denote the intercept points of each binary classifier. This variable has $O(NR\_CLASS^2)$ memory complexity.

- ***end*** is a two-dimensional integer array that occupies $O(NR\_CLASS^2)$ memory space. To understand its content, let us consider a combination of classes (i, j) in a one-versus-one strategy. Then, we use the support vectors in the interval $[end[i][j], end[i][j+1])$ to represent the class i, and the support vectors in the interval $[end[j][i], end[j][i+1])$ to denote the class j. These intervals correspond to indices of elements in the $index\_sv[i]$ and $index\_sv[j]$ arrays, respectively.

- ***index_sv*** is a two-dimensional array with integer values and has $O(MODEL\_L)$ memory complexity. This variable stores indices of positions from the *k_value* array, which contains the resulting values of the kernel function applied to the input and all support vectors of the model. The array $index\_sv[i]$ has the indices of the support vectors from binary ***SVC*** classifiers that involve class i. The elements of the array $index\_sv[i]$ are organized in the intervals defined by the variable $end[i]$. For instance, the vectors in *k_value* at indices from $index\_sv[i][end[i][j]]$ to $index\_sv[i][end[i][j+1]] - 1$ are those associated with the class i in the binary model responsible for the combination (i, j).

- ***classes*** is a one-dimensional integer array that saves the class labels and consumes $O(NR\_CLASS)$ space complexity.

In Algorithm 17, we exhibit the classification procedure implemented by EmbML output codes for ***SVC*** models. The first task comprises precomputing the values of the kernel function on the input example combined with all support vectors of the model. Next, it iterates over each one-versus-one class combinations to produce the votes of the binary classifiers. With the support vectors representing both classes, these votes are obtained by calculating the weighted sum of the results of the kernel function and their corresponding coefficients. In the end, the procedure returns the class that collected the highest number of votes.

As for the ***SVC*** kernel functions supported by EmbML, Algorithm 18 and Algorithm 19 describes the methods used to obtain the result of polynomial and RBF functions, respectively. These processes are similar in the sense that they both include comparing the attribute values of the input example and the support vector, and applying a transformation function to the resulting value.

Using an equivalent thought as the one presented for the ***SMO*** WEKA models in Subsection 5.2.4, we can acknowledge that the entire classification process of this current case has $O(MODEL\_L \times INPUT\_SIZE)$ time complexity.

Algorithm 17 – Classification algorithm for the ***SVC*** scikit-learn model.

```
1: procedure SVC_CLASSIFY(attributes)
2:     for i ← 0 to (MODEL_L − 1) do
3:         k_value[i] ← k_function(attributes, support_vectors[i]) ▷ Precalculates the kernel function values for the input and each support vector
4:     end for
5:     vote[NR_CLASS − 1] ← 0
6:     p ← 0
7:     for i ← 0 to (NR_CLASS − 2) do                ▷ Iterates over each class combination
8:         for j ← i + 1 to (NR_CLASS − 1) do
9:             acc_sum ← 0
10:            for k ← end[j][i] to (end[j][i + 1] − 1) do   ▷ Sums the values from support vectors of class j
11:                acc_sum ← acc_sum + (dual_coef[j][k] × k_value[index_sv[j][k]])
12:            end for
13:            for k ← end[i][j] to (end[i][j + 1] − 1) do   ▷ Sums the values from support vectors of class i
14:                acc_sum ← acc_sum + (dual_coef[i][k] × k_value[index_sv[i][k]])
15:            end for
16:            acc_sum ← acc_sum + intercept[p]              ▷ Adds the intercept point
17:            p ← p + 1
18:            if acc_sum > 0 then                  ▷ Saves the vote from the current binary classifier
19:                vote[i] ← vote[i] + 1
20:            else
21:                vote[j] ← vote[j] + 1
22:            end if
23:        end for
24:    end for
25:    return classes[argmax(vote)]                  ▷ Returns the class label
26: end procedure
```

Algorithm 18 – Kernel function for a polynomial model.

```
1: procedure K_FUNCTION(attributes, support_vectors)
2:     sum ← 0
3:     for i ← 0 to (INPUT_SIZE − 1) do              ▷ Iterates over each attribute
4:         sum ← sum + (attributes[i] × support_vectors[i])   ▷ Compares their values
5:     end for
6:     return pow((GAMMA × sum) + COEF0, DEGREE)     ▷ Applies the transformation function
7: end procedure
```

5.4 Discussion

In Chart 7, we present the time and memory complexities for the EmbML implementations of WEKA and scikit-learn classifiers. First of all, note that WEKA and scikit-learn versions of the same model are essentially asymptotic equivalents. This observation suggests that the decision of the training tool to use will not extraordinarily impact the classifier performance.

Algorithm 19 – Kernel function for a RBF model.

```
1: procedure K_FUNCTION(attributes, support_vectors)
2:     sum ← 0
3:     for i ← 0 to (INPUT_SIZE − 1) do          ▷ Iterates over each attribute
4:         sum ← sum + (attributes[i] − support_vectors[i])^2   ▷ Compares their values
5:     end for
6:     return exp(−GAMMA × sum)                  ▷ Applies the transformation function
7: end procedure
```

Chart 7 – Time and memory complexities for WEKA and scikit-learn classifiers.

Model class	Time complexity	Memory complexity
***J48* WEKA**	$O(LEN\_TREE)$	$O(LEN\_TREE)$
***Logistic* WEKA**	$O(NUM\_CLASSES \times NUM\_PREDICTORS)$	$O(NUM\_CLASSES \times NUM\_PREDICTORS)$
***MultilayerPerceptron* WEKA**	$O(NUMBER\_OF\_NEURONS^2)$	$O(NUMBER\_OF\_NEURONS^2)$
***SMO* WEKA (linear kernel)**	$O(NUM\_CLASSES^2 \times INPUT\_SIZE)$	$O(NUM\_CLASSES^2 \times INPUT\_SIZE)$
***SMO* WEKA (polynomial and RBF kernels)**	$O(TOTAL\_SV \times INPUT\_SIZE)$	$O(TOTAL\_SV \times INPUT\_SIZE)$
***DecisionTreeClassifier* scikit-learn**	$O(LEN\_TREE)$	$O(LEN\_TREE)$
***LinearSVC* and *LogisticRegression* scikit-learn**	$O(NUM\_CLASSES \times INPUT\_SIZE)$	$O(NUM\_CLASSES \times INPUT\_SIZE)$
***MLPClassifier* scikit-learn**	$O(N\_NEURONS^2)$	$O(N\_NEURONS^2)$
***SVC* scikit-learn**	$O(MODEL\_L \times INPUT\_SIZE)$	$O(MODEL\_L \times INPUT\_SIZE)$

Furthermore, observe that we expressed most of their complexity functions in terms of variables that are unique for the models. These variables, such as the length of a decision tree or the total number of support vectors, are also intrinsically dependent on other factors, such as the probability distribution of the addressed problem and the training hyperparameters. From the theoretical perspective, establishing an adequate comparison of processing time and memory consumption for these different models is a difficult task and may cause incorrect conclusions. For instance, the memory complexity of the decision tree classifiers is linearly proportional to the size of the tree, but this last value can grow exponentially in relation to the height of the tree. The number of neurons and layers in the MLP models are hyperparameters defined by the user in the training phase, and they can assume unpredictable values. Additionally, the number of support vectors in the SVM models with polynomial and RBF kernels can be as large as the size of the training set in a worst-case analysis. Consequently, the SVM classifiers would require storing all training examples in memory and iterate through them during classification. For these reasons, one solution that can promote a better comprehension and comparison of the time and

memory behaviors of these classifiers is to use empirical evaluation.

5.5 Final Considerations

In this chapter, we explained the details of the WEKA and scikit-learn classes supported by EmbML. We described the variables that store the parameters of the models and each classification procedure. This study also included an asymptotic analysis of the storage consumption of these variables and the classification processing time. Lastly, we compared the time and memory complexities of these models and stated the limitations of only performing such an analysis. As a complement of this chapter, we propose in the next chapter a comparative analysis of all models supported by EmbML using benchmark datasets to help understand their performance in a practical way.

CHAPTER

6

COMPARATIVE ANALYSIS

6.1 Initial Considerations

This chapter is an extension of the results presented in Silva, Souza and Batista (2019a) and describes the experimental evaluation to assess the performance of the EmbML classifiers. These assessments include six datasets of different real-world applications and six microcontrollers with diverse memory and processing specifications. First, we employ three metrics to evaluate their performance and measure the impact of the proposed classifier modifications. Then, we apply these same metrics to confront the performance of the classifiers generated by EmbML with classifiers provided by some other tools in the literature.

6.2 Experimental Setup

The experiments presented in this chapter consider three primary metrics to assess the performance of the classifiers generated with EmbML or related tools: accuracy rate, classification time, and memory usage. The accuracy and classification time are estimated using examples from a test set. The memory usage represents the total size of the compiled classifier code. In the context of these experiments, a high accuracy indicates that the classifier correctly solves the problem. Also, lowering the classification time and memory usage is advantageous, since it may allow opting for simpler hardware, which reduces the costs and power requirements for the system.

In the following sections, the values measured for the memory usage derives from the GNU *size* utility[1], which lists the section sizes and the total size for the input file. As for the classification time, we collected the mean value per instance using the *micros*[2] function, which returns the number of microseconds since the microcontroller began running the current program.

[1] <https://ftp.gnu.org/old-gnu/Manuals/binutils-2.12/html_node/binutils_8.html>
[2] <https://www.arduino.cc/reference/en/language/functions/time/micros/>

Therefore, we call this function twice, before and after calling the classification function, and use the difference between the collected values. To better estimate these values, we executed each classifier ten times in the test set. In all experiments, the microcontrollers read the test set examples from a microSD memory card. However, the results consider only the time spent in the classification process, not incorporating the time for reading and preparing the input. In some cases, the size of the classifiers surpassed the microcontroller's storage capacity, so we acknowledged them in the analysis.

The study of the classifiers produced by EmbML includes the performance comparison of three representations for (signed) real numbers:

- floating-point with 32 bits (referred as FLT), defined by the Institute of Electrical and Electronics Engineers (IEEE) 754 standard (KAHAN, 1996) and provided by the compiler;
- fixed-point with 32 bits (referred as FXP32), using the $Q22.10$ format provided by EmbML;
- and fixed-point with 16 bits (referred as FXP16), using the $Q12.4$ format provided by EmbML.

6.2.1 Datasets

The study of each classifier performance considers six benchmark datasets from real-world applications related to sensing. In general, sensors are embedded in low-power devices to perform the data gathering task. Therefore, such problems represent a typical use-case scenario for the EmbML classifiers. Follows a brief description of each dataset:

- **Aedes aegypti-sex** (REIS; MALETZKE; BATISTA, 2018). In this application, an optical sensor measures the wingbeat frequency and other audio related features of flying insects that cross through a sensor light. The classification task is to identify the sex of *Aedes aegypti* mosquitoes;
- **Asfault-streets** (SOUZA; GIUSTI; BATISTA, 2018). The problem explored in this dataset involves evaluating the pavement conditions of urban streets, using the data collected from an accelerometer sensor installed in a car. 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** (SOUZA; GIUSTI; BATISTA, 2018). This dataset represents the same previously presented problem of pavement conditions evaluation but performed on highways instead of urban streets. The main difference is the lack of the *poor* class. Also, the difference in car speed during data collection imposes some significant variations on the class patterns;

- **GasSensorArray** (VERGARA *et al.*, 2012). This problem 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*. The classification objective is to identify which gas constitutes each example;

- **PenDigits** (ALIMOGLU; ALPAYDIN, 1996). This dataset 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** (ANGUITA *et al.*, 2013). This application employs a waist-mounted smartphone and uses its accelerometer and gyroscope sensors to obtain data for the following human activities: *walking*, *climbing stairs*, *downstairs*, *sitting*, *standing*, and *lying down*.

Table 1 shows the main characteristics of these datasets. Note that they vary on the number of classes and attributes – variables that directly affect the time and memory complexities of the classifiers, as we saw in Section 5.4. The experimental evaluation references these datasets using the identifiers in the first column.

Table 1 – 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

To evaluate the classifier performances, we applied a 70/30 holdout validation. This method requires splitting each dataset into two stratified and mutually exclusive subsets, in which the training part takes 70% of the original dataset instances, and the test set incorporates the 30% remaining.

6.2.2 Classifiers

In the experiments described later in this chapter, we have not performed any search to find the best set of hyperparameters for each combination of the classification model and dataset. Instead, we trained most of the classifiers using their default hyperparameter values provided by WEKA and scikit-learn. Given that we wanted to explore all possibilities supported by EmbML, there were a few cases in which we had to manually set some hyperparameters:

- as the *SMO* WEKA class produces SVM models with 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* scikit-learn class, we adjusted it to produce SVM models with a polynomial kernel (using $degree = 2$), since the RBF kernel is its default choice;
- and, for the *MLPClassifier* scikit-learn class, we changed it to also create MLP networks that apply the sigmoid activation function, considering that the ReLU function is its default option.

Therefore, the accuracy rates reported in these analyses may not represent the best possible values for each dataset and classifiers. Keep in mind that the main concerns of these assessments are to study the viability of embedded implementation of the classifiers, determine whether the classifier execution in the microcontrollers maintain the same accuracy obtained in the desktop version or not, and also optimize their time and memory behaviors.

6.2.3 Microcontrollers

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

Thus, the following microcontrollers were considered for the experimental evaluation:

i. **ATmega328/P** (ATMEL CORPORATION, 2016) available in the Arduino Uno (Figure 9) and produced by Atmel. It is a low-power 8-bit microcontroller with simple characteristics when compared to the other chosen models;

ii. **ATmega2560** (ATMEL CORPORATION, 2014) available in the Arduino Mega 2560 (Figure 10). It is an 8-bit microcontroller manufactured by Atmel. Besides some improvement to memory storage, it is very similar to the previous microcontroller;

iii. **AT91SAM3X8E** (ATMEL CORPORATION, 20145) available in the Arduino Due (Figure 11) and also developed by Atmel. It is a high-performance 32-bit microcontroller and represents one of the most robust options available in Arduino platforms;

iv. **MK20DX256VLH7** (FREESCALE SEMICONDUCTOR, INC., 2012a; FREESCALE SEMICONDUCTOR, INC., 2012b) available in the Teensy 3.1 and Teensy 3.2 (Figure 12). It is a 32-bit microcontroller produced by Freescale. It represents a class of processors with intermediate processing and memory power;

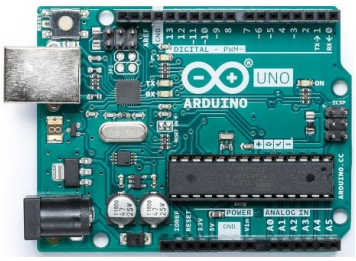
v. **MK64FX512VMD12** (NXP SEMICONDUCTORS, 2016; NXP SEMICONDUCTORS, 2017a) available in the Teensy 3.5 (Figure 13) and manufactured by Freescale. It has a

single-precision FPU. In addition, both the frequency of the clock and the memory storage are better compared to the previous version;

vi. **MK66FX1M0VMD18** (NXP SEMICONDUCTORS, 2017b; NXP SEMICONDUCTORS, 2015) available in the Teensy 3.6 (Figure 14), and also produced by Freescale. It is the most powerful processor in these experiments. This unit operates with 32 bits and includes a single-precision FPU.

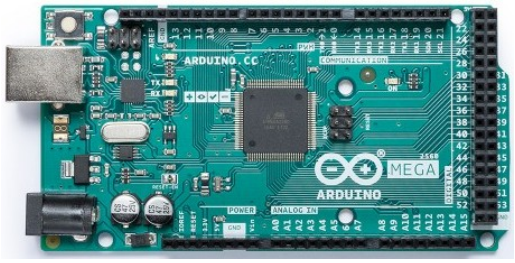
The microcontrollers of the Teensy platform have an ARM Cortex-M4 core and the Arduino Due platform has an ARM Cortex-M3 core. The Arduino Uno and Arduino Mega 2560 has a low-power microcontroller from the AVR family.

Figure 9 – Arduino Uno.



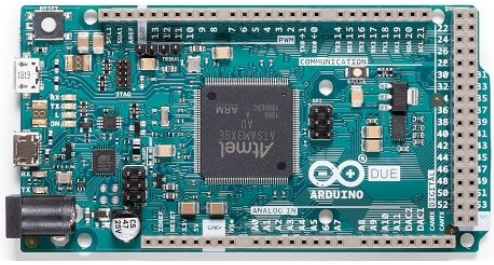
Source: <https://store.arduino.cc/usa/arduino-uno-rev3>

Figure 10 – Arduino Mega 2560.



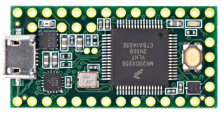
Source: <https://store.arduino.cc/usa/mega-2560-r3>

Figure 11 – Arduino Due.



Source: <https://store.arduino.cc/usa/due>

Figure 12 – Teensy 3.2.



Source: <https://www.pjrc.com/store/teensy32.html>

Figure 13 – Teensy 3.5.

Source: <https://www.pjrc.com/store/teensy35.html>

Figure 14 – Teensy 3.6.

Source: <https://www.pjrc.com/store/teensy36.html>

Table 2 shows some of the main specifications of the chosen embedded platforms, such as microcontroller models, clock rate, and the amount of Static Random-Access Memory (SRAM) and flash memory available.

Table 2 – Characteristics of the evaluated embedded platforms.

Platform	Microcontroller	Clock (MHz)	SRAM (kB)	Flash (kB)
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

6.3 Analysis of the EmbML Classifiers

The experiments described in this section consist of a sanity check and an analysis of classification time and memory consumption. First, we compare the accuracy rates obtained by the EmbML classifiers running on the embedded devices 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 the different real number representations.

6.3.1 Accuracy

Table 3 and Table 4 show 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 these tables, the symbol "-" means that the produced code was too large and did not fit in any microcontroller's memory.

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:

- in D2 with the *MultilayerPerceptron* WEKA from Table 3, the accuracy increases from 89.19% (in desktop) to 89.26% (with EmbML classifier using FLT);
- in D1 with the *DecisionTreeClassifier* scikit-learn from Table 4, the accuracy reduces from 98.54% (in desktop) to 98.53% (with EmbML classifier using FLT);
- and in D1, D4, and D5 with *SVC* (polynomial kernel) scikit-learn from Table 4, the accuracies from EmbML classifiers (using FLT) are lower than those obtained in desktop. In this specific case, this decrease mainly happens because the *SVC* scikit-learn employs double-precision (64 bits) floating-point operations, and EmbML only supports single-precision (32 bits).

Another important observation is that, in most cases, there is not a significant change in accuracy when using the FXP32, comparing to FLT representation. It is reasonable to assume

Table 3 – Accuracy (%) for the WEKA 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	99.00	88.48	84.28	97.41	84.71	94.34
	EmbML/FXP32	98.97	88.41	84.54	97.41	84.71	94.01
	EmbML/FXP16	97.25	87.06	68.56	58.65	84.71	79.61
Logistic	Desktop	97.71	91.61	89.00	98.97	73.00	97.35
	EmbML/FLT	97.71	91.61	89.00	98.97	73.00	97.35
	EmbML/FXP32	97.65	91.54	87.97	98.35	72.72	97.35
	EmbML/FXP16	50.06	67.57	17.96	34.86	40.81	94.40
MultilayerPerceptron	Desktop	98.67	89.19	90.29	92.84	80.46	93.62
	EmbML/FLT	98.67	89.26	90.29	92.84	80.46	93.62
	EmbML/FXP32	98.65	90.33	90.46	92.86	80.58	93.66
	EmbML/FXP16	54.40	88.62	88.49	18.38	79.88	92.72
SMO (linear kernel)	Desktop	98.39	91.96	91.75	97.13	80.67	98.38
	EmbML/FLT	98.39	91.96	91.75	97.13	80.67	98.38
	EmbML/FXP32	98.40	92.32	91.92	97.13	80.61	98.48
	EmbML/FXP16	89.97	81.44	71.39	22.35	78.34	16.73
SMO (polynomial kernel)	Desktop	98.76	92.39	91.15	99.40	89.11	98.96
	EmbML/FLT	98.76	92.39	91.15	99.40	89.11	-
	EmbML/FXP32	98.71	91.04	89.52	36.94	89.11	-
	EmbML/FXP16	59.03	27.31	39.78	14.04	44.48	-
SMO (RBF kernel)	Desktop	98.08	87.62	83.59	75.59	67.63	95.99
	EmbML/FLT	98.08	87.62	-	-	-	-
	EmbML/FXP32	97.99	87.77	-	-	-	-
	EmbML/FXP16	50.00	20.70	35.22	-	9.59	-

that they are equivalent in this aspect, so the user can opt for the one that obtains the best time or memory performance.

