Università di Genova

DIBRIS DIPARTIMENTO
DI INFORMATICA, BIOINGEGNERIA,
ROBOTICA E INGEGNERIA DEI SISTEMI

A pipeline to support users in selecting machine learning models and porting them to FPGAs

Yryskeldi Siddi

Master Thesis

Università di Genova, DIBRIS Via Opera Pia, 13 16145 Genova, Italy
https://www.dibris.unige.it/

Università
di Genova

MSc Computer Science
Data Science and Engineering Curriculum

A pipeline to support users in selecting machine learning models and porting them to FPGAs

Yryskeldi Siddi

Advisor: Daniele D'Agostino, Giorgio Delzanno
Examiner: Nicoletta Noceti

March, 2026

Abstract

This thesis presents a pipeline integrating Automated Machine Learning (AutoML) with Field-Programmable Gate Arrays (FPGAs) to simplify the development and deployment of high-performance predictive models. Using AutoWeka, the system automates algorithm selection and hyperparameter tuning, identifying Multinomial Logistic Regression (MLR) for classifying physical activities from physiological sensor data. The model is deployed on an AMD/Xilinx Artix™ UltraScale+ FPGA using Vitis High-Level Synthesis (HLS), which converts C++ descriptions into RTL hardware architectures. Results show a 9% speed improvement over CPU execution for single instances, up to 75% reduction in execution time through parallel replication, and significantly higher energy efficiency, consuming about one-sixth of the CPU's power.

Table of Contents

Chapter 1

Introduction

In recent years, Machine Learning (ML) has emerged as one of the cornerstone technologies in Artificial Intelligence (AI), thanks to its ability to extract knowledge from data and support decision-making through advanced predictive models. However, designing and training effective machine learning models is a complex task, requiring highly specialized skills as well as a significant investment in time and computational resources. This process involves several critical steps, including data selection and preprocessing, choosing the most appropriate algorithm, optimizing hyperparameters and evaluating model performance.
At the same time, the growing complexity of machine learning models has led to an exponential increase in both computational demands and the volume of data to be processed. This evolution has highlighted the limitations of traditional general-purpose computing architectures, such as CPUs, particularly regarding latency, energy consumption and temporal performance predictability. Moreover, the entire model development cycle is often iterative, costly and accessible primarily to experts in data science, machine learning and hardware design [Tri18].

FPGAs, on the other hand, are being adopted as an alternative and complementary hardware acceleration platform to traditional computing architectures. Although the use of FPGAs for accelerating machine learning models has historically been associated with high design complexity [BPT24], the introduction of High-Level Synthesis (HLS) techniques has significantly lowered the threshold for accessing these technologies [SW19]. HLS represents a significant evolution in hardware design flows, enabling the automatic translation of algorithmic descriptions expressed in high-level languages, such as C or C++, into Register-Transfer Level (RTL) architectures, intended for implementation on reconfigurable devices such as FPGAs or System-on-Chip (SoCs) [Tri18]. In this way, even designers without extensive hardware experience can benefit from the high computational capacity and energy efficiency offered by FPGAs.

To reduce this complexity and promote the democratization of the use of artificial intelligence in heterogeneous application areas, this thesis proposes a pipeline that integrates Automated Machine Learning (AutoML) with Field Programmable Gate Arrays (FPGAs). In this context, AutoML is used to automate and simplify the various phases of the machine learning model development process, enabling even users with limited expertise to build high-quality predictive models more quickly, efficiently and reproducibly.
The proposed pipeline was applied to the sports field, to recognize and classify the type of movement performed by individuals based on their physical state. This application scenario represents a significant case study, as it combines dynamic data analysis with efficient processing requirements and real-time responses. However, the high degree of generalizability of the developed approach allows the pipeline to be extended to different application contexts, characterized by constraints in terms of latency, energy consumption and computational efficiency.

For example, sectors such as industrial and automotive can benefit from the integration of AutoML techniques and hardware acceleration via FPGA, making the proposed solution versatile and potentially applicable to a wide range of real-world scenarios. In the industrial and manufacturing sectors [RMKR24], the pipeline can be used for predictive maintenance, production process monitoring and automatic quality control applications. AutoML simplifies the generation of predictive models from data from industrial sensors, while implementation on FPGAs enables real-time processing with high reliability and low energy consumption. In the automotive sector [YYH$^+$25], the combination of AutoML and FPGA is particularly effective for applications related to advanced driver assistance systems (ADAS) and environmental awareness. In these scenarios, the low latency and temporal predictability offered by FPGAs are key requirements, while AutoML allows machine learning models to be adapted and optimized for different operating and environmental conditions.

This thesis is structured as follows. Chapter 2 presents a review of the scientific literature on Automated Machine Learning techniques and Field Programmable Gate Arrays, with particular attention to related works and solutions already proposed in the literature that are comparable to the approach developed in this work.
Chapter 3 describes in detail the dataset used for the experiment, illustrating the preprocessing operations applied to the data and the process of building the machine learning models through the use of the AutoWeka tool.
Chapter 4 presents the technical specifications of the adopted FPGA platform and delves into the process of porting the machine learning models to reconfigurable hardware, achieved using the Vitis HLS tool.
Finally, Chapter 5 reports the general conclusions of the thesis and highlights possible future developments of the work carried out.

Chapter 2

Background and Related works

There are few studies in the literature that jointly analyze AutoML and FPGAs, despite both fields having extensive and well-established scientific output when considered separately.
The works that connect these two areas will be discussed in the final section of the chapter, as a preliminary discussion of each area is essential to understanding the contribution developed in this thesis.