On the other hand, the FXP16 representation can cause a notable reduction in the accuracy rate for most classifiers due to several reasons. For instance, its interval of representable numbers is extremely limited (from -2048 to 2047.9375 in two's complement format) and can commonly be the reason for overflow in arithmetic operations. Also, underflow is very likely to happen in operations with small numbers since all consecutive numbers in FXP16 have an absolute difference of 0.0625 and, therefore, numbers in the interval $]\frac{-0.0625}{2}, \frac{0.0625}{2}[$ are rounded to 0 – the closest representable number.

6.3.2 Classification Time

Next, we compare the average time that each model spent to classify an instance. Figure 15 and Figure 16 respectively show the executions of the EmbML codes derived from WEKA and scikit-learn models. In these figures, each column of graphs describes a microcontroller, and each row denotes a dataset. Then, for a given combination of dataset and microcontroller, these graphs illustrate the time performance of different classifiers using the supported representations

Table 4 – Accuracy (%) for the scikit-learn classifiers.

Classifier	Version	D1	D2	D3	D4	D5	D6
DecisionTreeClassifier	Desktop	98.54	86.13	84.02	97.03	83.83	93.20
	EmbML/FLT	98.53	86.13	84.02	97.03	83.83	93.20
	EmbML/FXP32	98.49	85.78	84.28	97.03	83.83	92.85
	EmbML/FXP16	70.46	81.37	63.06	61.00	83.83	75.18
LinearSVC	Desktop	90.51	92.11	88.83	80.02	36.74	98.58
	EmbML/FLT	90.51	92.11	88.83	80.02	36.74	98.58
	EmbML/FXP32	86.64	92.18	88.92	35.27	36.41	98.58
	EmbML/FXP16	50.00	91.82	83.08	18.45	9.59	48.35
LogisticRegression	Desktop	98.18	90.97	84.19	98.06	71.51	98.25
	EmbML/FLT	98.18	90.97	84.19	98.06	71.51	98.25
	EmbML/FXP32	98.15	90.90	84.11	46.17	71.75	98.28
	EmbML/FXP16	50.00	90.90	83.42	18.45	40.38	98.12
MLPClassifier (ReLU)	Desktop	95.96	92.46	91.41	96.43	89.96	98.54
	EmbML/FLT	95.96	92.46	91.41	96.43	89.96	98.54
	EmbML/FXP32	96.12	92.60	91.84	96.26	89.87	98.38
	EmbML/FXP16	56.44	5.12	64.09	16.67	57.77	38.32
SVC (polynomial kernel)	Desktop	98.47	77.17	64.78	98.87	90.75	93.95
	EmbML/FLT	51.56	77.17	64.78	97.03	71.51	-
	EmbML/FXP32	50.22	77.17	64.78	18.45	9.19	-
	EmbML/FXP16	50.00	76.81	35.22	18.45	10.13	-
SVC (RBF kernel)	Desktop	58.53	88.62	86.51	21.63	18.69	95.28
	EmbML/FLT	-	88.62	86.51	-	18.69	-
	EmbML/FXP32	-	88.76	86.08	-	18.33	-
	EmbML/FXP16	50.00	20.70	35.22	21.63	9.89	18.87

for real numbers. The missing points are those that could not be determined because their classifier code was larger than the microcontroller's memory. In general, it is possible to verify the following aspects:

- In the microcontrollers that lack an FPU, the fixed-point formats reduce the classification time when compared to the FLT. In the other microcontrollers, it is not possible to notice such an improvement.

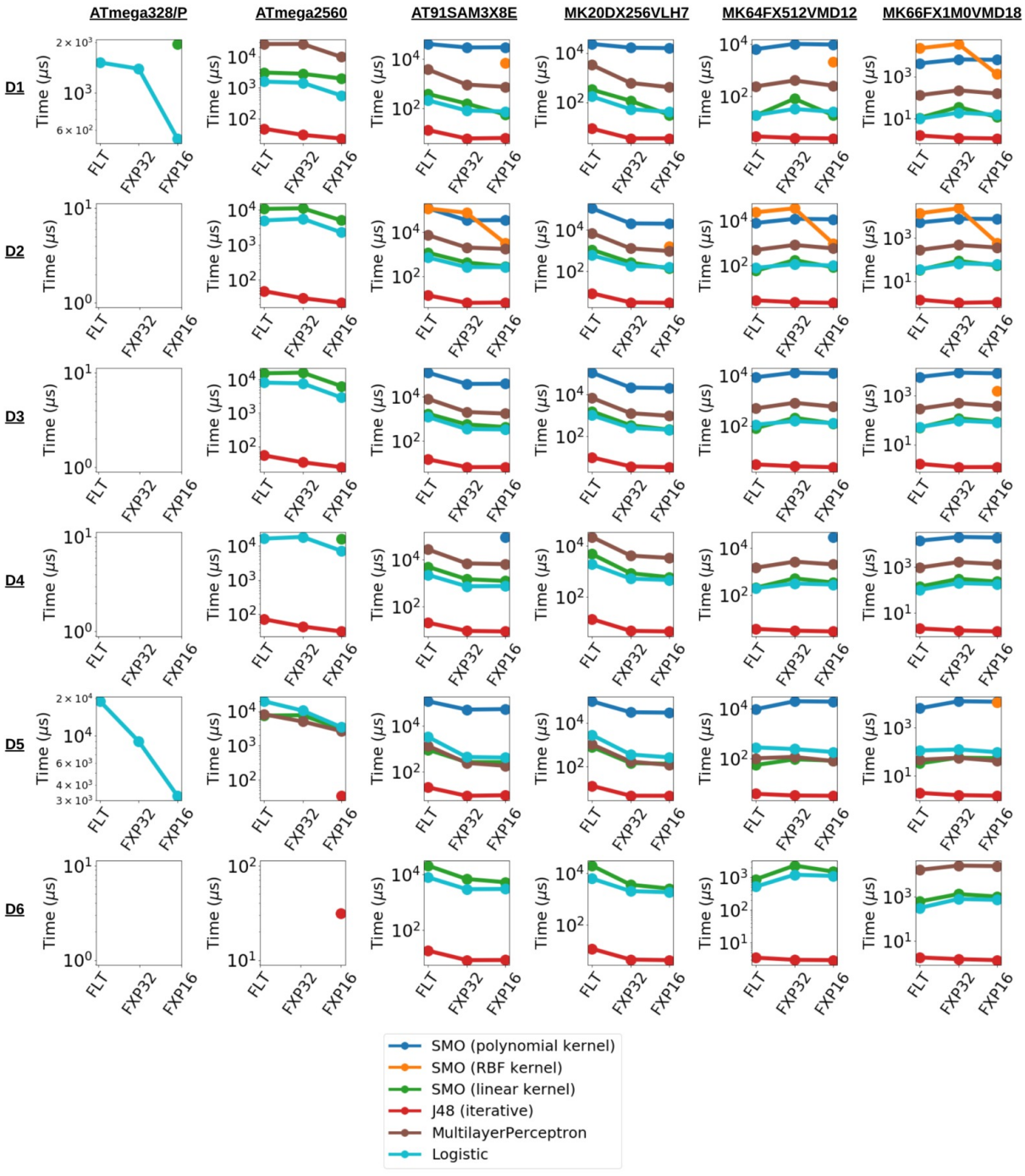
- The decision tree models deliver the lowest classification time in all cases that they can run.

- The logistic regression and SVM (linear kernel) models have similar performances, reaching the second-best results in most situations. Particularly for the scikit-learn models, it is not possible to see the *LinearSVC* results in some graphs since they coincide with the *LogisticRegression* results. These overlaps happen considering that these models have equivalent classification algorithms, as we saw in Subsection 5.3.2.

- The SVM models with the polynomial and RBF kernels are the classifiers that execute in the lowest number of cases and perform the worst results.

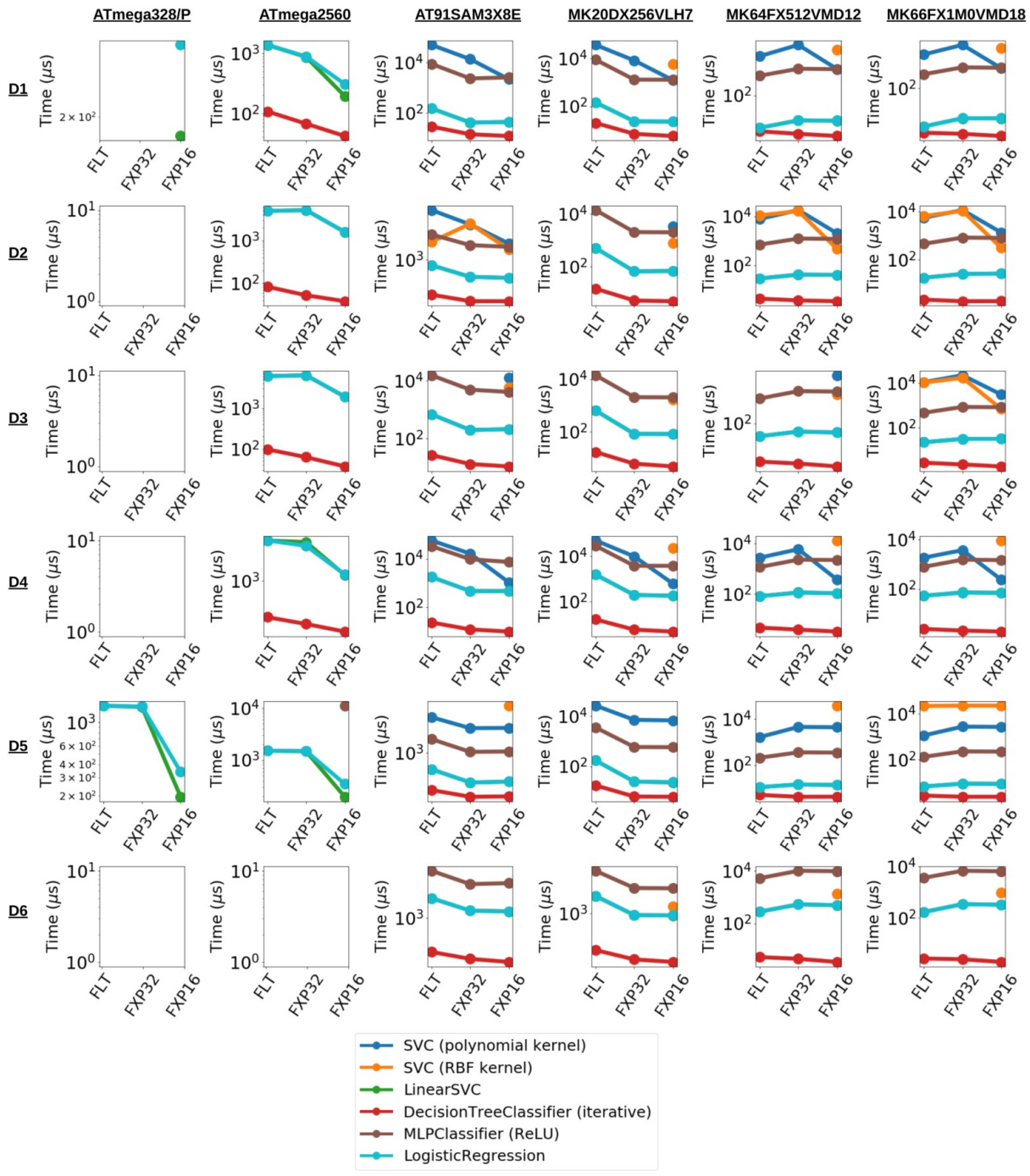
- The MLP models achieve an intermediate performance. They are faster than the SVM with polynomial and RBF kernels, but usually slower than the others.

Figure 15 – Mean classification time for WEKA classifiers.



Source: Elaborated by the author.

Figure 16 – Mean classification time for scikit-learn classifiers.



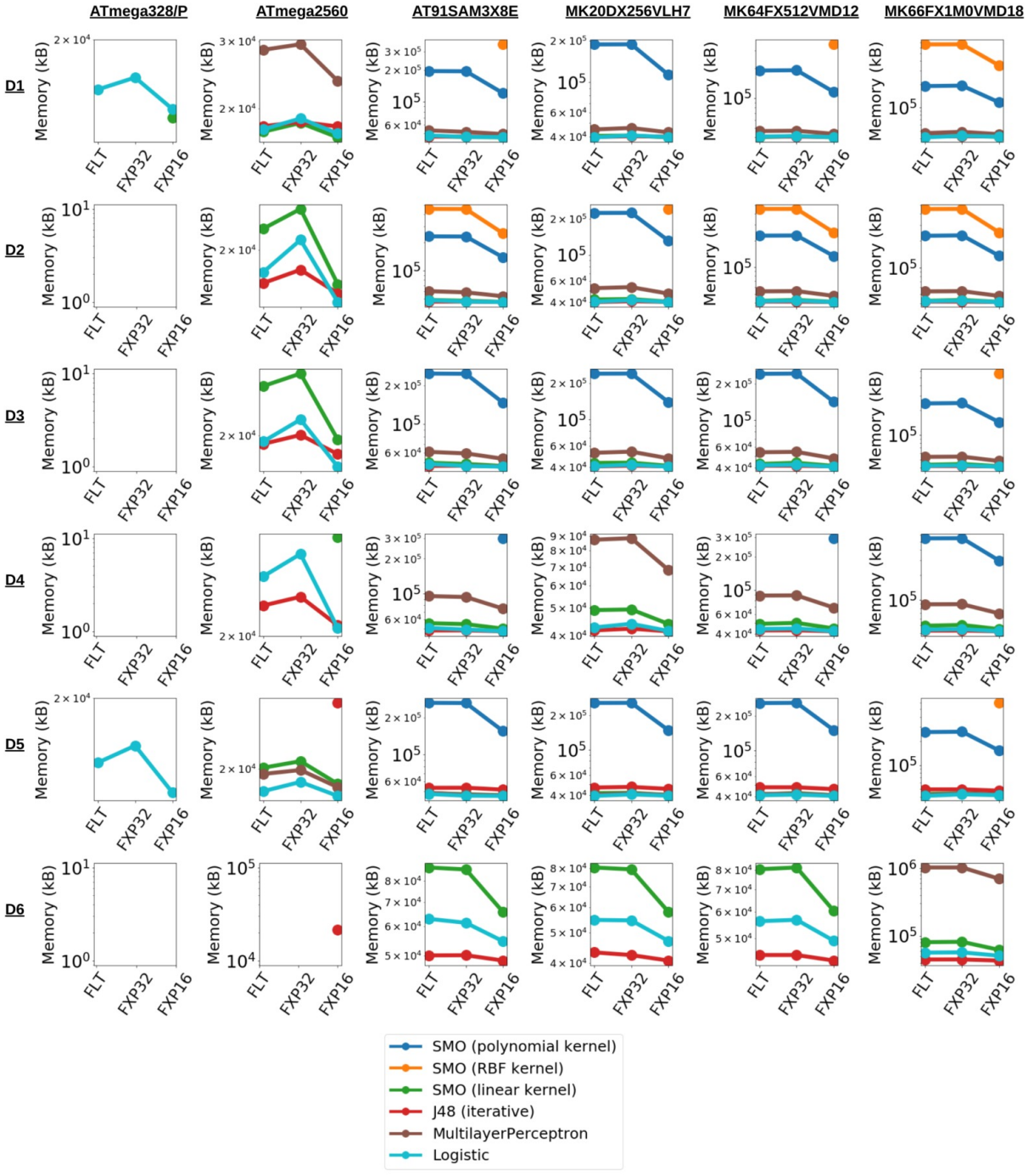
Source: Elaborated by the author.

6.3.3 Memory Usage

The last metric evaluated from the classifier performances is memory consumption. We compare the sum of the data (SRAM) and program (flash) memories among the supported classifiers in Figure 17 (for WEKA models) and Figure 18 (for scikit-learn models). The graphs presented in these figures are combined in rows and columns, as explained previously. An

examination of these graphs enables us to identify some common characteristics:

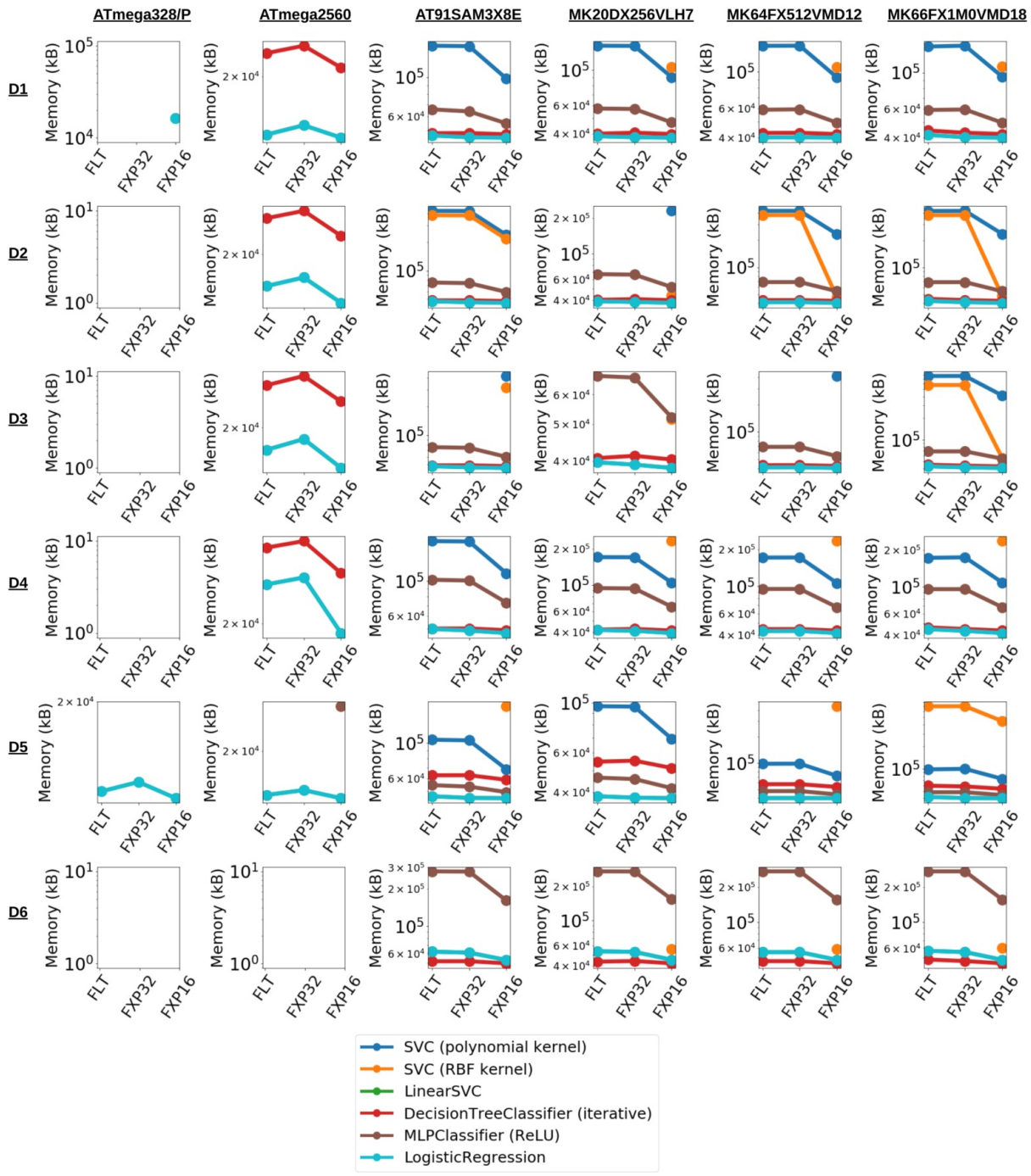
Figure 17 – Memory consumption for WEKA classifiers.



Source: Elaborated by the author.

- Comparing to the FLT representation, there is no advantage of employing the FXP32 in terms of this analysis. However, FXP16 representation is able to decrease memory usage.
- The source codes produced for the decision tree, logistic regression, and SVM (linear kernel) models are those that alternate as the best memory performance. Also, they are

Figure 18 – Memory consumption for scikit-learn classifiers.



Source: Elaborated by the author.

the classifiers most capable of executing in the two less robust microcontrollers. In the scikit-learn comparison, the *LogisticRegression* and *LinearSVC* results coincide since they have identical variables, presented in Subsection 5.3.2.

- On the other hand, the SVM classifiers with the polynomial and RBF kernels achieve the highest memory consumptions. In many situations, their variables are extremely large that

they can not fit in the available microcontroller's memory. This fact is even more evident for the SVM model with the RBF kernel.

- As for the MLP models, they again have an intermediate performance in most of the cases, comparing to the two previously defined groups.

6.4 EmbML Code Modifications

In order to understand if they positively affect the classification time performance and do not cause a negative effect on the accuracy rates, we compare in this section the modifications provided by EmbML to its output classifier source codes. It includes the analysis of approximations for sigmoid function in MLP models and the option to transform a decision tree model from an iterative structure, which is the default, to if-then-else statements. Also, since the previous section already assessed the impact of using fixed-point representations, this modification is not explicitly analyzed here as an independent case.

6.4.1 Approximations for Sigmoid Function in MLP

Table 5 and Table 6 present the accuracies estimated in each test set through applying the approximation functions – provided by EmbML and discussed in Section 4.6 – for substituting the sigmoid in MLP models of WEKA and scikit-learn, respectively. These tables contain the accuracy rates for the different real numbers representations and also the accuracy obtained by MLP models using the original sigmoid function for comparison.

Table 5 – Accuracy (%) for the *MultilayerPerceptron* WEKA 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	98.67	89.26	90.29	92.84	80.46	93.62
	EmbML/FXP32	98.65	90.33	90.46	92.86	80.58	93.66
	EmbML/FXP16	54.40	88.62	88.49	18.38	79.88	92.72
$0.5+0.5x/(1+\|x\|)$ function	EmbML/FLT	98.67	89.19	90.38	92.91	80.46	93.69
	EmbML/FXP32	98.65	89.19	90.46	92.98	80.49	93.72
	EmbML/FXP16	54.35	87.27	88.49	18.66	79.52	93.43
2-point PWL	EmbML/FLT	98.67	90.90	90.21	92.72	80.19	93.69
	EmbML/FXP32	98.65	91.04	90.12	92.76	80.22	93.69
	EmbML/FXP16	54.39	88.69	88.14	18.52	79.98	92.69
4-point PWL	EmbML/FLT	98.67	90.97	90.55	92.86	80.40	93.69
	EmbML/FXP32	98.65	90.97	90.38	92.86	80.37	93.66
	EmbML/FXP16	54.39	88.41	88.66	18.28	80.16	92.69

Contrasting with employing the original sigmoid, the highest difference in accuracy for the *MultilayerPerceptron* WEKA models (with an alternative modification) occurs in D2 using the 4-point PWL approximation and FLT. In this case, the accuracy rate increases from

Table 6 – Accuracy (%) for the *MLPClassifier* scikit-learn models with sigmoid activation function.

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	98.39	92.25	90.72	74.58	91.78	98.64
	EmbML/FXP32	98.22	92.18	90.72	74.39	91.84	98.71
	EmbML/FXP16	81.21	89.83	79.73	17.61	87.80	40.81
$0.5+0.5x/(1+\|x\|)$ function	EmbML/FLT	98.39	91.32	90.29	74.51	91.60	98.80
	EmbML/FXP32	98.26	91.32	90.21	74.39	91.54	98.71
	EmbML/FXP16	86.82	90.18	82.56	17.32	86.50	59.09
2-point PWL	EmbML/FLT	98.36	92.53	90.38	74.58	91.75	98.58
	EmbML/FXP32	98.18	92.46	90.64	74.39	91.87	98.58
	EmbML/FXP16	81.89	89.62	81.01	17.56	87.92	39.13
4-point PWL	EmbML/FLT	98.40	92.03	90.98	74.58	91.81	98.58
	EmbML/FXP32	98.19	92.11	91.24	74.39	91.87	98.64
	EmbML/FXP16	81.61	88.12	76.63	17.61	88.20	41.72

89.26% (with the sigmoid function) to 90.97% (with the 4-point PWL). As for the *MLPClassifier* scikit-learn classifiers, the maximum difference happens in D6 using the $0.5+0.5x/(1+|x|)$ function and FXP16. The accuracy increases from 40.81% (with the sigmoid version) to 59.09% (with the approximation function) in this situation. As a general rule, the accuracy values from the modified models are relatively close to the original models and can be acceptable in practice.

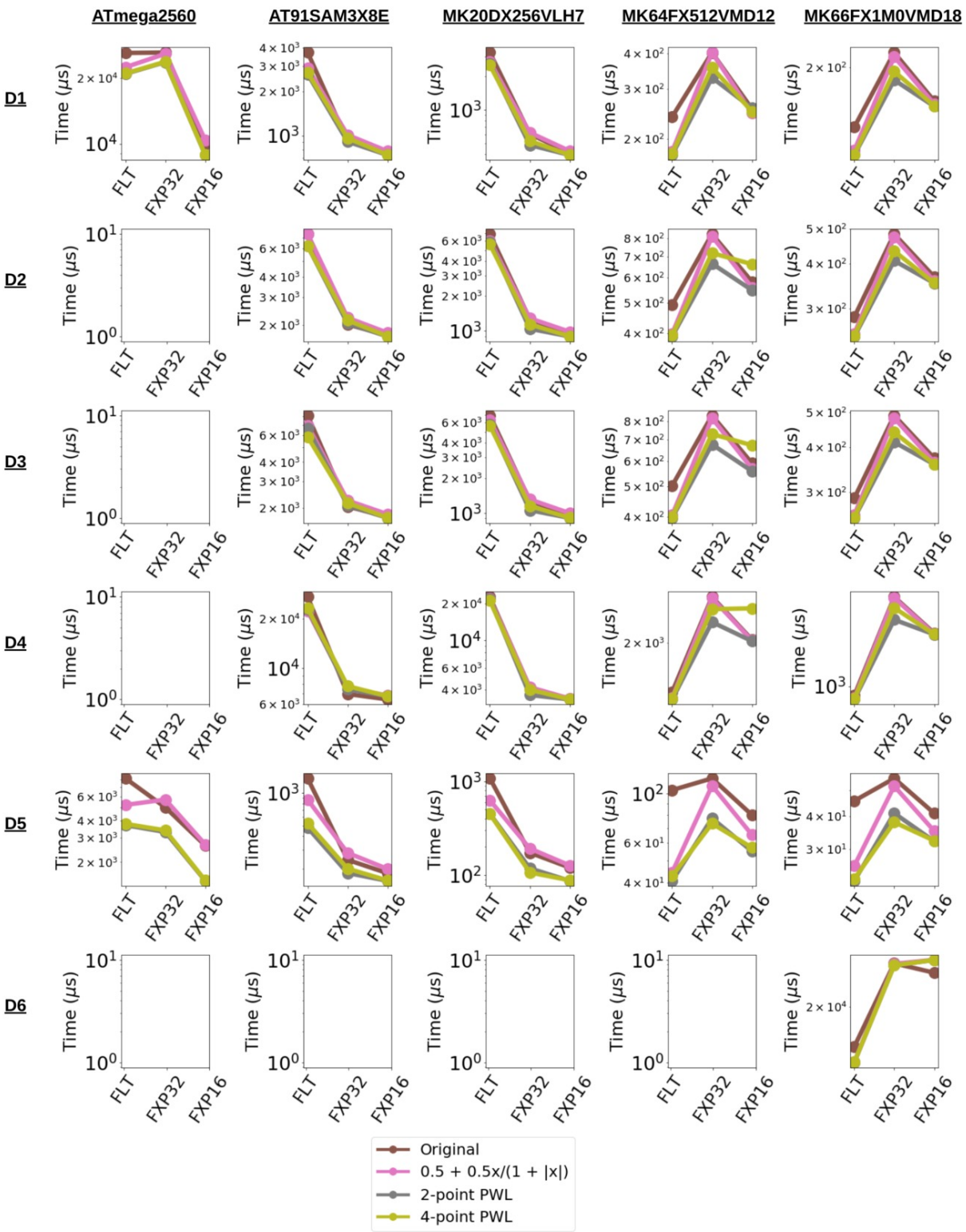
In Figure 19 and Figure 20, we exhibit the mean classification time comparison of the WEKA and scikit-learn MLP models, respectively, using the provided options for the sigmoid activation function. They do not contain the column corresponding to the ATmega328/P microcontroller, because there was not an MLP version able to execute in it. From these graphs, we can identify that these options produce similar time results in most of the cases. However, comparing among them, the use of PWL approximations can predominantly 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.

Although we collected the values for the memory usage analysis, we decided not to include them here, since the difference in this metric for using or not the modified classifiers is relatively inexpressive. This is also intuitively explained considering that the sigmoid approximations do not affect the size of the classifier variables, which usually define the memory bottleneck.

6.4.2 If-Then-Else Statements and Iterative Decision Trees

Now, we examine the source code modification provided by EmbML for decision tree models. The only specific option for this classifier consists of deciding to use the iterative structure (as in the original WEKA and scikit-learn implementations) or converting it into if-

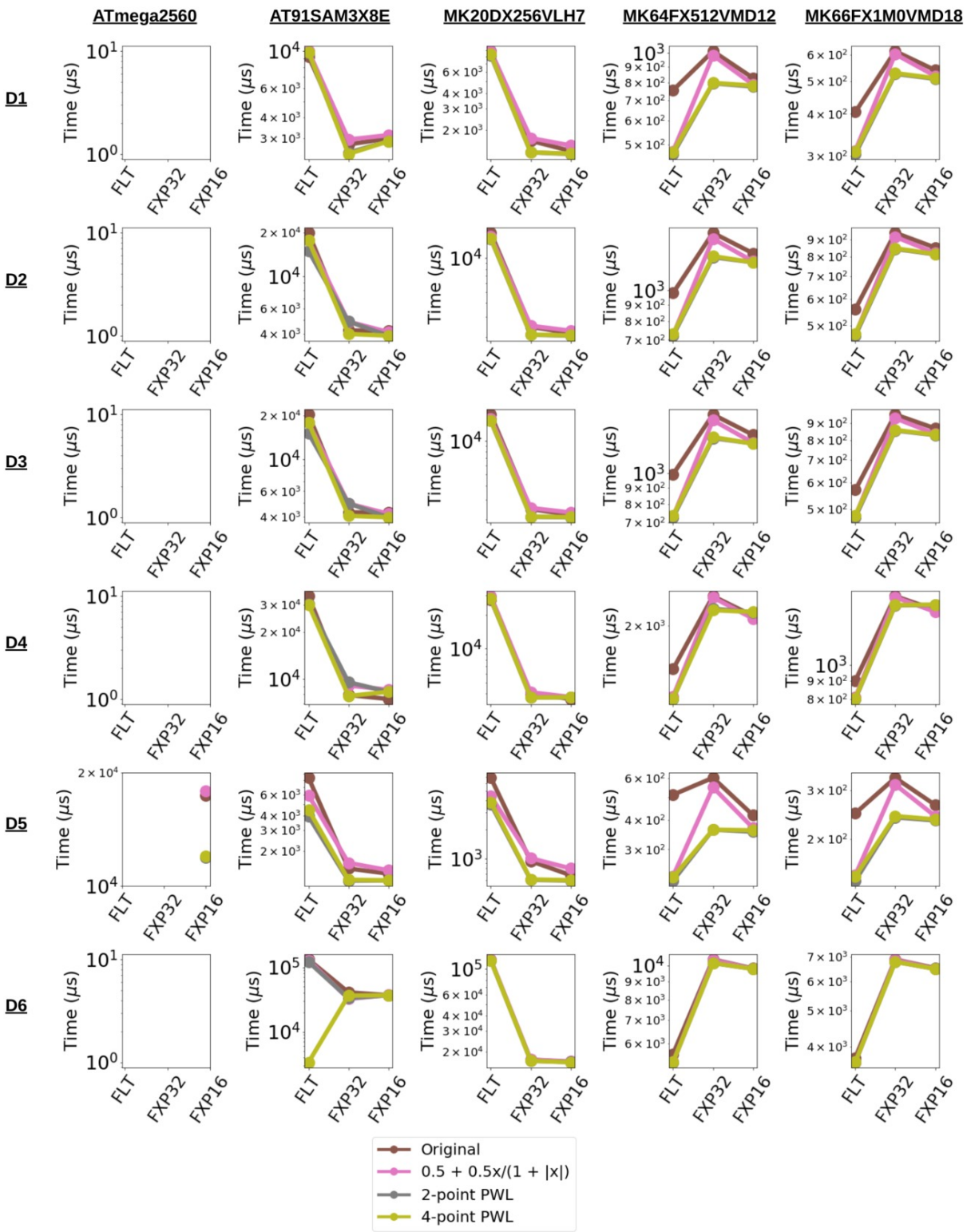
Figure 19 – Mean classification time for the *MultilayerPerceptron* WEKA classifiers.



Source: Elaborated by the author.

then-else statements. In this analysis, the only difference between these options is structural, which did not influence the accuracy results. As for memory usage, the amount of data to

Figure 20 – Mean classification time for the *MLPClassifier* scikit-learn classifiers.

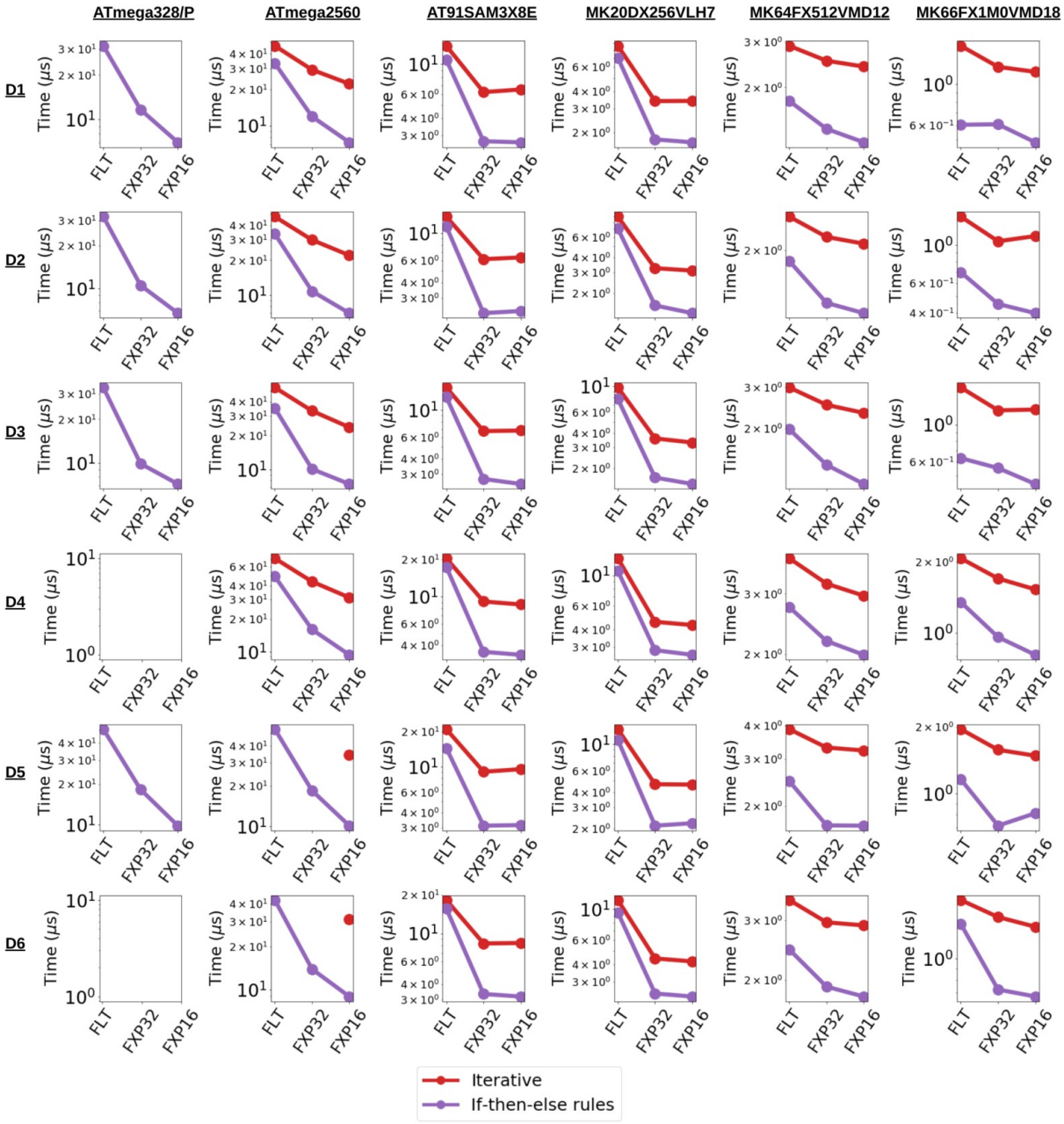


Source: Elaborated by the author.

store is not affected, but code size may increase using if-then-else statements (as discussed in Section 4.7). However, these two options achieved quite similar values in our experiments 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 the mean classification time results, as displayed in Figure 21 and Figure 22 for WEKA and scikit-learn decision tree models, respectively.

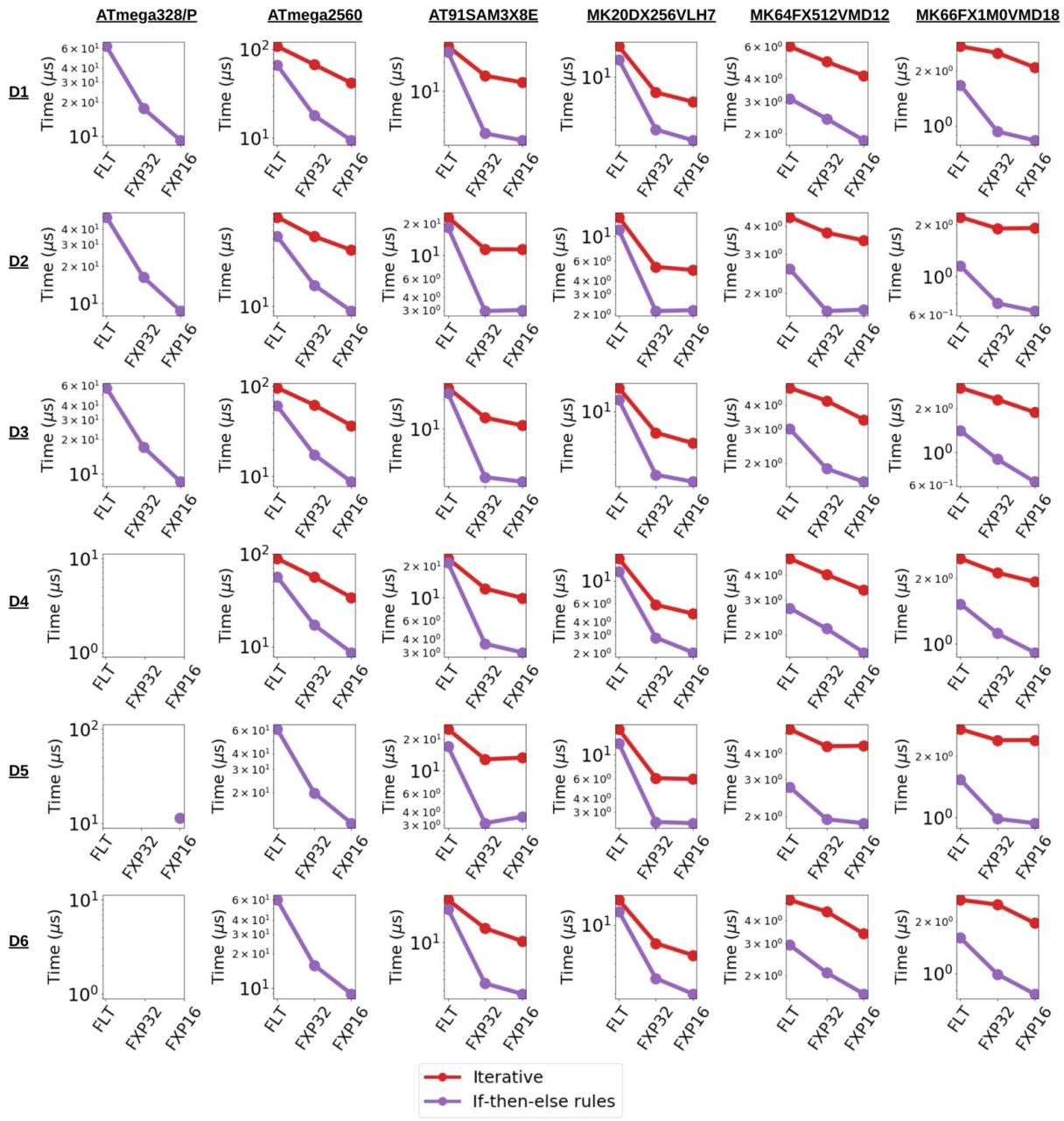
Figure 21 – Mean classification time for the *J48* WEKA classifiers.