2.1 AutoML

The term Automated Machine Learning (AutoML) was introduced in 2015 as part of the ChaLearn competition [GBC+15], with the aim of identifying a set of tools and methodologies aimed at automating machine learning processes. The need for such approaches arises from the intrinsic complexity of designing effective learning models, an activity that requires advanced expertise in both machine learning algorithms and the application domain. Lacking such expertise, users tend to select algorithms heuristically and use default configurations, or resort to brute-force techniques to explore the space of possible solutions. However, the large number of combinations of algorithms and hyperparameters quickly makes these strategies impractical [HHLB11]. Consequently, although machine learning is often presented as a paradigm that reduces the need for explicit programming, in practice, it requires a significant investment in time and manual development.

To properly understand how AutoML systems work, it's necessary to analyze the complexity of the tasks they aim to automate. To this end, we refer to the typical supervised machine learning workflow, which represents a meaningful case study. This pipeline is divided into three main phases: data preparation, feature engineering and model generation

as shown in Figure 2.1. Each of these phases presents specific critical issues, which have been the subject of extensive research aimed at their automation.

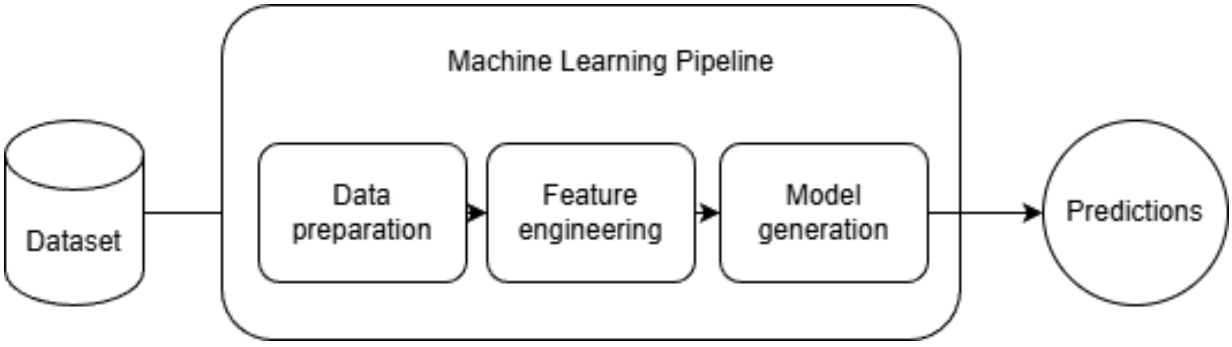


Figure 2.1: An example of a common machine learning pipeline.

2.1.1 Data preparation

Data preparation is the first step in the machine learning pipeline. The goal is to identify as much relevant and high-quality data as possible. Often, when a need and use case for a predictive model are identified, there is already an idea of what data to use as a starting point. This data can be collected from the individual or organization that will use the model, or it can come from a reliable source. To obtain additional data, a method for collecting essential information must be established or it can be searched online. One problem with using web data is that it can be incorrectly labeled or even unlabeled. This challenge can be addressed through the use of self-labeling methods. Several labeling methods exist that help reduce human involvement in the process and improve accuracy, i.e. Self-training, Co-training and Co-learning.

Self-training is a semi-supervised learning approach aimed at reducing reliance on manual labeling by leveraging the availability of large amounts of unlabeled data. The process begins with training a model on an initial set of labeled data, generally small but considered reliable. The model is then used to generate predictions on unlabeled data. Among these predictions, only those associated with a high level of confidence are selected, assuming they are likely correct. These selected instances are then self-labeled and integrated into the training dataset. The model is retrained using the extended dataset, progressively improving its predictive ability. This approach allows for the labeled dataset to be expanded automatically [Yar95].

Co-training is a semi-supervised learning method designed to exploit unlabeled data through the cooperative use of multiple models. Co-training works as follows. Initially, a small set of labeled data and a large set of unlabeled data are provided. Two separate classifiers are

trained, each using a different view of the same instances. Subsequently, each classifier is used to predict the labels of the unlabeled data. Highly confident predictions are selected and used to automatically label new instances [FAHS10].

Co-learning is a semi-supervised learning approach that extends and generalizes the concept of co-training. Specifically, co-learning does not require different views of the data to be conditionally independent, but is based on the idea that different models can learn collaboratively by leveraging complementary representations or hypotheses. Co-learning involves the simultaneous training of multiple classifiers, which may differ in architecture, feature space, or biases, using both labeled and unlabeled data. Initially, each model is trained on the same small set of labeled data. Subsequently, the models are used to generate predictions on the unlabeled data. Instances for which one or more models show high confidence are selected and used as new learning information [ZG04].

Furthermore, the distribution of web data can be very different from the dataset it is to be integrated with, making model training more complex, i.e. web activity, web images. This problem can be mitigated, for example, by fine-tuning web data. Web data fine-tuning is a strategy used to make web data more suitable for training a machine learning model. The underlying principle is that web data, despite being abundant, often does not follow the same distribution as the data on which the model will actually be used. This misalignment, known as a domain shift, can compromise the model's performance. Operationally, the process begins with preliminary training of the model on a reference dataset, which is generally smaller but high-quality and well-labeled. Subsequently, the model is exposed to web data, which is used in a controlled manner. Through an iterative process, the model is trained on web data while, in parallel, instances that are less consistent with the target domain are filtered or weighted. Instances that produce high errors or unstable predictions can be eliminated or reduced in importance [CG15].

If web data is unbalanced, the synthetic minority class oversampling technique, known as SMOTE [CBHK02], can be used to synthesize new samples from existing minority class samples. SMOTE works in the following basic steps. First, for each sample in the minority class, the k closest neighbors within the same class are identified. Next, one or more of these neighbors are randomly selected. From this selection, new synthetic samples are generated by linearly interpolating the features between the original sample and the chosen neighbor. Operationally, the new sample is created as a weighted combination of the two points in feature space. This process is repeated until the desired level of balance between the classes is achieved [CBHK02].