Source: Elaborated by the author.

These graphs help to recognize that the if-then-else versions of the decision tree models always obtain better time results than the iterative option, as we expected from the arguments presented in Section 4.7. For this reason, it is highly recommended to choose the if-then-else version when generating a decision tree classifier with EmbML. Another important observation

Figure 22 – Mean classification time for the *DecisionTreeClassifier* scikit-learn classifiers.



Source: Elaborated by the author.

is that there are situations in which the iterative classifier is not able to execute, but the other option is. For instance, only the classifiers composed of if-then-else statements can execute in the ATmega328/P microcontroller. To explain these distinct behaviors, we have to include memory consumption in this analysis. Although these versions have a similar overall memory usage, converting the classifier to an if-then-else structure enabled the compiler, in some cases, to use less of the microcontroller's data memory and more of its programming memory, compared to the iterative version. Since the data memory is very restricted in most microcontrollers, it can be a bottleneck to the classifier code.

6.5 Comparing With Related Tools

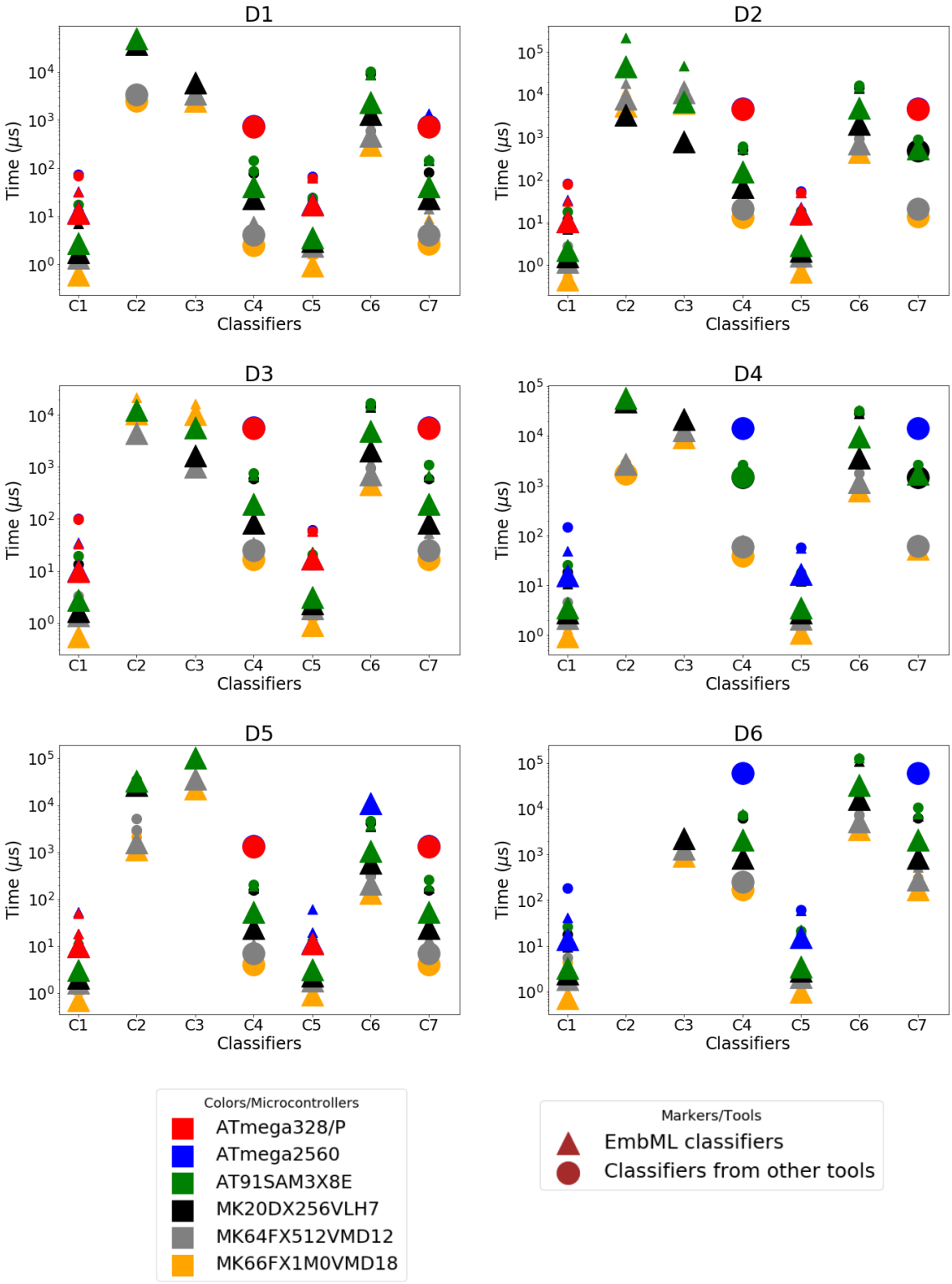
In order to show that EmbML produces competitive classifiers, this section evaluates their performance against classifiers generated by other related tools preseted in Section 2.2: emlearn (version 0.10.1), m2cgen (version 0.5.0), sklearn-porter (version 0.7.4) and weka-porter (version 0.1.0). To make a consistent comparison, we selected only models from other tools that have a direct correspondent in EmbML. For example, we consider the *MLPClassifier* scikit-learn model since both EmbML and emlearn support it. The chosen models, their identifiers and tools that support them are listed below.

- ***J48* WEKA** (referred as C1) is supported by EmbML and weka-porter. For this experiment, we used the if-then-else version of EmbML.
- ***SVC* (polynomial kernel) scikit-learn** (referred as C2) is supported by EmbML, m2cgen, and sklearn-porter.
- ***SVC* (RBF kernel) scikit-learn** (referred as C3) is supported by EmbML, m2cgen, and sklearn-porter.
- ***LinearSVC* scikit-learn** (referred as C4) is supported by EmbML, m2cgen, and sklearn-porter.
- ***DecisionTreeClassifier* scikit-learn** (referred as C5) is supported by EmbML, emlearn, m2cgen, and sklearn-porter. In this experiment, we selected the if-then-else version of EmbML.
- ***MLPClassifier* (ReLU) scikit-learn** (referred as C6) is supported by EmbML and emlearn.
- ***LogisticRegression* scikit-learn** (referred as C7) is supported by EmbML and m2cgen.

Also, we generated and evaluated the classifier versions with the same off-board-trained model and test set. For each classifier, we used the different datasets and microcontrollers to compare mean classification time, shown in Figure 23, and memory consumption (SRAM + flash), shown in Figure 24.

In these figures, each graph contains the results corresponding to a specific dataset. The graphs incorporate only the time or memory values associated with a high accuracy rate to prevent including poor solutions – *e.g.*, the FXP16 versions of EmbML classifiers are faster but usually achieve lower accuracy rate than their corresponding FLT version. Therefore, for each combination of microcontroller, dataset, and classifier, we used the following steps to decide the values to plot:

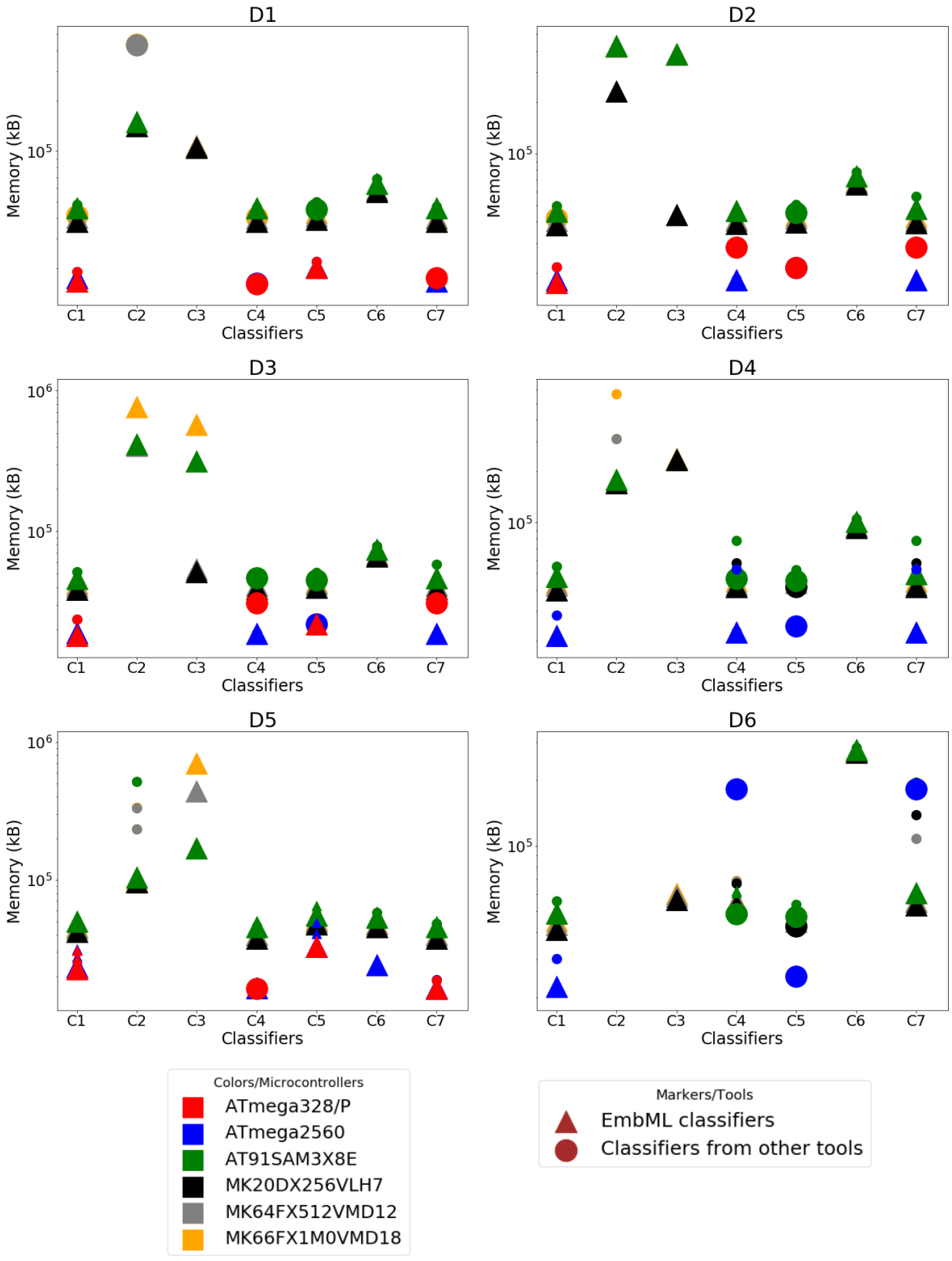
Figure 23 – Mean classification time comparison between classifiers from EmbML and related tools.



Source: Elaborated by the author.

1) unite the outcomes of the given combination produced by classifiers from EmbML and other tools;

Figure 24 – Memory usage comparison between classifiers from EmbML and related tools.



Source: Elaborated by the author.

2) determine the mean accuracy value of these results;

3) eliminate the results that achieve an accuracy lower than the mean value;

4) sort the remaining data in ascending order of classification time (or memory usage);

5) choose, at most, the three first results from the sorted list.

For instance, considering the mean classification time comparison for the *DecisionTreeClassifier* model, created using D2, and running on the AT91SAM3X8E microcontroller, Table 7 presents the values estimated using the test set during the experiments. Since the average accuracy rate is 80.54%, we exclude the EmbML/FXP16 version. Then, we sort the remaining results in ascending order of mean classification time and select the first three, which are: EmbML/FLT, EmbML/FXP32, and emlearn. Finally, the time results of these versions are the chosen points included in the D2 graph. This process repeats for each dataset, microcontroller, and classifier.

Table 7 – An example of mean classification time results for the *DecisionTreeClassifier* model, D2 dataset, and AT91SAM3X8E microcontroller.

Classifier version	Accuracy rate	Mean classification time
EmbML/FLT	84.02%	23.47 μs
EmbML/FXP32	84.28%	3.15 μs
EmbML/FXP16	63.06%	2.82 μs
emlearn	83.93%	20.85 μs
m2cgen	83.93%	27.13 μs
sklearn-porter	84.02%	36.93 μs

In the graphs, the large markers represent the lowest values for a specific combination of dataset, microcontroller, and classifier. Thus, for the example shown in Table 7, the EmbML/FXP32 point is larger than the EmbML/FLT and emlearn points in the D2 graph of Figure 23. Similarly, in many other cases, the EmbML classifiers are those that reach the best results for both classification time and memory consumption. In fact, Table 8 shows a comparison using these two metrics to evaluate, for each dataset, the number of cases that EmbML classifiers accomplish the best result and the total number of cases – *i.e.*, combinations of microcontroller and classifier – that at least one classifier code was able to execute. Therefore, the proposed method to compare the time and memory performances indicates that the EmbML classifiers were able to produce the best mean 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 frequently perform better than the other solutions, which is evidence that EmbML is an advantageous alternative.

6.6 Final Considerations

This chapter exhibited a comprehensive evaluation of the classifier source codes generated by EmbML. First, we presented the experimental setup, which contemplates a variety of datasets and microcontrollers with distinct features. Then, we analyzed the performance of

Table 8 – 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

the EmbML classifiers based on three metrics: accuracy, classification time, and memory usage. This section also focused on identifying the trade-offs about using the different representations for real numbers supported by EmbML. Next, we evaluated the impact of employing code modifications implemented and available for MLP and decision tree models. And finally, this chapter incorporated experiments to confront the performance of some classifiers produced by EmbML against others created by some related tools. The results demonstrated that EmbML was capable of generating competitive classifier codes.

CHAPTER

7

CASE STUDY: AN INTELLIGENT TRAP FOR FLYING INSECTS

7.1 Initial Considerations

This chapter presents an example of an application to employ the EmbML pipeline for developing an embedded classifier. We discuss this process using the case of an intelligent trap for automatic classification and selective capture of flying insects. In order to clearly understand the whole process, we present some of the work that has already been done by our research group. Next, we focus on the experimental analysis of performance using classification models provided by EmbML, to choose the most appropriate to deploy in the trap's resource-constrained hardware. It also considers practical experiments in two different setups to assess the behavior of the EmbML classifier.

7.2 The Intelligent Trap

The problem presented in this chapter consists of a real-world ML application for predicting flying insect species. Our research group has developed an intelligent trap (shown in Figure 25) able to classify and capture mosquitoes, according to their species (BATISTA *et al.*, 2011a; BATISTA *et al.*, 2011b; SOUZA; SILVA; BATISTA, 2013; CHEN *et al.*, 2014; QI *et al.*, 2015; SILVA *et al.*, 2015; SOUZA *et al.*, 2020). These previous studies also resulted in the development of a low-cost optical sensor to gather data from flying insects and data mining techniques to process them and distinguish the species.

The immediate importance of such a solution includes monitoring the population of specific disease-vector mosquitoes – *e.g.*, *Aedes aegypti*, a vector for transmitting dengue fever, chikungunya, Zika fever, yellow fever, and other diseases – and agricultural pests – *e.g.*, *Diaphorina citri*, a vector of the citrus greening disease. Collecting information about the spatial and

Figure 25 – The projected intelligent trap for flying insects.

Source: Souza *et al.* (2020).

temporal distribution of these insects allows local and customized actions, reducing the impact of some insect control methods to the ecological balance – such as insecticides eliminating insect pollinators – and improving their efficiency. In the context of classifying flying insects and estimating their population, the developed trap provides an automatic, faster, and cheaper option compared to existing solutions such as sticky traps, used in crop fields, that require the manual work of experts (SOUZA *et al.*, 2020).

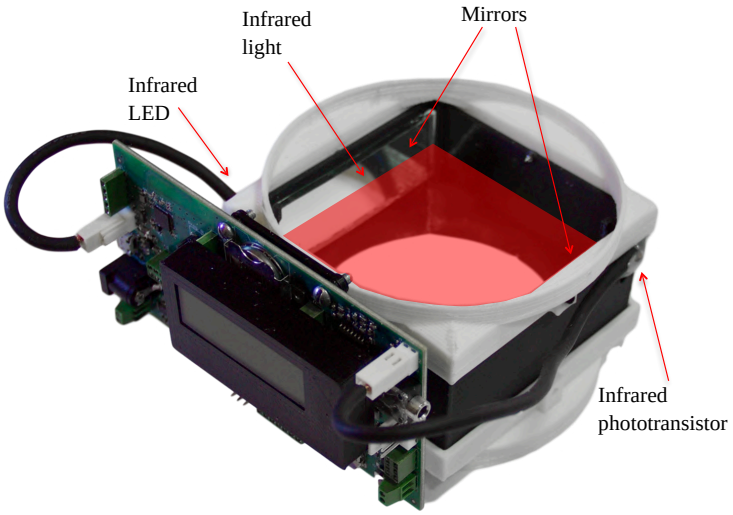
In this chapter, we particularly consider the problem of separating the *Aedes aegypti* mosquitoes into two classes: female and male. We understand that it is a relevant task for two reasons. First, we know that *Aedes aegypti* is a vector to many diseases, but only the female mosquitoes bite humans (HARRIS *et al.*, 2011; MAINS *et al.*, 2016) and, then, are the main responsible for spreading diseases. Second, since they belong to the same species and may share most physiological characteristics (such as flight behavior), we assume that accurately recognizing these classes is as challenging as distinguishing *Aedes aegypti* female from most of the other flying insects. In this scenario, the trap work to capture female *Aedes aegypti* and expel male mosquitoes.

7.2.1 Optical Sensor

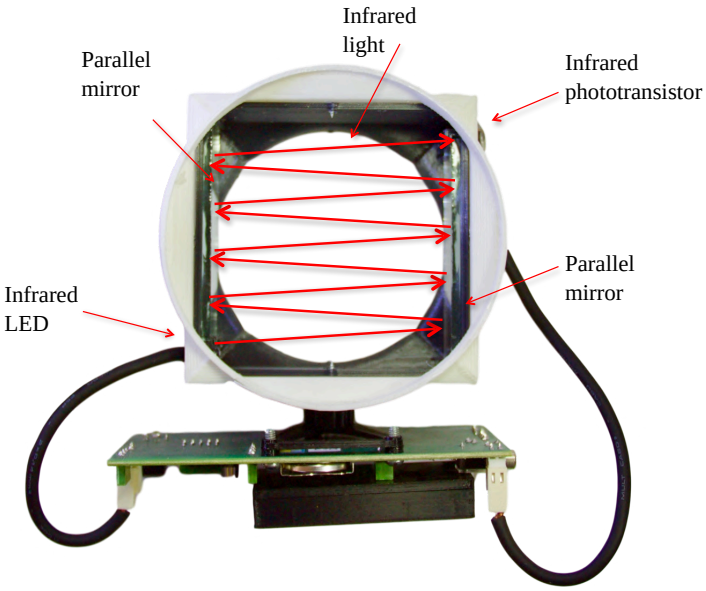
As described in Souza *et al.* (2020), the main component of the developed trap is a low-cost optical sensor that remotely gathers data from flying insects. The developed sensor, illustrated in Figure 26, is a result from previous work (BATISTA *et al.*, 2011a; BATISTA *et al.*, 2011b) and contains an infrared light-emitting diode (LED) pointed to a structure of two parallel mirrors, creating a light curtain that is captured by an infrared phototransistor. When a flying

insect crosses this curtain, the movements of its wings partially occlude the light, producing small variations captured by the phototransistor (BATISTA *et al.*, 2011a). Then, the sensor hardware is responsible for filtering, amplifying, and processing this input signal to generate the features used in the classification step. Compared to regular microphones, this sensor is a more robust solution because it is able to exclusively capture vibrations that affect the infrared light, ignoring the ambient noise or other acoustic waves.

Figure 26 – The developed optical sensor.



(a) Side-view.



(b) Top-view.

Source: Souza *et al.* (2020).

7.2.2 Developed Board

The trap system also includes a custom low-cost and low-power hardware board to interface with the sensors, processing the input data, and controlling the trap fan – that only activates to expel or capture an insect that crosses the infrared light. Some of the principal components of the hardware solution designed for this application are:

- an MK20DX256VLH7 microcontroller – the same used in the Teensy 3.2 platform – that performs the trap processing operations of capturing the input signal, preprocessing it, executing the predictive model, and activating the fan to capture (or expel) the flying insect;
- a digital filter circuit to amplify and remove noise from the input signal;
- an SGTL5000, which is a low-power stereo codec chip responsible for encoding the input analog signal as digital signal;
- an nRF51822 ultra-low-power System-on-a-Chip used for Bluetooth Low Energy communication to communicate with external systems;
- a microSD card interface to store relevant and large data, such as the recorded signals;
- and connections for the optical and ambient (light, temperature, and humidity) sensors.

Figure 27 displays an annotated photograph of the developed hardware board and its components. This hardware was also developed in previous work parallel to the optical sensor.

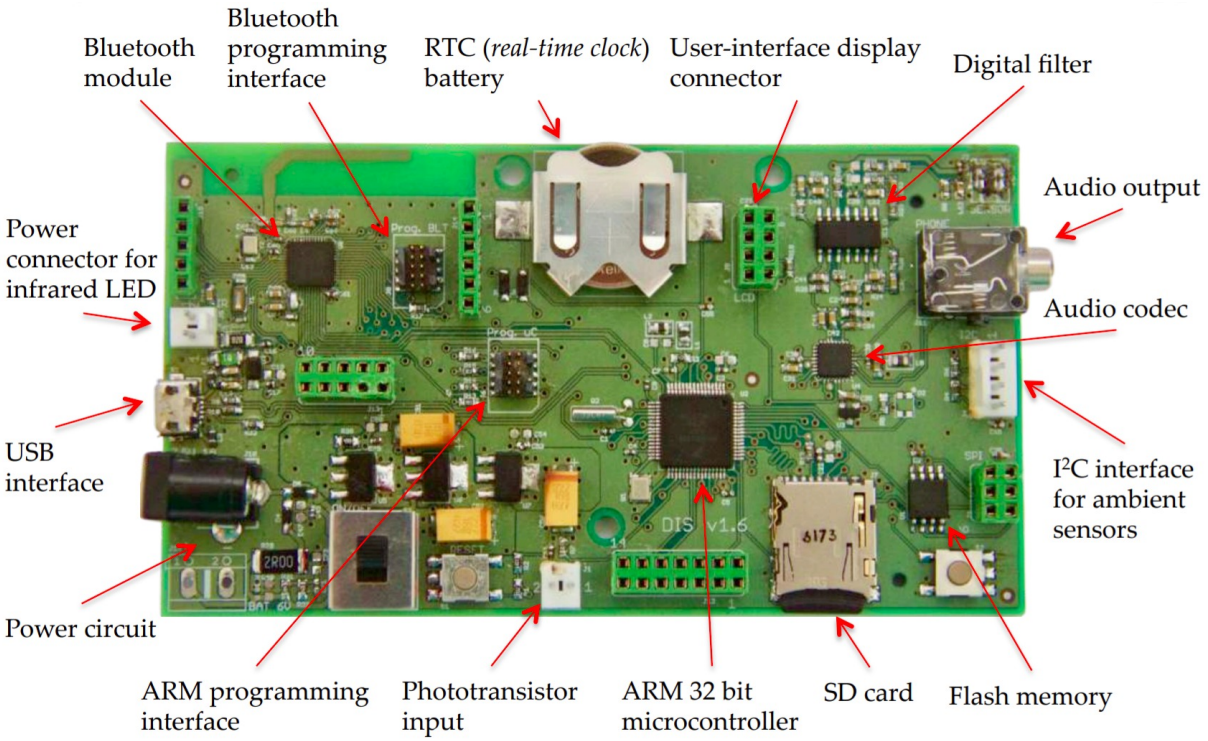
7.3 Predictive Features And Data Preprocessing

A continuous perturbation on the optical sensor generates an input signal representing the event, as illustrated in Figure 28a. The process of recording this signal starts when its amplitude stays higher than a predefined threshold for a determined amount of time, indicating that a flying insect is crossing the sensor light. It ends when the input signal amplitude stays lower than the threshold for the same amount of time, suggesting that the insect is not blocking the light curtain anymore. After this period, the collected input is processed and transformed into a feature vector, which contains several attributes extracted from the event data.

Some previous research studied features for this problem (BATISTA *et al.*, 2011b; QI *et al.*, 2015; SILVA *et al.*, 2015). A brief description of currently used attributes is presented below:

1) **Hour**: the microcontroller's real-time clock (RTC) records the hour that the event occurred. As illustrated in Figure 29, the insect flight activity varies in specific hours of the day. Thus, this feature is important to distinct species that may present different circadian rhythms (BATISTA *et al.*, 2011b);

Figure 27 – The trap's printed circuit board and its components.



Source: Elaborated by Dr. Gustavo E. A. P. A. Batista.

2) **Temperature**: a BME280 digital sensor measures the ambient temperature in Celsius degree. Temperature is a factor that influences the insect's metabolism (MELLANBY, 1936; TAYLOR, 1963) and can directly affect the flight performance of an insect (ROWLEY; GRAHAM, 1968), as shown in Figure 30;

3) **Wingbeat Frequency (WBF)**: the wingbeat frequency is estimated using the highest peak from the signal cepstrum between 100Hz and 1200Hz[1] (BATISTA *et al.*, 2011b). Since the harmonics are periodic in the spectrum, the cepstrum peak represents them in a single value, which defines the fundamental frequency – *i.e.*, the WBF. In Figure 31, we give an example of these representations for an input signal and show its fundamental frequency;

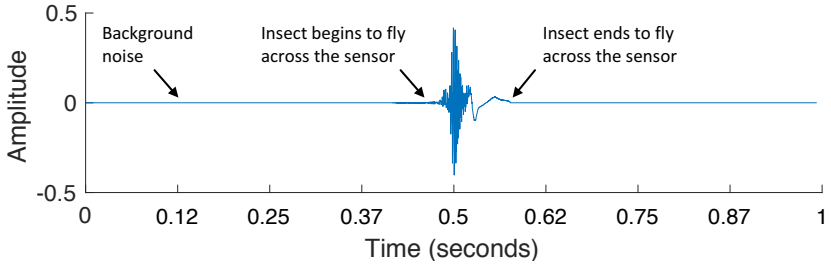
4) **Frequency Peaks (1-6)**: the values in Hertz of the six highest peaks directly obtained from the signal spectrum. As presented in Figure 28b, these peaks usually describe the fundamental frequency and other harmonics;

5) **Inharmonicity**: calculated as the standard deviation of the values from the previous six frequency peaks;

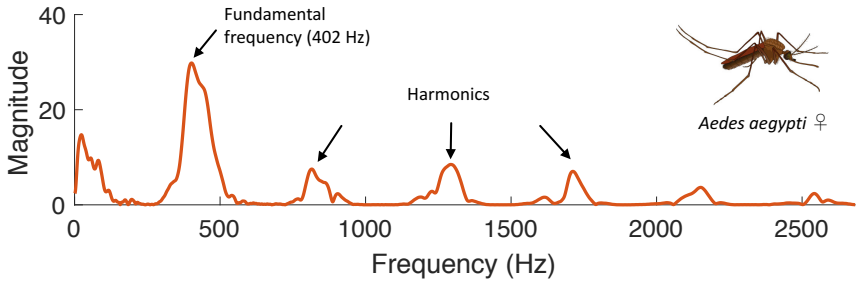
6) **Energy of Harmonics (1-26)**: the sums of magnitudes from the first four harmonics, considering 26 possible values of fundamental frequency between 100Hz and 1200Hz.

[1] Since most insects have a WBF in this range.

Figure 28 – An example of an *Aedes aegypti* female input signal obtained by the optical sensor.



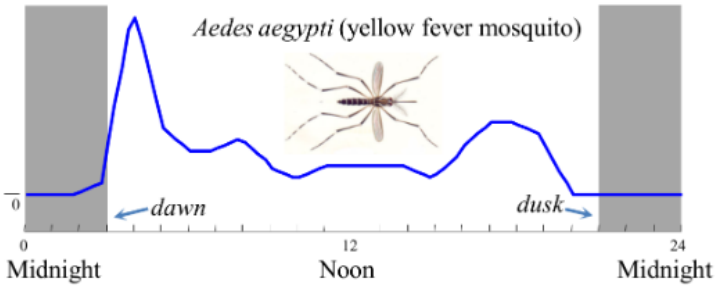
(a) Representation of a captured signal in the time domain.



(b) Representation of a captured signal in the frequency domain.

Source: Souza *et al.* (2020).

Figure 29 – The flight activity of *Aedes aegypti* mosquitoes during the day.



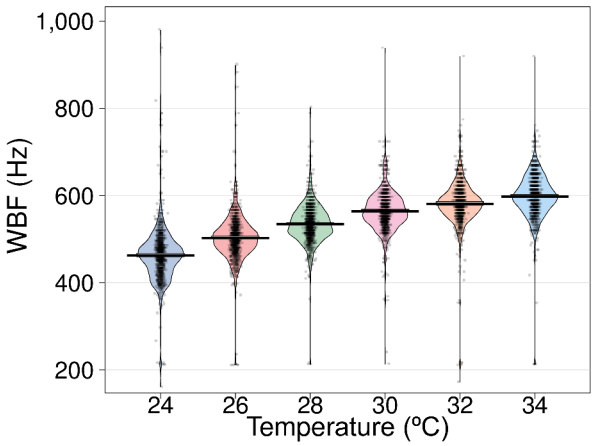
Source: Batista *et al.* (2011b).

The microcontroller, used in the designed board, is responsible for processing the input data of an event and producing the described features. For the first two features (hour and temperature), it simply uses the values provided by the RTC and the ambient sensor. But, for generating the other attributes, it has to perform some digital signal processing operations using the data gathered from the optical sensor. Considering that features 4, 5, and 6 derive from the signal spectrum, we apply the Welch's method (WELCH, 1967) to estimate it, using the steps described below.

1) Divide the signal data[2] into several intervals, each one contains 1024 points, and adjacent segments have a 50% overlap.

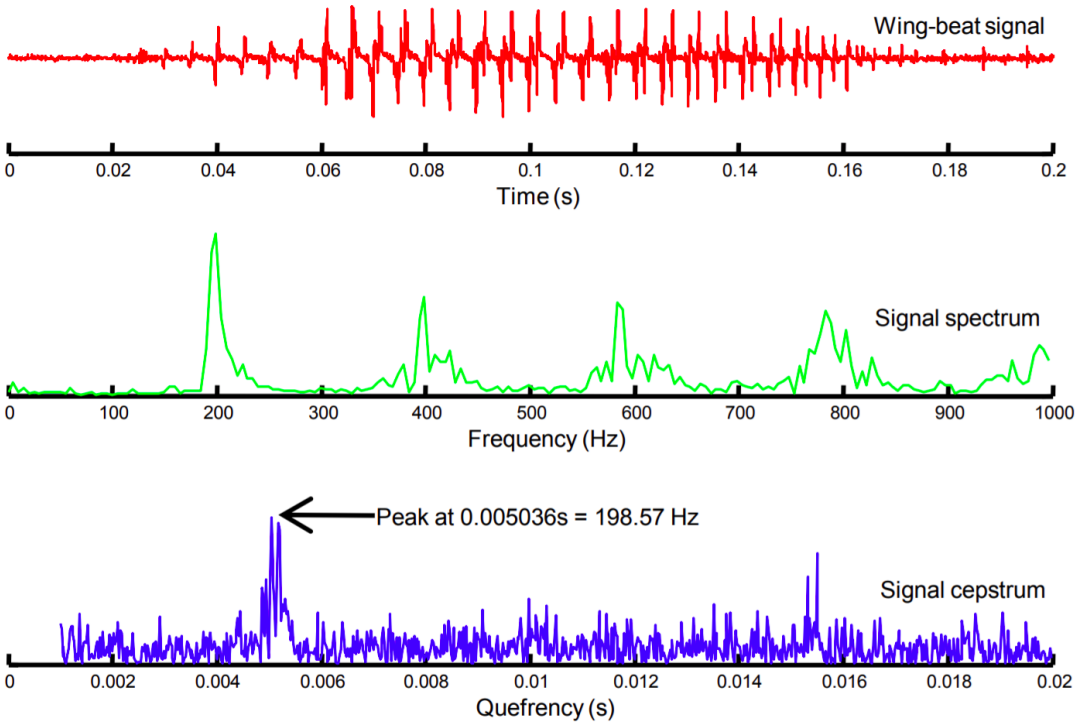
[2] Sampled at 44,100Hz.

Figure 30 – The impact of temperature on the WBF of *Aedes aegypti* female insects.



Source: Souza *et al.* (2020).

Figure 31 – An example of a *Bombus impatiens* signal captured by the optical sensor (top), the signal converted to the frequency domain (middle), and the signal cepstrum (bottom).



Source: Batista *et al.* (2011b).

2) Apply the Hanning window function to every interval.

3) Calculate the Discrete Fourier Transform (DFT) of each segment, using the Fast Fourier Transform (FFT) algorithm (NUSSBAUMER, 1981). In order to perform this step in the microcontroller, we used the FFT implementation provided by the CMSIS library.

4) Compute the squared magnitude of all FFT results.

5) Finally, combine the results of every segment by averaging them to obtain an array of energy versus frequency bins that represents the estimated spectrum of the whole signal data.

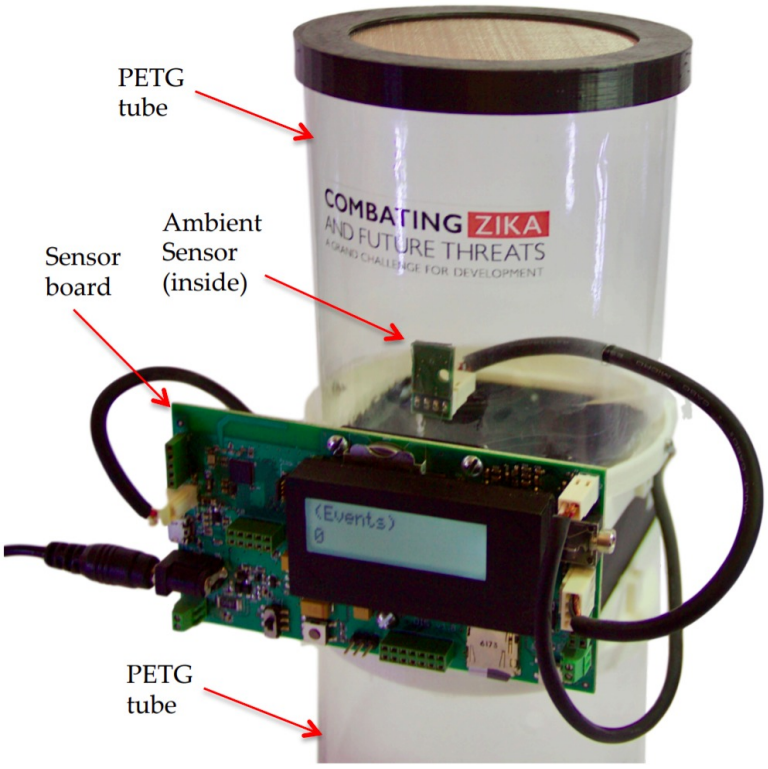
This method requires fewer processing operations than other methods because it divides the whole signal into shorter sequences. Moreover, it is an advantageous approach to generate the signal spectrum on hardware with limited memory (WELCH, 1967).

To obtain the signal cepstrum, needed in feature 3, we have to apply the Inverse Discrete Fourier Transform (IDFT) in the signal spectrum, using an adapted version of the FFT algorithm. For this step, we also employ the FFT implementation from CMSIS to produce the IDFT in the microcontroller, taking the estimated spectrum as input.

7.4 Data Collection

Souza *et al.* (2020) describe the created dataset that includes instances gathered from different species of flying insects using the developed optical sensor. This step required the development of a collector device, shown in Figure 32, that includes the ambient and optical sensors. Such apparatus was designed to easily get the correct label of the gathered data, considering that it allows having specimens of only one insect species per collector.

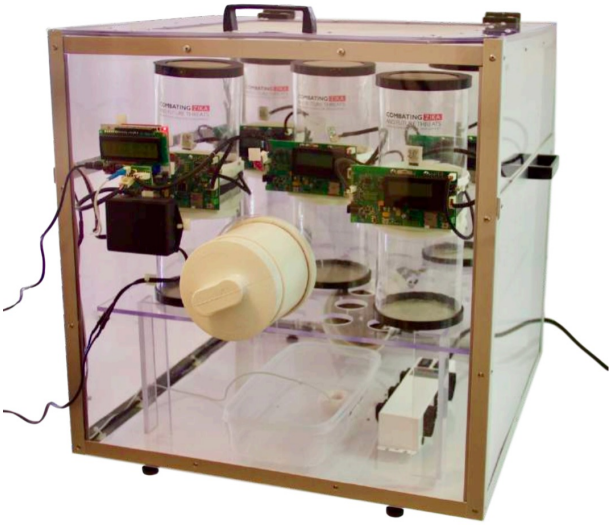
Figure 32 – The collector device produced to gather data from different flying insect species.



Source: Elaborated by Dr. Gustavo E. A. P. A. Batista.

Given that ambient conditions may affect the flight performance of insects, they collected data putting the collectors inside custom chambers (illustrated in Figure 33) that enable regulating temperature and humidity. These chambers allowed gathering data that combine different values of temperature (from 20°C to 40°C) and relative humidity (from 20% to 90%). In total, Souza *et al.* (2020) obtained approximately one million examples of 17 distinct flying insect species, including mosquitoes, houseflies, bees, and wasps. In some cases, it was possible to divide data of the same species into different sexes.

Figure 33 – Chamber projected to control ambient conditions.



Source: Elaborated by Dr. Gustavo E. A. P. A. Batista.

Finally, note that the described steps of data collection included recording the signal generated by the optical sensor as an audio file in an microSD card. Later, the signal processing operations were executed off-board, on a server computer, to construct the dataset. Therefore, implementing the data preprocessing operations in the trap microcontroller – to allow running the classifier on it – is an exclusive contribution of this dissertation.

7.5 Classifier Analysis

The experiments presented in this chapter required data from female and male *Aedes aegypti* mosquitoes. Thus, we used the *Aedes aegypti-sex* dataset presented in Chapter 6 with the set of features described in Section 7.3. This dataset, prepared by our research group, consists of a subset of the collected examples previously described and contains data that contemplate a diverse range of temperatures.

To decide which classifier to implement in the trap hardware, we first assess the accuracies of all classification models supported by EmbML. For this reason, we applied the holdout method for dividing the dataset examples into two mutually exclusive and stratified subsets: 70% for training and 30% for testing.

We start by performing a grid search to determine the best set of hyperparameters for each algorithm. In this case, we divided the training examples in 80% for creating the models and 20% to validate them. The values of hyperparameters explored in this search are presented in Table 9.

Table 9 – Searched values of hyperparameters for each classification algorithm.

Classification Models	**Hyperparameters**	**Searched Values** (initial:step:final)
J48 WEKA	pruning confidence minimum number of instances	$0.05 : 0.05 : 0.5$ $2^i, i = 0 : 1 : 7$
Logistic WEKA	ridge	$10^i, i = -10 : 1 : -4$
MultilayerPerceptron WEKA	learning rate momentum hidden layer size	$0.1 : 0.1 : 0.8$ $0.1 : 0.1 : 0.8$ $2^i, i = 2 : 1 : 7$
SMO (linear kernel) WEKA	complexity constant	$2^i, i = -4 : 1 : 4$
SMO (polynomial kernel) WEKA	complexity constant kernel exponent	$2^i, i = -4 : 1 : 4$ $2^i, i = -4 : 1 : 4$
SMO (RBF kernel) WEKA	complexity constant gamma coefficient	$2^i, i = -4 : 1 : 4$ $2^i, i = -4 : 1 : 4$
DecisionTreeClassifier scikit-learn	criterion splitter maximum depth minimum samples split	{gini, entropy} {best, random} $2^i, i = 2 : 1 : 6$ $2^i, i = 1 : 1 : 5$
LinearSVC scikit-learn	regularization parameter	$2^i, i = -4 : 1 : 4$
LogisticRegression scikit-learn	regularization parameter	$2^i, i = -4 : 1 : 4$
MLPClassifier scikit-learn	hidden layer size activation function solver	$2^i, i = 2 : 1 : 7$ {logistic, relu} {lbfgs, sgd, adam}
SVC (polynomial kernel) scikit-learn	regularization parameter kernel degree	$2^i, i = -4 : 1 : 4$ $2^i, i = -4 : 1 : 4$
SVC (RBF kernel) scikit-learn	regularization parameter gamma coefficient	$2^i, i = -4 : 1 : 4$ $2^i, i = -4 : 1 : 4$

After finding the best combination of hyperparameters for each algorithm, we built the models using the entire training set. Then, we employed EmbML to produce their C++ corresponding codes, measured their memory usage, and estimated their accuracy and classification time (per instance) on the trap microcontroller with the testing examples. Table 10 shows the accuracy results and highlights the five highest values of each column. In this table, the symbol "-" represents the cases in which the produced code did not execute because it was larger than the microcontroller's memory. We compare the accuracy rates achieved from executing the models on a desktop and the trap microcontroller, using the same representations for real numbers described in Section 6.2. Disregarding *SVC* models (with polynomial and RBF kernels) and considering only FLT and FXP32 versions, all classifiers delivered a relatively high accuracy ($\geq 96\%$), especially the *J48*, *MultilayerPerceptron*, *SMO* (RBF kernel) and *DecisionTreeClassifier* models.