Data augmentation is a process that can be traced, at least in part, back to data collection, as it increases the size and diversity of an existing dataset through various transformation processes. It helps the model learn better generalization and is commonly used as a method to avoid overfitting [HZC21]. The techniques used for data augmentation vary depending on the type of data. For example, in the case of images, it is possible to apply rotations,

resizing, cropping and reflections to existing images. For textual data, however, it is possible to perform synonym insertion or translation-based augmentation, translating the text into a new language and then reverting it to the original language. However, these processes still require explicit human intervention in defining augmentation policies. One of the proposed techniques is AutoAugment [CZM+19], which aims to automatically search for augmentation policies. AutoAugment works by searching for possible augmentations using reinforcement learning algorithms. A controller, often a neural network, proposes combinations of operations (policies) to apply to the dataset. Each proposed policy is evaluated by training a test model on the transformed data. The model's performance, typically accuracy on a validation dataset, serves as a reward for the controller. AutoAugment completely automates the selection of the most effective transformations, replacing manual work and allowing for larger datasets. Although this initial technique had obvious limitations in terms of efficiency, more recent solutions have been proposed to address this problem.

Gradient descent-based [LHW+20] approaches for data augmentation, function as an automated and differentiable method for finding the best data transformation policies. Unlike AutoAugment, which uses an external search algorithm, these approaches integrate search directly into the gradient optimization process. During model training, the derivative of the loss function with respect to the transformation parameters is computed. This way, the transformations are directly updated to maximize model performance and at each iteration, both the model weights and the transformation parameters are updated using gradient descent. This allows for the efficient identification of the most effective augmentation policies. Once the parameters are optimized, the most effective transformations are applied to the dataset, generating artificial data that improves generalization and reduces overfitting.

Another approach to data augmentation is greedy search strategies [NAM20]. These are heuristic and iterative methods designed to identify effective augmentation policies while significantly reducing computational cost compared to more complex approaches such as reinforcement learning. The fundamental principle of greedy search is to make the best possible local decision at each iteration, assuming that a sequence of locally optimal choices leads to an overall valid solution. However, their main limitation is the possibility of converging to local optima, since short-sightedly selecting the best immediate solutions does not guarantee the identification of the globally optimal policy.

Because high-quality data is crucial for machine learning development, dataset cleaning is often necessary before using it for model training. This is a process in which corrupt, incorrectly formatted, duplicate, or incomplete data is corrected or removed [CIKW16]. Broadly speaking, this process can be divided into two main functions: error detection and error correction. Traditionally, data cleaning has been a step in the ML pipeline requiring specialized knowledge; however, research into its automation has already led to

the development of several techniques.

Some studies have proposed cleaning only a portion of the dataset and then applying similar processes to the rest of the data, but this approach still requires the initial intervention of a data scientist [KHD+15]. Another solution that addresses the same issue was proposed by [CMI+15] with Katara, a knowledge-based, crowdsourced data cleaning system. However, there are also solutions that completely eliminate the data scientist's involvement in the process.

BoostClean [KFGW17] automates data cleaning by treating it as a boosting problem. Each cleaning operation applied to the dataset adds a new transformation to the input of the downstream machine learning model. Through a combination of feature selection and boosting, it is possible to generate different operations that positively impact the model's accuracy. The cleaning process can also be treated as a hyperparameter optimization problem, as is done in AlphaClean [KW19].

2.1.2 Feature engineering

Feature engineering is a process aimed at maximizing the effectiveness of a dataset for the model. The three sub-operations of feature engineering process are: feature selection, feature construction and feature extraction, through them is possible to train a more accurate model than using raw data alone. The first two operations can be defined as feature transformation, as they lead to the generation of new features [ML02]. In most cases, feature extraction is used to reduce dimensionality using mapping functions, feature construction to augment the original features and feature selection to remove redundant features. The main feature selection methods fall into three categories, i.e. filter, wrapper and embedded methods [LM12].

- The filter method evaluates each feature based on divergence or correlation measures and selects features based on a threshold; variance, correlation coefficients, chi-square tests and mutual information are typical evaluation criteria.
- In the wrapper method, a set of samples is classified using the selected subset of features and classification accuracy is used as a quality criterion.
- The embedded method integrates variable selection directly into the learning process; regularization, decision trees and deep learning are common examples of this approach.

Feature construction involves generating new variables from raw data, with the aim of increasing the model's robustness and improving the descriptive capacity of the original

features. Traditionally, this process is highly dependent on the expert's experience and relies on preprocessing transformations such as standardization, normalization and discretization. The techniques adopted vary depending on the type of features. To reduce the dependence on human intervention, automated approaches have been proposed that can achieve results comparable to or superior to those of experts. In these approaches, methods based on decision trees and genetic algorithms [Gam04] [Zhe98] [Son09] [VDJ98] operate within a predefined transformation space, while annotation-based methods directly exploit domain knowledge. The latter are inserted into an interactive feature construction protocol, in which an agent identifies inadequate representations and refines them with the support of semantic resources and the domain expert.

Feature extraction is a process aimed at reducing the dimensionality of data through the use of mapping functions. Unlike feature selection, this operation involves a transformation of the original feature space, allowing only the most relevant information to be isolated and preserved based on predefined metrics. Numerous methods have been proposed in the literature to implement these mapping functions, including Principal Component Analysis (PCA), Independent Component Analysis, Isomap, nonlinear dimensionality reduction techniques and Linear Discriminant Analysis (LDA). In recent years, approaches based on feed-forward neural networks have also gained increasing relevance. In particular, algorithms based on autoencoders have been introduced [MCSK17] [IA17], which allow for learning compact data representations by exploiting both the intrinsic characteristics of the features and the relationships between them, allowing for unsupervised feature extraction.

2.1.3 Model generation

Model generation in AutoML systems is primarily structured around two fundamental elements: defining the search space and using optimization methods. The strategies adopted vary depending on the type of model being trained, distinguishing between traditional machine learning algorithms, such as Support Vector Machines and k-nearest neighbors and deep neural networks, for which Neural Architecture Search (NAS) represents an increasingly important approach. Because model generation is computationally expensive, the efficiency of the search process plays a crucial role in identifying the best-performing solutions.

In traditional contexts, model and hyperparameter selection was entrusted to the data scientist's experience; in AutoML frameworks, this task is automated. For classic models, the system explores available algorithms, defines hyperparameter spaces and uses an optimizer to identify the optimal configuration [NP19] [THHLB13] [KBE14] [OM16] [FKE+15]. In the case of NAS, the search space codifies the design principles of neural architectures, enabling the automatic identification of high-performance structures [EMH19] [PGZ+18] [ZL16] [RAHL19].

2.2 FPGA

Field-Programmable Gate Arrays (FPGAs) represent a class of reconfigurable logic devices that have played a key role in the evolution of modern digital electronics. As highlighted by Trimberger in his retrospective analysis of the first thirty years of FPGA development, these devices were conceived with the aim of combining the flexibility typical of software with the high performance of dedicated hardware [Tri18]. From an architectural point of view, an FPGA is made up of a regular matrix of programmable logic blocks (Configurable Logic Blocks, CLB), interconnected via a configurable routing network and equipped with input/output blocks for interaction with the external environment. An example of the FPGA architecture can be seen in Figure 2.2.

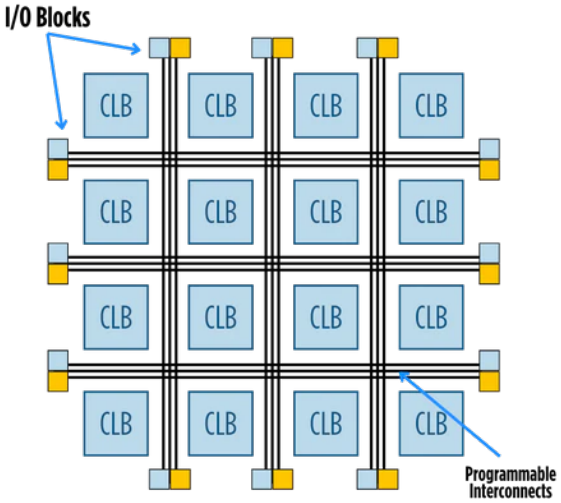


Figure 2.2: An example of a simple FPGA architecture.

Configurable Logic Blocks (CLBs): the core of FPGA functionality, these blocks contain logic gates and a small amount of memory that can be programmed to perform various logical functions. They are the building blocks for creating more complex digital circuits, an example of the CLB in Figure 2.3.

Programmable Interconnects: a network of programmable wiring that connects the logic blocks. These interconnects can be configured to route signals between different logic blocks, allowing the creation of complex digital circuits by defining how data flows through the FPGA.

I/O Blocks (Input/Output Blocks): located next to every physical input or output

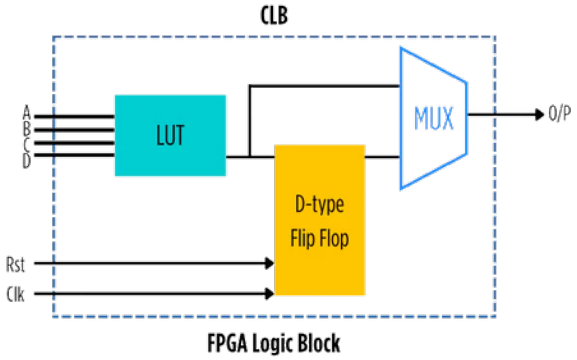


Figure 2.3: An example of a CLB composition. LUT (Look Up Table) implements the combinational logical functions; the MUX (Multiplexer) is used for selection logic and DFF (D type flip flop) stores the output of the LUT.

pin, these blocks connect the internal logic of the FPGA to the external environment. They can be programmed to act as inputs, outputs or tri-states, enabling the FPGA to communicate with external devices and systems.

The distinctive feature of FPGAs is their post-fabrication programmability, which allows for the implementation and modification of logic functions after the integrated circuit has been produced. This property differentiates them from Application-Specific Integrated Circuits (ASICs), offering an effective compromise between flexibility, reduction of development costs and exploitation of hardware parallelism [Tri18] [REGSV02]. According to Trimberger, the evolution of FPGAs can be divided into different phases, each characterised by a progressive increase in architectural complexity and computational capabilities.

Over time, FPGAs have integrated increasingly heterogeneous resources, such as embedded memories, dedicated multiplication units and soft-core or hard-core processors, evolving into true reconfigurable computing platforms [Tri18]. This transformation has significantly expanded the application domains, including telecommunications, digital signal processing, embedded systems and hardware acceleration for high-performance applications. As reported in the literature, FPGAs today constitute a reference solution for systems requiring high energy efficiency, adaptability and massive parallelism [REGSV02] [CH02].