In Table 11, we compare, for each model and real number representation, the mean

Table 10 – Accuracies (%) for each classification model supported by EmbML.

Classifier	Desktop	MK20DX256VLH7		
		EmbML/FLT	EmbML/FXP32	EmbML/FXP16
J48 WEKA	**98.92**	**98.92**	**98.92**	**98.57**
Logistic WEKA	97.85	97.85	97.85	50.17
MultilayerPerceptron WEKA	**98.64**	**98.65**	**98.65**	**82.14**
SMO (linear kernel) WEKA	98.35	98.35	98.35	**82.64**
SMO (polynomial kernel) WEKA	**98.67**	**98.67**	97.32	**66.89**
SMO (RBF kernel) WEKA	**98.74**	**98.74**	**98.68**	50.00
DecisionTreeClassifier scikit-learn	**98.67**	**98.67**	**98.68**	**96.63**
LinearSVC scikit-learn	97.86	97.86	**98.46**	50.00
LogisticRegression scikit-learn	98.33	98.33	96.00	50.00
MLPClassifier scikit-learn	98.51	98.57	98.44	58.74
SVC (polynomial kernel) scikit-learn	98.53	55.18	52.04	50.00
SVC (RBF kernel) scikit-learn	50.68	-	-	-

and maximum time to classify an instance. As in Chapter 6, we estimated these numbers from running a classifier ten times through the whole testing set. We also highlighted the five lowest values of each column and eliminated the row corresponding to the *SVC* (RBF kernel) classifier because the trap microcontroller was not able to execute any of its versions. Using these metrics, the best results were achieved by the *J48*, *Logistic*, *DecisionTreeClassifier*, *LinearSVC*, and *LogisticRegression* classifiers. On the other hand, the *SMO* (polynomial and RBF), *MLPClassifier*, and *SVC* (polynomial) delivered the slowest performances ($> 1\,ms$).

Next, Table 12 presents the memory usage of the evaluated classifiers. We separated the results into data memory (SRAM) and program memory (flash) to perform a more detailed analysis. As before, we highlighted the five (or more, in case of a tie) lowest values of each column. The *J48*, *Logistic*, *DecisionTreeClassifier*, *LinearSVC*, and *LogisticRegression* models have the smallest data memory consumption: at most 7% of the total SRAM capacity of the trap microcontroller. As for the program memory, the *J48*, *SMO* (linear), *DecisionTreeClassifier*, *LinearSVC*, and *LogisticRegression* models have the best performance and occupy less than 14% of its flash memory. In contrast, the *SMO* (polynomial and RBF) and *SVC* (polynomial) models are impractical to employ in our solution due to their high cost of program memory and considering that we also need memory space available to implement other methods, such as the signal processing operations.

Table 11 – Classification time (μs) for each classification model supported by EmbML.

Classifier	EmbML/FLT		EmbML/FXP32		EmbML/FXP16	
	Mean	Max.	Mean	Max.	Mean	Max.
J48 WEKA	**4.41**	**14**	**1.26**	**6**	**1.17**	**6**
Logistic WEKA	**159.56**	**170**	**47.09**	**56**	36.14	47
MultilayerPerceptron WEKA	947.01	971	218.38	227	138.68	148
SMO (linear kernel) WEKA	288.51	298	99.96	109	**26.69**	**36**
SMO (polynomial kernel) WEKA	66,328.53	66,449	13,300.14	13,322	12,735.14	13,079
SMO (RBF kernel) WEKA	172,527.96	173,693	25,459.76	25,469	19,657.88	19,825
DecisionTreeClassifier scikit-learn	**7.75**	**16**	**1.81**	**7**	**1.74**	**7**
LinearSVC scikit-learn	**124.72**	**134**	**20.26**	**29**	**19.72**	**29**
LogisticRegression scikit-learn	**124.74**	**133**	**20.29**	**29**	**20.07**	**29**
MLPClassifier scikit-learn	8,479.30	8,668	1,295.58	1,303	1,344.36	1,360
SVC (polynomial kernel) scikit-learn	78,454.07	89,210	17,905.48	17,918	2,681.04	2,686

By analyzing these three metrics, we decided to implement the *J48* (FXP32) in our solution because it achieves the best accuracy rate, the second-best time performance, and demands a relatively low memory consumption.

Finally, as a further step supported by EmbML, we assessed different fixed-point formats with 32 bits for the selected model. Basically, we explored different values (from 2 to 30) to use as the number of bits in the fractional part, and evaluated the effects on time and accuracy using the testing instances – since memory usage has not modified. Figure 34 presents the results of accuracy for several sizes of the fractional part. We can observe that the values are close to 100%, from 2 to 20 bits, and equal to 50% after 22 bits, indicating loss of precision. As for the classification time, Figure 35 shows the maximum and mean values for each format. First, we note that the results are very similar. But, considering only the numbers of bits in the fractional part that maintain high accuracy (*i.e.*, from 2 to 20), we see that using 2 bits achieves the lowest value for maximum classification time. Therefore, we decided to implement the J48 model with the $Q30.2$ fixed-point format in the trap microcontroller.

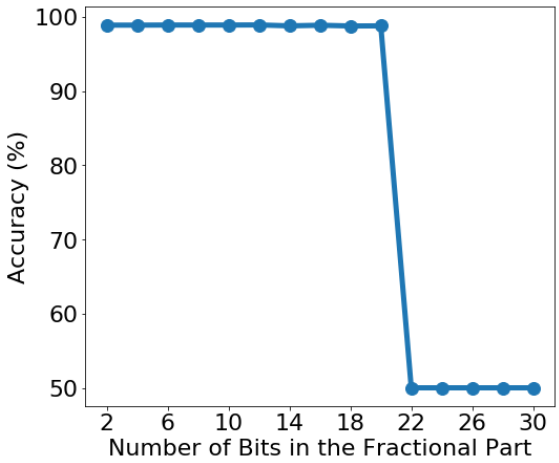
7.6 Experiments With Collectors

Our first practical evaluation of the classifier performance using *Aedes aegypti* mosquitoes included using two collectors (shown in Figure 36): one containing five female mosquitoes, and the other having five male mosquitoes. The main focus of this experiment was to validate the

Table 12 – Memory consumption (*kB*) for each classification model supported by EmbML.

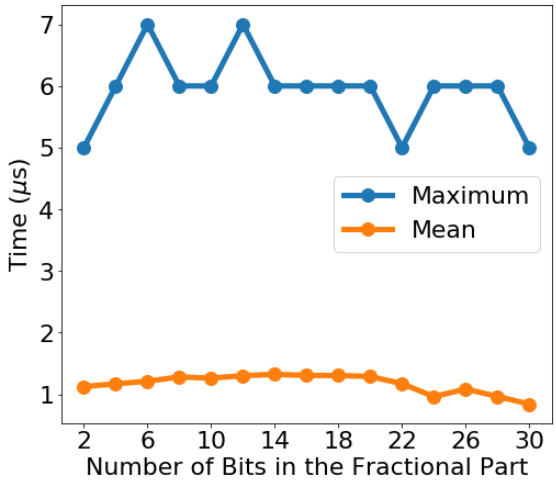
Classifier	EmbML/FLT		EmbML/FXP32		EmbML/FXP16	
	Data Memory	Program Memory	Data Memory	Program Memory	Data Memory	Program Memory
J48 WEKA	**4.20**	**32.10**	**4.20**	**32.60**	**4.13**	**32.60**
Logistic WEKA	**4.21**	34.24	**4.20**	34.09	**4.13**	33.13
MultilayerPerceptron WEKA	4.39	35.33	4.39	35.58	4.23	33.87
SMO (linear kernel) WEKA	4.22	**33.66**	4.22	**33.98**	4.15	**32.88**
SMO (polynomial kernel) WEKA	4.23	149.21	4.23	149.65	4.16	90.88
SMO (RBF kernel) WEKA	4.24	204.77	4.23	204.42	4.16	118.27
DecisionTreeClassifier scikit-learn	**4.20**	**32.91**	**4.20**	**33.04**	**4.13**	**33.10**
LinearSVC scikit-learn	**4.20**	**33.39**	**4.20**	**32.84**	**4.13**	**32.69**
LogisticRegression scikit-learn	**4.20**	**33.39**	**4.20**	**32.84**	**4.13**	**32.69**
MLPClassifier scikit-learn	4.86	52.49	4.86	51.80	4.47	42.30
SVC (polynomial kernel) scikit-learn	**4.21**	250.09	4.21	249.66	4.14	142.64

Figure 34 – Accuracy comparison.



Source: Elaborated by the author.

Figure 35 – Classification time comparison.



Source: Elaborated by the author.

classifier combined with the optical sensor before implementing it on the trap. We also wanted to verify the impact of a possible loss of precision, considering that we trained the model using a dataset in which the feature values were obtained from processing the recorded signals in a server computer. Thus, for this phase, we programmed the microcontroller for collecting the data, preprocessing the input signal to produce the predictive features, classifying the event, and storing the results in the microSD card.

Figure 36 – Collectors used in the experiment. The left one contains five female *Aedes aegypti* mosquitoes and the right one contains five males.

Source: Elaborated by the author.

We ran this experiment for a total of three rounds. They happened on different consecutive days, starting at dusk and ending after approximately 24 hours. At the end of a round, we replaced all mosquitoes in the both collectors (given that some could have died), saved the produced data (*e.g.*, recorded signals, counted events and classification results), cleaned microSD data, and restarted the microcontrollers. Specialists helped us by providing adult *Aedes aegypti* mosquitoes bred on appropriate laboratory-controlled conditions and already separated into male and female. We also performed the rounds in a closed room with a heater and humidifier to keep proper ambient conditions (around 28°C and 50% relative humidity) that preserve the mosquitoes during the experiments.

Table 13 and Table 14 present the results gathered during this experiment from the collectors with only female mosquitoes and with only males, respectively. In these tables, we expose, for each round, the number of events classified as female and male, and the rate of corrected classified events – since we know the true label of every instance. We also present other metrics obtained from the recorded data: the mean temperature and relative humidity during the events and their standard deviation (SD); and the mean and maximum time for (only) classifying an event.

Table 13 – Results gathered from the collector containing only female *Aedes aegypti* mosquitoes.

	Counts			Classification Time (μs)		Temperature (°C)		Humidity (%)	
	Female	**Male**	**Rate (%)**	**Mean**	**Max.**	**Mean**	**SD**	**Mean**	**SD**
Day 1	71	3	95.95	1.43	2	29.55	0.25	56.94	6.10
Day 2	353	17	95.41	1.46	3	28.74	0.34	55.61	2.43
Day 3	322	14	95.83	1.45	2	31.69	0.79	42.35	3.03

Table 14 – Results gathered from the collector containing only male *Aedes aegypti* mosquitoes.

	Counts			Classification Time (μs)		Temperature (°C)		Humidity (%)	
	Female	**Male**	**Rate (%)**	**Mean**	**Max.**	**Mean**	**SD**	**Mean**	**SD**
Day 1	3	145	97.97	1.82	2	28.98	0.41	52.91	2.34
Day 2	3	411	99.28	1.76	2	28.63	0.34	59.98	3.19
Day 3	3	183	98.39	1.76	2	30.82	1.11	49.42	3.02

Interpreting the results from Table 13 and Table 14, we can notice that the embedded model was able to maintain its classification efficiency estimated on test, with rates of correctly labeled events of over 95%. Therefore, training with off-board preprocessed data seems to cause an inexpressive loss in these results. Also, we had little variation in temperature and humidity. But, as we previously stated, it was intentional because we wanted to keep the mosquitoes alive (on friendly ambient conditions) to be able to continuously gather data during each round. Finally, we mention that the mean and maximum classification times are relatively close to those estimated using the testing set in the previous section.

Now, we examine the overall time, *i.e.*, the time interval between the moment when the mosquito started crossing the optical sensor light and when the classifier returned the predicted label. Note that this value includes the time spent to preprocess the data and record it. Table 15 and Table 16 show – respectively for the female and male collectors – the mean and maximum values for this metric and the recorded signal length.

Table 15 – Time results collected from the female *Aedes aegypti* mosquitoes.

	Overall Time (*ms*)		Signal Length (*ms*)		
	Mean	**Max.**	**Mean**	**Max.**	**SD**
Day 1	211.30	528	160.37	462.54	60.28
Day 2	230.10	601	173.39	532.20	56.87
Day 3	217.03	504	161.18	445.12	49.72

Table 16 – Time results collected from the male *Aedes aegypti* mosquitoes.

	Overall Time (*ms*)		Signal Length (*ms*)		
	Mean	**Max.**	**Mean**	**Max.**	**SD**
Day 1	179.66	307	129.50	253.56	23.62
Day 2	185.64	368	131.52	311.61	25.53
Day 3	175.53	323	125.48	270.98	16.63

Let us assume that the signal length is equal to the amount of time that the mosquito stayed occluding the optical sensor light. Thus, the difference between the overall time and signal

length shall be equivalent to the microcontroller delay in producing a label after the mosquito flew away from the sensor light. In these results, the highest difference of their mean values is 56.71 *ms* and occurred on day 2 of the female mosquito data. If we had decided to use the SMO (polynomial kernel) classifier with FLT representation, for instance, this amount of time would probably have been twice as much.

7.7 Experiments With The Trap

The last experiment with *Aedes aegypti* mosquitoes consists of using the developed trap in a laboratory simulated ambient. The focus of this phase is to investigate the consequences of embedding the classifier on the intelligent trap to actually accomplish its main objective: selectively capture the desired class(es) of flying insects. Therefore, it required the construction of a cage, illustrated in Figure 37 and Figure 38, that allows releasing only female and male mosquitoes inside and analyzing the trap performance in capturing or expelling them. The designed cage has approximately $1.8\,m \times 1.8\,m \times 1.8\,m$ ($length \times width \times height$) dimensions and includes a double protection of mosquito netting fabric connected to a plastic pipe structure that creates an internal space isolated from external insects. We placed this cage in the same room used for the previous experiment – with controlled ambient conditions.

Figure 37 – Front-view of the cage.

Source: Elaborated by the author.

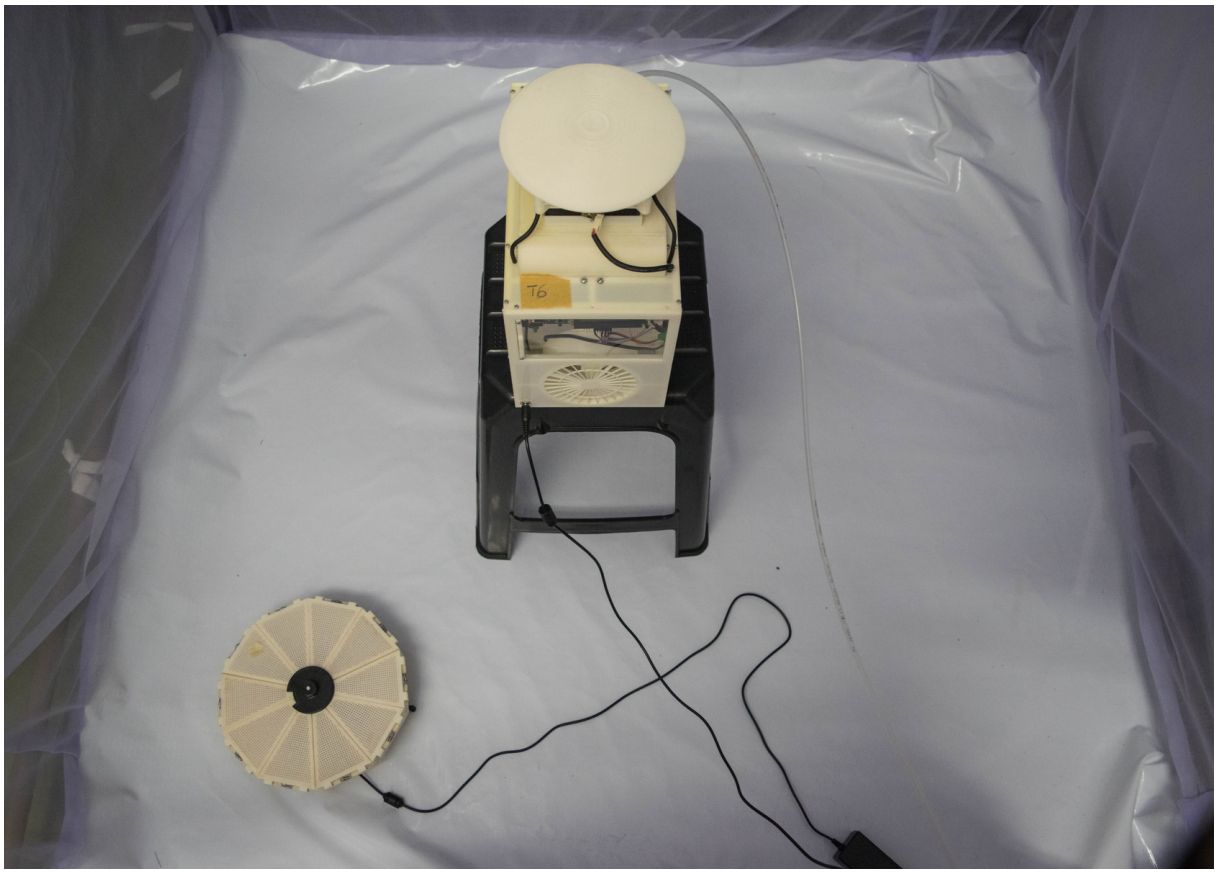
Figure 38 – Side-view of the cage.

Source: Elaborated by the author.

To prepare the setup, we placed the trap in the center of the cage, on top of a plastic stool, keeping it approximately 45 *cm* from the ground. During the experiment, we also had a mosquito release device inside the cage, as shown in Figure 39. This device, shown in Figure 40 and

Figure 41, contains 10 separated compartments and automatically opens one of them, releasing the mosquitoes inside it, after each hour completed. This method is useful for distributing the mosquitoes over time and preventing them from going directly and simultaneously to the trap.

Figure 39 – Arrangement of the trap and the release device inside the cage.



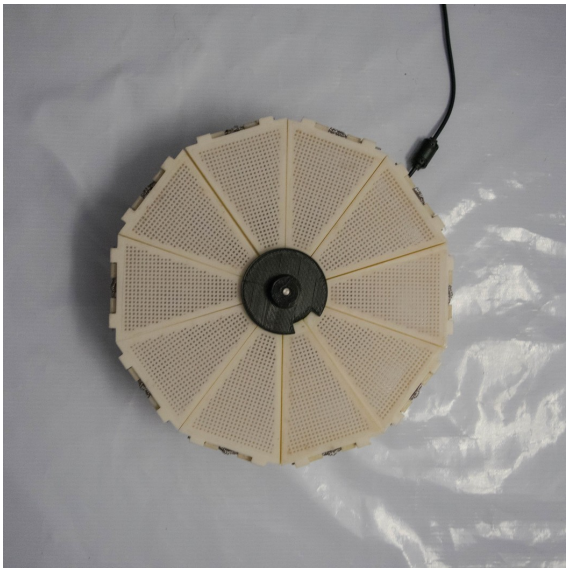
Source: Elaborated by the author.

In order to attract the mosquitoes to the trap, we chose carbon dioxide (CO2) since it is an attractant to many mosquito biting species (KLINE *et al.*, 1990). We used a pipe connected to a CO2 cylinder (placed outside the cage) to conduct the gas to the trap entrance, near the optical sensor.

As in the previous experiment, we performed this experiment in three rounds of approximately 24 hours each, distributed in consecutive days, and starting at dusk. For each round, we used 30 mosquitoes (15 females and 15 males) and divided them on the first three boxes of the release device, putting five females and five males in each of these containers. Therefore, after three hours from the round beginning, we should have 30 mosquitoes either flying freely in the cage or captured by the trap.