However, the increasing complexity of modern FPGAs has made the traditional Register-Transfer Level (RTL) design approach increasingly less efficient and scalable. Classic RTL design requires the designer to explicitly manage aspects such as parallelism and pipelining, task scheduling, resource allocation, time synchronization and communication protocols. Writing RTL code manually is inherently time-consuming and highly error-prone, especially for complex algorithms; furthermore, even limited functional changes often require a substantial revision of the hardware architecture. A further relevant limitation is the

practical impossibility of systematically exploring large design spaces: evaluating different RTL solutions in order to balance area, performance and power consumption is extremely costly in terms of time and resources. Such activities also require highly specialized skills in digital architectures, timing and logic synthesis, restricting the number of designers able to develop complex FPGA accelerators [REGSV02] [CH02]. An example of RTL code can be seen in Code 2.1, which describes a 4-bit synchronous binary counter.

```
1   module counter_4bit (
2   input  wire clk,
3   input  wire reset,
4   output reg  [3:0] count
5   );
6
7   always @(posedge clk) begin
8       if (reset)
9           count <= 4'b0000;
10      else
11          count <= count + 1'b1;
12  end
13  endmodule
```

Listing 2.1: A 4-bit synchronous binary counter in Verilog

In the Code 2.1, you can see four main elements of the RTL/Verilog language. The ***module*** element represents the fundamental design unit in Verilog. At the RTL level, the module defines: the status registers (***count***); the control signals (***clk***, ***reset***); the data transfer relationships. The *reg* element, which at the RTL level represents a register that holds its value between two consecutive clock cycles and is typically implemented as a bank of flip-flops. The ***always @(posedge clk)*** element defines the time boundary within which data transfers between registers occur; in this case, the contents of the registers are updated only on the rising edge of the clock, representing synchronous behavior. The last ***if/else*** element represents the control logic. If there is a reset signal, the count register is reset to zero; otherwise, the count register is incremented on each rising edge of the clock.

In this context, the high level of detail required by manual RTL design makes the process poorly scalable as functional complexity increases. High-Level Synthesis (HLS) is proposed as an effective alternative, allowing the algorithmic behavior of the system to be described at a higher level of abstraction and delegating the generation of the micro-architecture to the hardware compiler. This approach significantly reduces the cognitive load on the designer and introduces a clear separation between the algorithmic and architectural levels. The designer can thus focus on the correctness and efficiency of the algorithm, while HLS tools automatically explore the space of possible hardware implementations, allowing a faster evaluation of the trade-offs between area, performance and energy consumption [CH02] [BPT24].

2.2.1 High-Level Synthesis

HLS is based on the idea of separating the functional description of the algorithm from its micro-architectural implementation. The designer describes the computational behavior of the system, while the HLS tool takes care of transforming this description into a concrete hardware architecture. This process includes code analysis, operation scheduling, resource allocation and final generation of the synthesizable RTL code [BPT24].

Unlike traditional logic synthesis, which operates on already structured RTL descriptions, HLS faces a much broader design space exploration problem. Every scheduling, pipeline, or parallelization decision directly impacts key metrics such as area, latency, throughput and energy consumption. For this reason, HLS tools provide directives (pragmas) that allow the designer to guide the synthesis process towards specific design goals [CGMT09].

One of the central aspects of HLS is the evaluation of the Quality of Results (QoR). The most commonly considered metrics include the utilization of logic resources (LUT, flip-flop, DSP, BRAM), execution latency, throughput and power consumption. The trade-off between these metrics defines the goodness of a design solution [BPT24].

Design space exploration (DSE) consists of evaluating different configurations of HLS directives to find optimal or near-optimal solutions. However, the design space grows combinatorially with the number of parameters considered, making exhaustive approaches impractical. This problem is widely recognized in the literature and is one of the main factors limiting the large-scale industrial adoption of HLS [CZ06].

In particular, in the context of HLS, ML is used to build predictive models capable of rapidly estimating QoR metrics from features extracted from high-level code or intermediate design representations. Approaches based on regression, neural networks and ensemble models have been proposed to predict area, latency and power consumption without having to execute the entire synthesis [HZC21]. Such models drastically reduce the number of synthesis iterations required during DSE, accelerating the design process. This paradigm is consistent with the view proposed by Biscontini in [BPT24], according to which ML allows capturing complex relationships between design parameters and final outcomes that are difficult to model analytically.

The state of the art in High-Level Synthesis for FPGAs is the result of over two decades of academic research and industrial development, with a progressive shift from experimental approaches to mature tools adopted in production settings. Early work on HLS was mainly focused on the automatic translation of behavioral specifications into hardware, but suffered from significant limitations in terms of result quality compared to manual RTL designs. However, the evolution of FPGA architectures and compilers has allowed a substantial improvement in the performance and reliability of HLS flows [BPT24].

Currently, the state of the art is dominated by commercial and open-source tools that

leverage advanced compilation infrastructures, often based on LLVM and offer broad support for optimization directives. Tools such as Xilinx Vivado HLS, Vitis HLS and Intel Altera HLS Compiler are examples of established industrial solutions capable of generating highly parameterizable hardware that can be integrated into complex systems. At the same time, academic frameworks such as LegUp [CCA+11] or PandA-bambu [FCC+21] have contributed to the development of innovative scheduling, binding and automatic optimization techniques.

A central aspect of the state of the art is the growing emphasis on automated Design Space Exploration [SW19]. Recent literature recognizes that the complexity of the HLS design space makes manual or fixed heuristics-based approaches ineffective. Consequently, advanced methods have been proposed that combine analytical models, rapid simulations, and, increasingly, machine learning techniques to guide the exploration of design configurations. In this context, HLS is no longer seen as a simple hardware compiler, but as part of an intelligent optimization system.

From an application perspective, the state of the art shows a strong convergence between HLS and computationally intensive domains, such as signal processing, computer vision, cryptography and machine learning algorithm acceleration. In these areas, HLS significantly reduces the gap between software development and hardware implementation, fostering hardware-software co-design. Recent studies demonstrate that, with appropriate use of directives and guided optimization flows, HLS designs can achieve performance comparable to manual RTL implementations, with significant savings in development time [UDP+20].