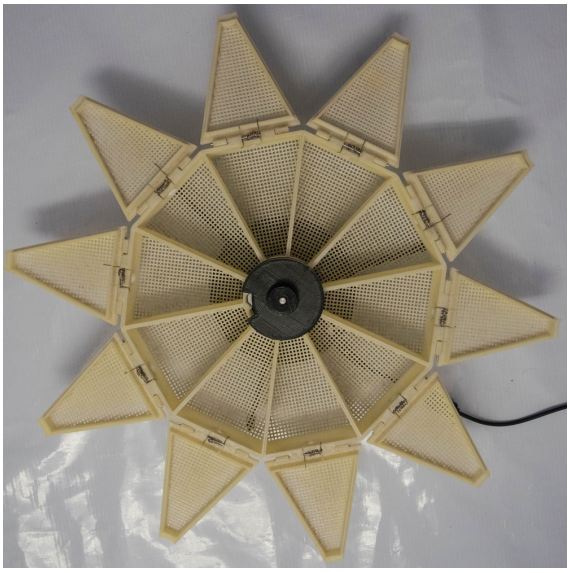
At the beginning of a round, we inserted the mosquitoes in the release device, prepared the apparatus on their positions, and started them. At the end of every round, we performed the following tasks: stopped the trap execution; manually captured, counted, and classified the insects

Figure 40 – Mosquito release device totally closed.



Source: Elaborated by the author.

Figure 41 – Mosquito release device totally open.



Source: Elaborated by the author.

outside and inside the trap; copied the produced data from the events; cleaned the microSD card; and reset the trap microcontroller. A specialist also assisted us in the process of manually capturing and classifying the mosquitoes.

The trap microcontroller was programmed using a modified version of the previous experiment firmware. The updated version includes all the previous tasks – collecting, preprocessing, classifying and storing the input data – and additionally can activate the trap fan, after getting the predicted label, to capture a female mosquito or expel a male. Once the mosquito is inside the trap, it is very unlikely to escape.

Table 17 exhibits the results obtained at the end of each round of this experiment. For both classes, we divided the numbers of counted mosquitoes inside and outside the trap. It also includes other relevant information: the number of events classified as female, *i.e.*, in which the trap activated the fan to capture a mosquito; the real value of counted mosquitoes inside the cage; and the total number of events registered by the trap during the rounds.

Table 17 – Results from the trap experiment.

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

These results reveal that the trap was very effective in capturing female *Aedes aegypti* mosquitoes. In the three rounds, it could catch all the released female mosquitoes. On the other hand, it also wrongly captured at least 20% of the males. This value is quite higher than

expected from previous results (Section 7.6), in which it was possible to classify correctly over 95% of the events. One explanation is that, when capturing a female mosquito, the trap could have caught some nearby males, since males are attracted to the sound produced by female mosquitoes (BELTON; COSTELLO, 1979; BELTON, 1994). However, the fact that the total number of mosquitoes inside the trap was relatively close to the number of events classified as female suggests that the trap capture mechanism is considerably robust and it was able to catch approximately only one mosquito at a time.

Another hypothesis consists of observing that the trap mechanism to expel mosquitoes is strong enough to kill them after some attempts. Therefore, we consider the possibility that it may cause injury to the expelled mosquitoes and affect their flight behavior. Another evidence of this thought is that a male *Aedes aegypti* usually has a higher WBF than a female. Assuming a mosquito gets injured from the expelling movement, its WBF may change to a lower value, which directly affects the classification process.

In Table 18, we show the mean and SD values for temperature and relative humidity gathered during the events of a round. As desired, we were able to maintain these conditions at an adequate level to preserve mosquito activity, close to 28°C and 50%, respectively.

Table 18 – Temperature and relative humidity values gathered by the trap.

	Temperature (°C)		**Humidity (%)**	
	Mean	**SD**	**Mean**	**SD**
Day 1	28.37	0.63	53.23	3.71
Day 2	29.95	0.95	49.50	3.72
Day 3	30.88	0.99	45.27	3.56

Finally, we present in Table 19 the time results of these rounds. First, we provide the values for the time consumed in the classification step. Observe that they are consistent with the results estimated in Section 7.5 and collected in Section 7.6. Next, in the table, we show the overall time and recorded signal length. Considering the same evaluation performed in the previous section, the highest difference between their mean values is 48.47 *ms* and occurred on day 3. The results from Table 17 suggest that this value is acceptable for our application. Yet, we still lack an analytical evaluation that demonstrates it.

Table 19 – Time results collected in the experiment with the trap.

	Classification Time (μs)		**Overall Time (*ms*)**		**Signal Length (*ms*)**		
	Mean	**Max.**	**Mean**	**Max.**	**Mean**	**Max.**	**SD**
Day 1	1.61	3	201.79	410	153.94	358.05	46.63
Day 2	1.56	2	197.38	333	150.09	282.59	43.57
Day 3	1.47	2	196.03	334	147.56	282.59	38.04

7.8 Limitations

The experiments presented in this chapter achieve successful results, but there are a few limitations to our analysis. First of all, three rounds are not an ideal number to get a realistic estimation of the trap performance. However, the main restrictive factors are mosquito availability and their life span. Breeding mosquitoes in the laboratory is a laborious process that demands the manual effort of specialists, has to follow a protocol to keep proper conditions, and it takes several days for them to reach adult age. Other factors, such as the seasons, also have a significant impact on this process. Nevertheless, we were able to get a considerable number of mosquitoes that allowed us to repeat different experiments and analyze the classifier performance in more realistic scenarios.

Besides, this study lacks defining real-time requirements and proving the solution meets them. The main reasons for it are the additional steps involved to carry out such evaluation since we should estimate the flying speed of the target insects and calculate the amount of time they take to escape the trap's reach. Then, we would be able to modify the trap design or its firmware, considering these steps and the total time it takes to produce a classification result. As we used a previously designed trap, we had restrictions in physically adjusting it to meet those requirements. Also, we understand that it is a cyclical process, and the results presented in this chapter will enable to adapt the trap and its software components for future work.

At final, the experiments have not explored different values of temperature, which is a variable with strong potential to affect the results. However, previously in this chapter, we presented the reasons that guided our chosen approach.

7.9 Final Considerations

In this chapter, we considered a real-world application to employ the EmbML in the process of developing and evaluating classifiers to it. First, we described the problem and some of the previous work involved in proposing an intelligent trap for selectively capture flying insects. Then, we investigated the classifiers supported by EmbML to solve the stated problem and constructed an extensive analysis that explored all the provided options. This step helped to get a better sense of their performance and be able to choose the most appropriate alternative. After embedding the classifier into the trap microcontroller, we evaluated the projected solution in two realistic situations. They consisted of experiments that employed the developed devices to classify and capture *Aedes aegypti* mosquitoes automatically, according to their sex. Finally, we presented the achieved results and discussed their limitations.

CHAPTER

8

CONCLUSION

8.1 Initial Considerations

In this chapter, we review some of the subjects presented throughout this dissertation. It starts highlighting its achieved objectives and contributions to the ML community. From a general perspective, we examine how it may positively impact the development of future technological projects. Particularly, we also consider its importance in completing the pipeline of the project that has been developed by our research group in the past years. Lastly, we discuss the limitations of this study and offer a few suggestions for extending the presented work.

8.2 Dissertation Review

The main objective of the work presented in this dissertation is to introduce an alternative solution to facilitate the process of using classification models in low-power microcontrollers. To achieve this goal, we proposed a tool named EmbML, which takes as input a model trained in popular ML frameworks. The output of our solution is a classifier source code specifically designed to execute in low-power microcontrollers, allowing modifications in its implementations such as an optimized usage of the data memory and support to a fixed-point format for real number operations.

Compared to the existing related tools described in Chapter 2, EmbML offers a few advantages, such as:

- it is an open-source solution;
- it takes models trained with popular ML frameworks as input, which simplify this process;
- it supports a variety of classification models representing different learning paradigms;

- and it generates classifier codes specifically designed to execute in low-power hardware.

Overall, EmbML consists of a robust solution for evaluating and employing classifiers to resource-constrained hardware. Thus, we understand that it is a valuable contribution, especially for offering support aimed at the development of technological applications, as those presented in Section 2.3.

Besides introducing this solution, this work also developed a comparative analysis of EmbML classifiers in Chapter 6. We evaluated them using different metrics (accuracy, classification time, and memory consumption), benchmark datasets, and microcontrollers. This step provided a better comprehension of the behavior of the classifiers in diverse situations and the benefits achieved by the provided modifications. In Section 6.5, we compared classifiers from EmbML and some related tools. In at least 70% of the cases, our classifiers achieved the best time and memory results, demonstrating their efficiency.

In Chapter 7, we considered an application to demonstrate the complete EmbML pipeline: from training and choosing the ML model, to deploying and assessing it in a microcontroller. The evaluated case consisted of a research project aiming to produce an intelligent trap that selectively captures flying insects using ML models. With the achievements of this dissertation, we were able to leverage the trap development process to a stage in which it can now operate effectively in a realistic scenario, as presented in Section 7.7.

8.3 Limitations

In this section, we complement the discussion of Section 7.8, presenting the following limitations of this work:

1) In the comparative analysis from Chapter 6, we examined the EmbML classifiers using two families of microcontrollers: ARM and AVR. Although they are representative options, we still need to certify that these classifiers obtain comparable results in other popular chip families, such as PIC and 8051.

2) When comparing classifier performance in Section 6.5, we examined models from related tools that have a corresponding in EmbML. The main intention of this strategy was to perform a fair comparison since we used the same trained model in the different tools to generate the compared classifier codes. Yet, this approach led us to ignore some robust solutions presented in Section 2.2, such as EdgeML and FANN-on-MCU. As a result, we are not able to claim that our solution outperforms the state-of-the-art, but it produces competitive results compared to the options available.

3) In the experiments from Chapter 6, we defined two versions of fixed-point formats (FXP32 and FXP16). As shown in Section 7.5, we could have explored many other formats with 32 or 16 bits, but we chose to employ only two of them to report concise results.

4) For our experiments in Chapter 7, we considered an ambient with controlled values of temperature and humidity. Furthermore, we exposed the trap only to insect classes that belong to the training set. These situations are different from a natural setting, in which ambient conditions fluctuate regularly, the classifier may confront examples of unknown labels, and the correct label of each event is not available, especially for expelled insects. Also, the insect population may be small in an open environment compared to laboratory experiments, which implies a fewer number of events and less confidence in the results.

8.4 Future Work

Finally, we describe some topics that can be explored as extensions of this work:

1) Deep NNs are currently popular algorithms for classification problems due to their outstanding performance. Since they usually produce large models, using them on low-power microcontrollers is impractical. However, we understand it would be interesting to use the experience with EmbML to investigate techniques (HAN; MAO; DALLY, 2015; HOWARD *et al.*, 2017; ZHANG *et al.*, 2018) that adapt these models to execute in embedded hardware with support for graphical operations – such as NVIDIA Jetson Series, Intel Movidius Neural Compute Stick, and Google Coral.

2) Most of the work involved in evaluating and testing the EmbML classifiers in microcontrollers still depend on manual efforts. For this reason, adjusting EmbML to generate a complete report of the classifier analysis automatically is a natural next step.

3) EmbML could directly interact with scikit-learn and WEKA to facilitate the pipeline of using it. One option consists of allowing EmbML to call the functions that train the classification models. Therefore, the process would be simplified since the user would need to deal with only one tool to produce a classifier source code.

4) In Section 7.6, we estimated that the trap microcontroller took around 56.71 ms to produce a label for an event. Since the deployed classifier took only 3 μs to process an example in the worst case, we assume that the most time-consuming task is preprocessing the input data. Consequently, optimizing it is a critical step to allow reacting faster to events and meeting the real-time requirements of the problem.

5) In the current approach, the feature preprocessing step only begins after the insect stopped occluding the optical sensor light. Future studies should address the impact of limiting the length of the signal to start preprocessing it. In other words, we should investigate if a

smaller sample of the input signal would be enough to classify it correctly. Hence, this method might reduce the overall processing time and define an upper bound to the latency for reacting to an event.

BIBLIOGRAPHY

ABADI, M.; BARHAM, P.; CHEN, J.; CHEN, Z.; DAVIS, A.; DEAN, J.; DEVIN, M.; GHEMAWAT, S.; IRVING, G.; ISARD, M. *et al.* Tensorflow: A system for large-scale machine learning. In: **12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)**. [S.l.: s.n.], 2016. p. 265–283. Citation on page 35.

ALIMOGLU, F.; ALPAYDIN, E. Methods of combining multiple classifiers based on different representations for pen-based handwritten digit recognition. In: **TAINN**. [S.l.: s.n.], 1996. Citation on page 87.

ALIPPI, C.; PELOSI, G.; ROVERI, M. Computational intelligence techniques to detect toxic gas presence. In: IEEE. **Computational Intelligence for Measurement Systems and Applications, Proceedings of 2006 IEEE International Conference on**. [S.l.], 2006. p. 40–44. Citation on page 39.

ANGUITA, D.; GHIO, A.; ONETO, L.; PARRA, X.; REYES-ORTIZ, J. A public domain dataset for human activity recognition using smartphones. In: CIACO. **21th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN)**. [S.l.], 2013. p. 437–442. Citation on page 87.

ATMEL CORPORATION. **Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V DATASHEET**. [S.l.], 2014. Rev. 2549Q-02/2014. Citation on page 88.

____. **Atmel SAM3X / SAM3A Series DATASHEET**. [S.l.], 20145. Rev. 11057C. Citation on page 88.

____. **Atmel ATmega328/P DATASHEET**. [S.l.], 2016. Rev. B. Citation on page 88.

ATZORI, L.; IERA, A.; MORABITO, G. The internet of things: A survey. **Computer networks**, v. 54, n. 15, p. 2787–2805, 2010. Citation on page 28.

BALDI, P. F.; HORNIK, K. Learning in linear neural networks: A survey. **IEEE Transactions on neural networks**, IEEE, v. 6, n. 4, p. 837–858, 1995. Citation on page 46.

BATISTA, G. E.; KEOGH, E. J.; MAFRA-NETO, A.; ROWTON, E. Sigkdd demo: sensors and software to allow computational entomology, an emerging application of data mining. In: ACM. **Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining**. [S.l.], 2011. p. 761–764. Citations on pages 27, 28, 30, 41, 109, 110, and 111.

BATISTA, G. E. A. P. A.; HAO, Y.; KEOGH, E.; MAFRA-NETO, A. Towards automatic classification on flying insects using inexpensive sensors. In: IEEE. **2011 10th International Conference on Machine Learning and Applications and Workshops**. [S.l.], 2011. v. 1, p. 364–369. Citations on pages 109, 110, 112, 113, 114, and 115.

BELTON, P. Attractton of male mosquitoes to sound. **J Am Mosq Control Assoc**, v. 10, p. 297–301, 1994. Citation on page 127.

BELTON, P.; COSTELLO, R. A. Flight sounds of the females of some mosquitoes of western canada. **Entomologia experimentalis et applicata**, Wiley Online Library, v. 26, n. 1, p. 105–114, 1979. Citation on page 127.

BERTHOLD, M. R.; CEBRON, N.; DILL, F.; GABRIEL, T. R.; oTTER, T. K.; MEINL, T.; OHL, P.; SIEB, C.; THIEL, K.; WISWEDEL, B. KNIME: The Konstanz Information Miner. In: **Studies in Classification, Data Analysis, and Knowledge Organization**. [S.l.: s.n.], 2007. Citation on page 51.

BIFET, A.; HOLMES, G.; KIRKBY, R.; PFAHRINGER, B. MOA: massive online analysis. **Journal of Machine Learning Research**, v. 11, p. 1601–1604, 2010. Citation on page 51.

BISHOP, C. **Pattern Recognition and Machine Learning**. [S.l.]: Springer-Verlag New York, 2006. Citation on page 48.

BOJARSKI, M.; TESTA, D. D.; DWORAKOWSKI, D.; FIRNER, B.; FLEPP, B.; GOYAL, P.; JACKEL, L. D.; MONFORT, M.; MULLER, U.; ZHANG, J. *et al.* End to end learning for self-driving cars. **arXiv preprint arXiv:1604.07316**, 2016. Citation on page 27.

BOTTA, A.; DONATO, W. D.; PERSICO, V.; PESCAPÉ, A. Integration of cloud computing and internet of things: a survey. **Future generation computer systems**, Elsevier, v. 56, p. 684–700, 2016. Citation on page 28.

BRADLEY, S. P.; HAX, A. C.; MAGNANTI, T. L. **Applied mathematical programming**. [S.l.]: Addison-Wesley, 1977. Citation on page 60.

BREIMAN, L.; FRIEDMAN, J. H.; OLSHEN, R. A.; STONE, C. J. **Classification and regression trees**. [S.l.]: Routledge, 1984. Citation on page 45.

CHANG, C.-C.; LIN, C.-J. LIBSVM: A library for support vector machines. **ACM Transactions on Intelligent Systems and Technology**, v. 2, p. 27:1–27:27, 2011. Citations on pages 34 and 79.

CHELLI, A.; PÄTZOLD, M. A machine learning approach for fall detection and daily living activity recognition. **IEEE Access**, IEEE, v. 7, p. 38670–38687, 2019. Citation on page 27.

CHEN, J.; KWONG, K.; CHANG, D.; LUK, J.; BAJCSY, R. Wearable sensors for reliable fall detection. In: IEEE. **2005 IEEE Engineering in Medicine and Biology 27th Annual Conference**. [S.l.], 2006. p. 3551–3554. Citation on page 27.

CHEN, Y.; WHY, A.; BATISTA, G.; MAFRA-NETO, A.; KEOGH, E. Flying insect classification with inexpensive sensors. **Journal of insect behavior**, Springer, v. 27, n. 5, p. 657–677, 2014. Citation on page 109.

CHOI, Y.; RALHAN, A.; KO, S. A study on machine learning algorithms for fall detection and movement classification. In: IEEE. **2011 International Conference on Information Science and Applications**. [S.l.], 2011. p. 1–8. Citation on page 27.

COOPERSMITH, E. J.; MINSKER, B. S.; WENZEL, C. E.; GILMORE, B. J. Machine learning assessments of soil drying for agricultural planning. **Computers and electronics in agriculture**, v. 104, p. 93–104, 2014. Citation on page 27.

CORTES, C.; VAPNIK, V. Support-vector networks. **Machine learning**, Springer, v. 20, n. 3, p. 273–297, 1995. Citations on pages 48 and 49.

DENNIS, D.; PABBARAJU, C.; SIMHADRI, H. V.; JAIN, P. Multiple instance learning for efficient sequential data classification on resource-constrained devices. In: **Advances in Neural Information Processing Systems**. [S.l.: s.n.], 2018. p. 10953–10964. Citation on page 36.

DIETTERICH, T. G. Ensemble methods in machine learning. In: SPRINGER. **International workshop on multiple classifier systems**. [S.l.], 2000. p. 1–15. Citation on page 44.

DOMINGOS, P. A few useful things to know about machine learning. **Communications of the ACM**, ACM New York, NY, USA, v. 55, n. 10, p. 78–87, 2012. Citation on page 44.

FAROOQ, U.; AMAR, M.; HAQ, E. ul; ASAD, M. U.; ATIQ, H. M. Microcontroller based neural network controlled low cost autonomous vehicle. In: **ICMLC**. [S.l.: s.n.], 2010. p. 96–100. Citations on pages 29, 38, and 41.

FERNANDES, L. C.; SOUZA, J. R.; PESSIN, G.; SHINZATO, P. Y.; SALES, D.; MENDES, C.; PRADO, M.; KLASER, R.; MAGALHAES, A. C.; HATA, A. *et al.* Carina intelligent robotic car: architectural design and applications. **Journal of Systems Architecture**, Elsevier, v. 60, n. 4, p. 372–392, 2014. Citation on page 27.

FERNÁNDEZ-DELGADO, M.; CERNADAS, E.; BARRO, S.; AMORIM, D. Do we need hundreds of classifiers to solve real world classification problems? **The Journal of Machine Learning Research**, v. 15, n. 1, p. 3133–3181, 2014. Citation on page 30.

FREESCALE SEMICONDUCTOR, INC. **K20 Sub-Family Data Sheet**. [S.l.], 2012. Rev. 3. Citation on page 88.

____. **K20 Sub-Family Reference Manual**. [S.l.], 2012. Rev. 1.1. Citation on page 88.

GOPINATH, S.; GHANATHE, N.; SESHADRI, V.; SHARMA, R. Compiling kb-sized machine learning models to tiny iot devices. In: **Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation**. [S.l.: s.n.], 2019. p. 79–95. Citations on pages 36 and 57.