Another trend characterizing the state of the art is the systematic integration of machine learning into HLS flows. As highlighted by Biscontini in [BPT24], ML is used not only for the prediction of quality metrics, but also for the automatic optimization of directives, architecture selection and reduction of synthesis times. This data-driven approach represents one of the most relevant and promising trends in current research.

2.3 AutoWeka

In recent decades, machine learning has assumed a central role in scientific research and industrial applications, thanks to its ability to extract knowledge and complex patterns from data. However, designing effective machine learning models requires a series of complex technical decisions, such as choosing the learning algorithm, selecting features, configuring hyperparameters and evaluating performance [HKV19]. This necessitates the presence of skilled people who know how to solve problems related to the design of machine learning models. To address the complexities inherent in the model selection pipeline, the discipline of Automated Machine Learning (AutoML) emerged, aimed at automating key phases of

the modeling process, reducing the need for skilled human intervention and improving workflow efficiency [FKE+15]. For this thesis, automation was applied to the selection of the algorithm, the selection of hyperparameters and validation using AutoWeka.

WEKA (Waikato Environment for Knowledge Analysis) is an open-source platform for data mining and machine learning, developed at the University of Waikato, New Zealand in 2005 [WFH+05]. It offers a vast collection of algorithms for classification, regression, clustering and feature selection and is used primarily in academic settings for experimentation and teaching. However, the platform, despite offering a wide variety of tools, requires the user to manually select the algorithm and configure its parameters, operations that are often non-trivial even for experts. For this thesis, Weka was used only for simple data visualization and data relationships, given the option to use it in a GUI rather than the command line. AutoWeka is one of the first AutoML systems developed and is still considered a benchmark in the field, developed by Thornton, Hutter, Hoos and Leyton-Brown in 2013. AutoWeka introduced a system that automates the selection and configuration of machine learning models in the WEKA library. The goal is to reduce the need for manual intervention and improve model quality through automatic optimization [THHLB13].

In the field of Automated Machine Learning, there are several open source or proprietary tools such as auto-sklearn [FKE+15], TPOT [OBUM16] and H2O AutoML [LP+20]. The choice of AutoWeka as an automation tool was made for several reasons: it is one of the first automation tools; since AutoWeka is a WEKA package, it maintains the same feature of being able to be used in GUI form, simplifying use compared to the command line interface, which requires more attention; AutoWeka does not have any additional requirements compared to WEKA. If you can run WEKA, you should be able to run AutoWeka; another advantage of AutoWeka is the fact that it is open source; it has access to a multitude of algorithms.

The algorithms are:

- Bayes Net
- Naive Bayes
- Naive Bayes Multinomial
- Gaussian Process
- Linear Regression
- Logistic Regression
- Single-Layer Perceptron
- Stochastic Gradient Descent
- SVM
- Simple Linear Regression
- Simple Logistic Regression
- Voted Perceptron
- KNN
- K-Star
- Decision Table
- RIPPER

- M5 Rules
- 1-R
- PART
- 0-R
- Decision Stump
- C4.5 Decision Tree
- Logistic Model Tree
- M5 Tree
- Random Forest
- Random Tree
- REP Tree
- Locally Weighted Learning
- AdaBoost M1
- Additive Regression
- Attribute Selected
- Bagging
- Classification via Regression
- LogitBoost
- MultiClass Classifier
- Random Committee
- Random Subspace
- Voting
- Stacking

Each algorithm is associated with a set of hyperparameters, totaling hundreds of configurable variables. The resulting space easily exceeds 700 parameters, making manual search impractical [THHLB13]. AutoWeka, thanks to Bayesian optimization, reduces the number of experiments needed to find optimal configurations [HHLB11]. In several benchmarks, AutoWeka has achieved results comparable to or superior to human experts [THHLB13].

2.4 Related Works

Among them, two contributions stand out which, while sharing the adoption of AutoML techniques for FPGA-based systems, differ in terms of objectives, application domain and design abstraction level.

The work AutoML for Multilayer Perceptron and FPGA Co-design [CSSM20] proposes a co-design approach between Multilayer Perceptron (MLP) neural networks and FPGA hardware architectures, aiming to overcome the limitations of traditional design flows, in which model and hardware optimization are approached as independent problems. This separation often leads to suboptimal solutions in terms of overall system performance. The main contribution of this work lies in the definition of a Neural Architecture Search (NAS) framework, in which the search space simultaneously includes architectural parameters of

the neural network and configuration parameters of the FPGA hardware. On the model side, the NAS automatically explores the number of layers, the number of neurons per layer, the activation functions and the numerical precision of the operations. At the same time, the hardware design considers aspects such as the degree of parallelism, pipeline strategies, the use of FPGA resources (DSP, LUT and BRAM) and the operating frequency.

Candidate architecture selection is formulated as a multi-objective optimization problem, taking into account metrics such as accuracy, inference latency, throughput and resource utilization. Hardware performance is estimated using cost models or rapid HLS-based synthesis, allowing for the automatic exclusion of configurations that cannot be mapped to the target FPGA. The process produces a set of Pareto-optimal solutions, which are subsequently validated on real FPGAs, demonstrating the effectiveness of the AutoML-driven co-design approach for general-purpose neural inference systems in embedded and high-performance contexts.