GUPTA, C.; SUGGALA, A. S.; GOYAL, A.; SIMHADRI, H. V.; PARANJAPE, B.; KUMAR, A.; GOYAL, S.; UDUPA, R.; VARMA, M.; JAIN, P. Protonn: Compressed and accurate knn for resource-scarce devices. In: JMLR. ORG. **Proceedings of the 34th International Conference on Machine Learning-Volume 70**. [S.l.], 2017. p. 1331–1340. Citations on pages 36 and 39.

HALL, M.; FRANK, E.; HOLMES, G.; PFAHRINGER, B.; REUTEMANN, P.; WITTEN, I. H. The weka data mining software: an update. **ACM SIGKDD explorations newsletter**, v. 11, n. 1, p. 10–18, 2009. Citations on pages 29 and 51.

HAN, S.; MAO, H.; DALLY, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. **arXiv preprint arXiv:1510.00149**, 2015. Citation on page 131.

HANKE, M.; HALCHENKO, Y. O.; SEDERBERG, P. B.; HANSON, S. J.; HAXBY, J. V.; POLLMANN, S. Pymvpa: A python toolbox for multivariate pattern analysis of fmri data. **Neuroinformatics**, Springer, v. 7, n. 1, p. 37–53, 2009. Citation on page 51.

HARRIS, A. F.; NIMMO, D.; MCKEMEY, A. R.; KELLY, N.; SCAIFE, S.; DONNELLY, C. A.; BEECH, C.; PETRIE, W. D.; ALPHEY, L. Field performance of engineered male mosquitoes. **Nature biotechnology**, Nature Publishing Group, v. 29, n. 11, p. 1034–1037, 2011. Citation on page 110.

HAYKIN, S. S.; HAYKIN, S. S.; HAYKIN, S. S.; HAYKIN, S. S. **Neural networks and learning machines**. [S.l.]: Pearson Upper Saddle River, 2009. Citation on page 47.

HECHT-NIELSEN, R. Theory of the backpropagation neural network. In: **Neural networks for perception**. [S.l.]: Elsevier, 1992. p. 65–93. Citation on page 47.

HOFMANN, M.; KLINKENBERG, R. **RapidMiner: Data mining use cases and business analytics applications**. [S.l.]: CRC Press, 2013. Citation on page 51.

HOWARD, A. G.; ZHU, M.; CHEN, B.; KALENICHENKO, D.; WANG, W.; WEYAND, T.; ANDREETTO, M.; ADAM, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. **arXiv preprint arXiv:1704.04861**, 2017. Citation on page 131.

HWANG, M.-C.; KIM, N.-h.; PARK, C.-S.; KO, S.-J. *et al.* Person identification system for future digital tv with intelligence. **IEEE Transactions on Consumer Electronics**, IEEE, v. 53, n. 1, 2007. Citation on page 39.

IRANI, J.; PISE, N.; PHATAK, M. Clustering techniques and the similarity measures used in clustering: A survey. **International Journal of Computer Applications**, Citeseer, v. 134, n. 7, p. 9–14, 2016. Citation on page 44.

JORDAN, M. I.; MITCHELL, T. M. Machine learning: Trends, perspectives, and prospects. **Science**, American Association for the Advancement of Science, v. 349, n. 6245, p. 255–260, 2015. Citation on page 44.

KAHAN, W. Ieee standard 754 for binary floating-point arithmetic. **Lecture Notes on the Status of IEEE**, v. 754, n. 94720-1776, p. 11, 1996. Citation on page 86.

KARANTONIS, D. M.; NARAYANAN, M. R.; MATHIE, M.; LOVELL, N. H.; CELLER, B. G. Implementation of a real-time human movement classifier using a triaxial accelerometer for ambulatory monitoring. **IEEE transactions on information technology in biomedicine**, IEEE, v. 10, n. 1, p. 156–167, 2006. Citation on page 39.

KINGMA, D. P.; BA, J. Adam: A method for stochastic optimization. **arXiv preprint arXiv:1412.6980**, 2014. Citation on page 77.

KLINE, D.; TAKKEN, W.; WOOD, J.; CARLSON, D. Field studies on the potential of butanone, carbon dioxide, honey extract, l-octen-3-ol, l-lactic acid and phenols as attractants for mosquitoes. **Medical and veterinary entomology**, Wiley Online Library, v. 4, n. 4, p. 383–391, 1990. Citation on page 125.

KOTSIANTIS, S. B.; ZAHARAKIS, I.; PINTELAS, P. Supervised machine learning: A review of classification techniques. **Emerging artificial intelligence applications in computer engineering**, Amsterdam, v. 160, p. 3–24, 2007. Citation on page 45.

KREMPL, G.; ŽLIOBAITE, I.; BRZEZIŃSKI, D.; HÜLLERMEIER, E.; LAST, M.; LEMAIRE, V.; NOACK, T.; SHAKER, A.; SIEVI, S.; SPILIOPOULOU, M. *et al.* Open challenges for data stream mining research. **ACM SIGKDD explorations newsletter**, ACM New York, NY, USA, v. 16, n. 1, p. 1–10, 2014. Citation on page 28.

KUMAR, A.; GOYAL, S.; VARMA, M. Resource-efficient machine learning in 2 kb ram for the internet of things. In: JMLR. ORG. **Proceedings of the 34th International Conference on Machine Learning-Volume 70**. [S.l.], 2017. p. 1935–1944. Citation on page 36.

KUSUPATI, A.; SINGH, M.; BHATIA, K.; KUMAR, A.; JAIN, P.; VARMA, M. Fastgrnn: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network. In: **Advances in Neural Information Processing Systems**. [S.l.: s.n.], 2018. p. 9017–9028. Citation on page 36.

LAI, L.; SUDA, N.; CHANDRA, V. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. **arXiv preprint arXiv:1801.06601**, 2018. Citation on page 36.

LEVENBERG, K. A method for the solution of certain non-linear problems in least squares. **Quarterly of applied mathematics**, v. 2, n. 2, p. 164–168, 1944. Citation on page 58.

LIU, D. C.; NOCEDAL, J. On the limited memory bfgs method for large scale optimization. **Mathematical programming**, Springer, v. 45, n. 1-3, p. 503–528, 1989. Citation on page 76.

LUŠTREK, M.; KALUŽA, B. Fall detection and activity recognition with machine learning. **Informatica**, v. 33, n. 2, 2009. Citation on page 27.

LUXBURG, U. V.; SCHÖLKOPF, B. Statistical learning theory: Models, concepts, and results. In: **Handbook of the History of Logic**. [S.l.]: Elsevier, 2011. v. 10, p. 651–706. Citation on page 44.

MAAS, A. L.; HANNUN, A. Y.; NG, A. Y. Rectifier nonlinearities improve neural network acoustic models. In: **Proc. icml**. [S.l.: s.n.], 2013. v. 30, n. 1, p. 3. Citation on page 46.

MAINS, J. W.; BRELSFOARD, C. L.; ROSE, R. I.; DOBSON, S. L. Female adult aedes albopictus suppression by wolbachia-infected male mosquitoes. **Scientific reports**, Nature Publishing Group, v. 6, p. 33846, 2016. Citation on page 110.

MARQUARDT, D. W. An algorithm for least-squares estimation of nonlinear parameters. **Journal of the society for Industrial and Applied Mathematics**, SIAM, v. 11, n. 2, p. 431–441, 1963. Citation on page 58.

MARTIN, B. Instance-based learning: nearest neighbour with generalisation. University of Waikato, Department of Computer Science, 1995. Citation on page 44.

MELLANBY, K. Humidity and insect metabolism. **Nature**, Nature Publishing Group, v. 138, n. 3481, p. 124–125, 1936. Citation on page 113.

MITCHELL, T. M. **Machine Learning**. 1. ed. [S.l.]: McGraw-Hill, 1997. Citations on pages 43, 45, and 46.

MONARD, M. C.; BARANAUSKAS, J. A. Conceitos sobre aprendizado de máquina. **Sistemas Inteligentes-Fundamentos e Aplicações**, v. 1, n. 1, p. 32, 2003. Citations on pages 43 and 44.

MURTHY, S. K. Automatic construction of decision trees from data: A multi-disciplinary survey. **Data mining and knowledge discovery**, Springer, v. 2, n. 4, p. 345–389, 1998. Citation on page 45.

NUSSBAUMER, H. J. The fast fourier transform. In: **Fast Fourier Transform and Convolution Algorithms**. [S.l.]: Springer, 1981. p. 80–111. Citation on page 115.

NXP SEMICONDUCTORS. **K66 Sub-Family Reference Manual**. [S.l.], 2015. Rev. 2. Citation on page 89.

____. **Kinetis K64F Sub-Family Data Sheet**. [S.l.], 2016. Rev. 7. Citation on page 88.

____. **K64 Sub-Family Reference Manual**. [S.l.], 2017. Rev. 3. Citation on page 88.

____. **Kinetis K66 Sub-Family**. [S.l.], 2017. Rev. 4. Citation on page 89.

O'BARD, B.; GEORGE, K. Classification of eye gestures using machine learning for use in embedded switch controller. In: IEEE. **2018 IEEE International Instrumentation and Measurement Technology Conference (I2MTC)**. [S.l.], 2018. p. 1–6. Citations on pages 37 and 41.

OLIPHANT, T. E. **A guide to NumPy**. [S.l.]: Trelgol Publishing USA, 2006. Citation on page 51.

PAL, S. K.; MITRA, S. Multilayer perceptron, fuzzy sets, and classification. **IEEE Transactions on Neural Networks**, IEEE Press Piscataway, NJ, USA, v. 3, n. 5, p. 683–697, 1992. Citations on pages 45 and 47.

PATIL, S. G.; DENNIS, D. K.; PABBARAJU, C.; SHAHEER, N.; SIMHADRI, H. V.; SESHADRI, V.; VARMA, M.; JAIN, P. Gesturepod: Enabling on-device gesture-based interaction for white cane users. In: **Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology**. [S.l.: s.n.], 2019. p. 403–415. Citations on pages 39 and 41.

PEDREGOSA, F.; VAROQUAUX, G.; GRAMFORT, A.; MICHEL, V.; THIRION, B.; GRISEL, O.; BLONDEL, M.; PRETTENHOFER, P.; WEISS, R.; DUBOURG, V.; VANDERPLAS, J.; PASSOS, A.; COURNAPEAU, D.; BRUCHER, M.; PERROT, M.; DUCHESNAY, E. Scikit-learn: Machine learning in Python. **Journal of Machine Learning Research**, v. 12, p. 2825–2830, 2011. Citations on pages 29 and 51.

PENG, C.-Y. J.; LEE, K. L.; INGERSOLL, G. M. An introduction to logistic regression analysis and reporting. **The journal of educational research**, Taylor & Francis, v. 96, n. 1, p. 3–14, 2002. Citation on page 48.

PLATT, J. Fast training of support vector machines using sequential minimal optimization. In: SCHOELKOPF, B.; BURGES, C.; SMOLA, A. (Ed.). **Advances in Kernel Methods - Support Vector Learning**. MIT Press, 1998. Available: <http://research.microsoft.com/~jplatt/smo.html>. Citation on page 71.

POLAT, K.; GÜNEŞ, S. A novel hybrid intelligent method based on c4. 5 decision tree classifier and one-against-all approach for multi-class classification problems. **Expert Systems with Applications**, Elsevier, v. 36, n. 2, p. 1587–1592, 2009. Citation on page 48.

PREGIBON, D. *et al.* Logistic regression diagnostics. **The Annals of Statistics**, Institute of Mathematical Statistics, v. 9, n. 4, p. 705–724, 1981. Citation on page 48.

QI, Y.; CINAR, G. T.; SOUZA, V. M. A.; BATISTA, G. E. A. P. A.; WANG, Y.; PRINCIPE, J. C. Effective insect recognition using a stacked autoencoder with maximum correntropy criterion. In: IEEE. **2015 International Joint Conference on Neural Networks (IJCNN)**. [S.l.], 2015. p. 1–7. Citations on pages 109 and 112.

QUINLAN, J. R. Induction of decision trees. **Machine learning**, Springer, v. 1, n. 1, p. 81–106, 1986. Citation on page 45.

QUINLAN, R. **C4.5: Programs for Machine Learning**. San Mateo, CA: Morgan Kaufmann Publishers, 1993. Citation on page 45.

REIS, D. M. dos; MALETZKE, A. G.; BATISTA, G. E. A. P. A. Unsupervised context switch for classification tasks on data streams with recurrent concepts. In: **ACM SAC**. [S.l.: s.n.], 2018. p. 518–524. Citation on page 86.

ROSENBLATT, F. The perceptron: a probabilistic model for information storage and organization in the brain. **Psychological review**, American Psychological Association, v. 65, n. 6, p. 386, 1958. Citation on page 45.

ROWLEY, W. A.; GRAHAM, C. L. The effect of temperature and relative humidity on the flight performance of female aedes aegypti. **Journal of Insect Physiology**, Elsevier, v. 14, n. 9, p. 1251–1257, 1968. Citation on page 113.

RÚA, S.; ZULUAGA, S. A.; REDONDO, A.; OROZCO-DUQUE, A.; RESTREPO, J. V.; BUSTAMANTE, J. Machine learning algorithms for real time arrhythmias detection in portable cardiac devices: microcontroller implementation and comparative analysis. In: **STSIVA**. [S.l.: s.n.], 2012. p. 50–55. Citations on pages 29, 39, and 41.

SAMPAIO, F.; SILVA, L. C. da; FILHO, P. P. R.; SILVA, E. T. da. Reducing computational costs of an embedded classifier to determine leather quality. In: **SBESC**. [S.l.: s.n.], 2017. p. 211–216. Citation on page 29.

SCHAUL, T.; BAYER, J.; WIERSTRA, D.; SUN, Y.; FELDER, M.; SEHNKE, F.; RÜCKSTIESS, T.; SCHMIDHUBER, J. Pybrain. **Journal of Machine Learning Research**, Massachusetts Institute of Technology Press, v. 11, n. ARTICLE, p. 743–746, 2010. Citation on page 51.

SCHMIDHUBER, J. Deep learning in neural networks: An overview. **CoRR**, abs/1404.7828, 2014. Available: <http://arxiv.org/abs/1404.7828>. Citation on page 44.

SCHÖLKOPF, B.; SMOLA, A. J.; BACH, F. *et al.* **Learning with kernels: support vector machines, regularization, optimization, and beyond**. [S.l.]: MIT press, 2002. Citations on pages 49 and 50.

SHI, G.; CHAN, C. S.; LI, W. J.; LEUNG, K.-S.; ZOU, Y.; JIN, Y. Mobile human airbag system for fall protection using mems sensors and embedded svm classifier. **IEEE Sensors Journal**, v. 9, n. 5, p. 495–503, 2009. Citations on pages 28, 38, and 41.

SILVA, D. F.; SOUZA, V. M. A.; ELLIS, D. P.; KEOGH, E. J.; BATISTA, G. E. A. P. A. Exploring low cost laser sensors to identify flying insect species. **Journal of Intelligent & Robotic Systems**, Springer, v. 80, n. 1, p. 313–330, 2015. Citations on pages 27, 30, 109, and 112.

SILVA, D. F.; SOUZA, V. M. A. D.; BATISTA, G. E. A. P. A.; KEOGH, E.; ELLIS, D. P. W. Applying machine learning and audio analysis techniques to insect recognition in intelligent traps. In: **ICMLA**. [S.l.: s.n.], 2013. p. 99–104. Citation on page 28.

SILVA, L. T. da; SOUZA, V. M. A.; BATISTA, G. E. A. P. A. EmbML tool: supporting the use of supervised learning algorithms in low-cost embedded systems. In: IEEE. **2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)**. [S.l.], 2019. p. 1633–1637. Citations on pages 29, 31, 53, 54, and 85.

____. Uma ferramenta de suporte ao uso de classificadores em sistemas embarcados. In: **Anais do 14º Simpósio Brasileiro de Automação Inteligente**. [S.l.: s.n.], 2019. p. 2901–2907. Citations on pages 29, 31, and 53.

SOUZA, V. M. A.; GIUSTI, R.; BATISTA, A. J. L. Asfault: A low-cost system to evaluate pavement conditions in real-time using smartphones and machine learning. **Pervasive and Mobile Computing**, v. 51, p. 121–137, 2018. Citation on page 86.

SOUZA, V. M. A.; REIS, D. M.; MALETZKE, A. G.; BATISTA, G. E. A. P. A. Challenges in benchmarking stream learning algorithms with real-world data. **Data Mining and Knowledge Discovery**, p. 1–54, 2020. Citations on pages 109, 110, 111, 114, 115, 116, and 117.

SOUZA, V. M. A.; SILVA, D. F.; BATISTA, G. E. A. P. A. Classification of data streams applied to insect recognition: Initial results. In: **BRACIS**. [S.l.: s.n.], 2013. p. 76–81. Citations on pages 27, 30, and 109.

STANKOVIC, J. A. Misconceptions about real-time computing: A serious problem for next-generation systems. **Computer**, v. 21, n. 10, p. 10–19, 1988. Citation on page 28.

TAYLOR, L. Analysis of the effect of temperature on insects in flight. **The Journal of Animal Ecology**, JSTOR, p. 99–117, 1963. Citation on page 113.

TOCCHETTO, M. A.; BAZANELLA, A. S.; GUIMARAES, L.; FRAGOSO, J.; PARRAGA, A. An embedded classifier of lung sounds based on the wavelet packet transform and ann. **IFAC Proceedings Volumes**, v. 47, n. 3, p. 2975–2980, 2014. Citations on pages 29, 38, and 41.

TUBAISHAT, M.; MADRIA, S. Sensor networks: an overview. **IEEE potentials**, v. 22, n. 2, p. 20–23, 2003. Citation on page 27.

VAPNIK, V. **The Nature of Statistical Learning Theory**. [S.l.]: Springer Science & Business Media, 1999. Citations on pages 49 and 50.

VERGARA, A.; VEMBU, S.; AYHAN, T.; RYAN, M. A.; HOMER, M. L.; HUERTA, R. Chemical gas sensor drift compensation using classifier ensembles. **Sensors and Actuators B: Chemical**, v. 166, p. 320–329, 2012. Citation on page 87.

Virtanen, P.; Gommers, R.; Oliphant, T. E.; Haberland, M.; Reddy, T.; Cournapeau, D.; Burovski, E.; Peterson, P.; Weckesser, W.; Bright, J.; van der Walt, S. J.; Brett, M.; Wilson, J.; Jarrod Millman, K.; Mayorov, N.; Nelson, A. R. J.; Jones, E.; Kern, R.; Larson, E.; Carey, C.; Polat, İ.; Feng, Y.; Moore, E. W.; Vand erPlas, J.; Laxalde, D.; Perktold, J.; Cimrman, R.; Henriksen, I.; Quintero, E. A.; Harris, C. R.; Archibald, A. M.; Ribeiro, A. H.; Pedregosa, F.; van Mulbregt, P.; Contributors, S. . . SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. **Nature Methods**, v. 17, p. 261–272, 2020. Citation on page 51.

VLĂDUŢIU, M. **Computer arithmetic: algorithms and hardware implementations**. [S.l.]: Springer Science & Business Media, 2012. Citation on page 58.

Wang, X.; Magno, M.; Cavigelli, L.; Benini, L. Fann-on-mcu: An open-source toolkit for energy-efficient neural network inference at the edge of the internet of things. **IEEE Internet of Things Journal**, 2020. Citation on page 36.

WELCH, P. The use of fast fourier transform for the estimation of power spectra: a method based on time averaging over short, modified periodograms. **IEEE Transactions on audio and electroacoustics**, IEEE, v. 15, n. 2, p. 70–73, 1967. Citations on pages 114 and 116.

WITTEN, I. H.; FRANK, E.; HALL, M. A.; PAL, C. J. **Data Mining: Practical Machine Learning Tools and Techniques**. [S.l.]: Morgan Kaufmann, 2016. Citation on page 52.

YICK, J.; MUKHERJEE, B.; GHOSAL, D. Wireless sensor network survey. **Computer networks**, Elsevier, v. 52, n. 12, p. 2292–2330, 2008. Citation on page 28.

ZHANG, X.; ZHOU, X.; LIN, M.; SUN, J. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. [S.l.: s.n.], 2018. p. 6848–6856. Citation on page 131.

ZITO, T.; WILBERT, N.; WISKOTT, L.; BERKES, P. Modular toolkit for data processing (mdp): a python data processing framework. **Frontiers in neuroinformatics**, Frontiers, v. 2, p. 8, 2009. Citation on page 51.


ICMC
USP
SÃO CARLOS