The other work Hardware-Aware AutoML for Exploration of Custom FPGA Accelerators for RadioML [Jen23] addresses a specific application domain, addressing the problem of efficiently designing FPGA accelerators for Radio Machine Learning (RadioML) applications. In this case, the focus is not on a particular class of neural networks, but rather on the automatic hardware-software co-design of custom accelerators optimized for radio signal processing. The central contribution of this work is the introduction of a hardware-aware AutoML framework that directly integrates hardware metrics into the model optimization process, enabling a joint exploration of the space of network architectures and FPGA accelerator hardware configurations. The search space includes machine learning model parameters, such as network depth, parallelism and quantization and hardware implementation parameters, including the use of LUTs, BRAMs and DSPs, as well as latency and throughput metrics. The goal is to identify solutions that maximize radio signal classification performance while ensuring efficient use of available hardware resources.

Unlike the first work, which focuses primarily on the co-design of MLP networks and the generality of the NAS framework, this second contribution is strongly oriented towards the RadioML domain and the stringent requirements of radio signal processing systems, such as real-time constraints, low latency and limited power consumption. The proposed framework fits into modern FPGA design flows oriented towards streaming and neural acceleration, extending them with a level of automation that allows for systematic evaluation of the trade-offs between model accuracy and hardware costs. Overall, while both works demonstrate the advantages of integrating AutoML techniques into FPGA design, the first stands out for its more general approach, focused on the co-optimization of MLP and hardware architecture, while the second favors domain-specific design, aimed at the efficient development of custom FPGA accelerators for high-performance RadioML applications.

Chapter 3

Model selection

This chapter describes the process of selecting the most suitable model for classifying a physiological dataset, as well as the preprocessing operations applied to the data. The dataset under consideration is a sports-related one, and the reference application scenario involves monitoring physical activities performed by multiple subjects. A real-world context consistent with this scenario could be represented by endurance competitions and individual disciplines, such as marathons, track and field championships, and triathlons.

The ability to monitor athlete performance through physiological signals such as heart rate, respiration rate, and breath frequency allows for a detailed analysis of individual performance. Particularly in long-distance competitions, continuous monitoring of physiological parameters can also serve a preventative purpose, serving as a safety support system for the timely identification of any signs of discomfort.

Based on these considerations, the selected model must ensure high accuracy in classifying the type of activity performed by subjects based on their physiological data. At the same time, the model's computational complexity must remain low, as the system is designed to operate in real time on data streams coming simultaneously from multiple individuals, ensuring efficient and scalable monitoring of a large number of athletes.

3.1 Dataset

The dataset used for this thesis contains real-time monitoring data collected from wearable sensors during physical activities. It includes measurements for Heart rate, Breath frequency, Respiration, ECG, R2R, stress index, HRVscore, position, phone longitude, phone latitude, phone altitude, acceleration X, acceleration Y and acceleration Z, all recorded at 1-second intervals for a total of 554.751 records.

Types of measurements that are in the Dataset:

- **Timestamp**: The date and time of the reading (in the format YYYY-MM-DD HH:MM:SS). Data is recorded at a frequency of one second.
- **Heart Rate** (num): The heart rate measured in beats per minute. Values range from 60 to 180 bpm.
- **Breath frequency** (num): The number of breaths taken per minute, ranging from 12 to 30 bpm.
- **Respiration** (list): The value of Respiration in a list format
- **AccelerationX, AccelerationY, AccelerationZ**: Numerical values representing acceleration in three directions. Ranges are (-0.90, 0.44) X-axis, (-0.97, 1.08) Y-axis.
- **Position, phone longitude, phone latitude, phone altitude**: Numeric values representing the position of the device.
- **Stress index** (num): Value representing fatigue, there are few records for this feature.
- **HrvScore** (num): Value representing the state of recovery and stress, there are few records for this feature.
- **Ecg** (list): Electrocardiogram data (numeric values).
- **R2R**: Numeric values related to the Record-Record interval of the ECG.
- **Label**: Nominal value associated with each row (numerical values): 1-Rest; 2-Walking on plane; 3-Stairs; 4-Downhill; 5-Walking uphill.

The dataset is made up of multiple files, and in these files, there is the data of four people. For each person, approximately three files were generated for a total of 10 files in NDJSON (Newline Delimited JSON) format, where: each line contains a separate and complete JSON object; the objects are not enclosed in an array [ ]; there are no commas between the objects and each object is separated by a newline. A visual example of this type of format can be seen in Figure 3.1.

This type of formatting is given by the fact that these files were exported from a MongoDB database, but this formatting will have to be changed, as it makes preprocessing difficult and, in general, is too complex to use the dataset. The amount of data at the end of the processing will be much smaller, as in NDJSON format, each record represents a value of a single column, so the number of records recorded is much larger than in tabular format data.

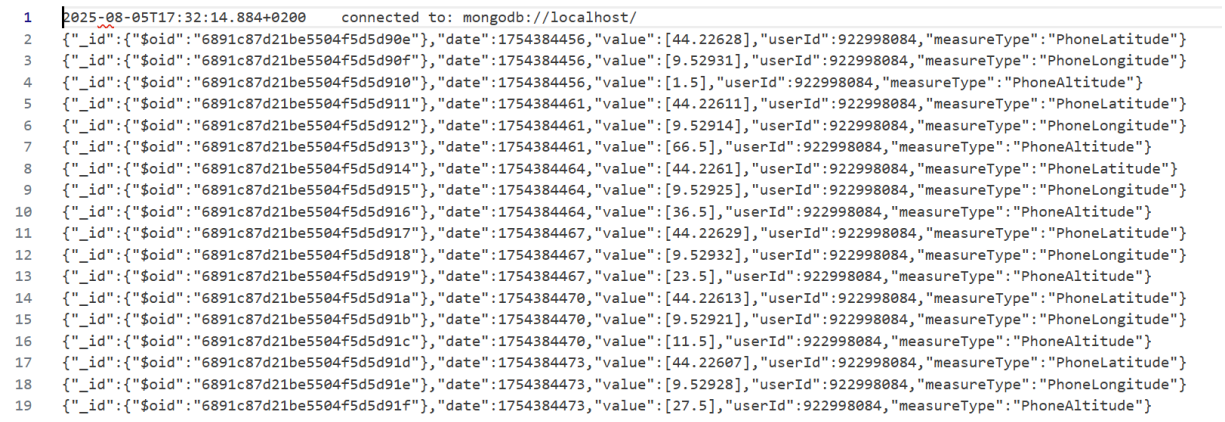


Figure 3.1: The raw data in NDJSON format

3.2 Preprocessing

This dataset is not suitable for use in the subsequent steps as it presents several issues that must first be resolved: one of these is the presence of records not related to JSON objects, which prevents the file from being read in JSON format, making it difficult to read the files. These non-readable lines represent log flags written during data collection by the sensors. Another problem is the presence of data in NDJSON format, which could generate difficulties in the subsequent steps; therefore, a conversion of the data into tabular format or dataframe format is necessary. Another issue is the presence of null values and duplicate features that need to be handled and removed. Another feature of this data is that it is divided into 10 separate files. To improve the usability of this data, it is necessary to merge it into a single file.

To correct the first problem within the data files—the presence of records not in JSON format —regular expressions were used because the files to be analyzed are large, the presence of records generating errors is unknown and the use of standard Python parsers is not possible, as each error case would have to be encoded. Using regular expressions (regex) to clean text files is an efficient and versatile solution, more advantageous than using Python libraries dedicated to data manipulation. Using regex allows you to operate directly on strings, identifying complex text patterns with just a few lines of code. This approach reduces the need for loops, conditions and structured parsing, speeding up processing. An example of these records can be seen in Figure 3.2.

Regular expressions, commonly called regexes, are a powerful tool for searching, recognizing and manipulating text patterns within strings or files. Essentially, a regex is a sequence of symbols and special characters that defines a text pattern. This pattern allows you to quickly identify portions of text that match a certain structure, in this case such as incorrect records that have the timestamp_FLAG_iteration pattern.

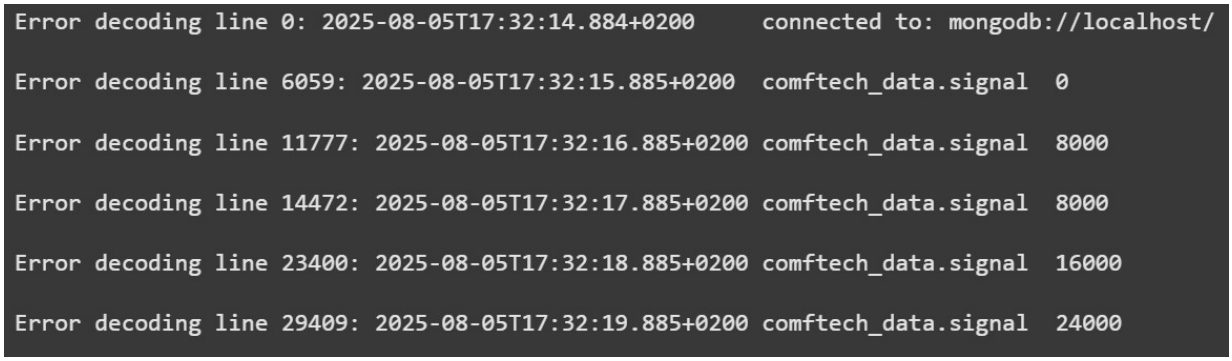


Figure 3.2: Example of records that generate problems when parsing data files

Regexes are highly flexible, allowing a single expression to address hundreds of different scenarios and adapt to various data formats. With only a few characters, you can identify and remove all lines beginning with a timestamp pattern or automatically replace problematic characters that are incompatible with formats like CSV or JSON, even in large files. In contrast to libraries such as Pandas or csv, which require loading data into larger structures like DataFrames or lists, regexes operate instantly and directly on the file, regardless of its size. Additionally, regexes are format-agnostic: they can process logs, semi-structured data, or plain text files without any preliminary conversion.

The files were cleaned using regular expressions in VSCode. Two patterns were created that allow for complete cleanup of the 10 files $\hat{}.*comftech.*\$n?$ and $\hat{}.*(?:exported|mongodb).*\$n?$

The first regular expression,$\hat{}.*comftech.*\$n?$, is a pattern to find and remove any line of text containing the word "comftech." The symbol $\wedge$ indicates the beginning of the line: it starts the search from the beginning of each line, not just the beginning of the file. Immediately after it, we find .*, which means "any sequence of characters, even empty": the dot . represents a single character (excluding a newline), while the asterisk * allows for unlimited repetition. In this position, it captures everything before the word "comftech." The term comftech is the literal string the pattern searches for: the regex only matches lines containing it. After it, .* appears again, which in this case captures everything after the word on the same line, up to the $ symbol, which represents the end of the line. Adding n? also includes any newline character, thus removing the line completely, leaving no blank spaces in the resulting text.

Instead, the second regular expression, $\hat{}.*(?:exported|mongodb).*\$n?$ finds and handles all rows that contain references to MongoDB and the Flag that signals the export, making it a regex similar to the previous one but which handles two different literal strings, allowing filtering of references to the MongoDB database and the export. At the end of the data cleanup, **743** error records were removed, leaving **554,008** useful data records.

Once the 10 files were filtered, it was possible to read them with the Pandas functions for reading JSON files as there were no longer any parsing errors. During data analysis, the second problem was encountered, the formatting of the data in NDJSON format: each line contains a separate and complete JSON object; the objects are not enclosed in an array [ ]; there are no commas between the objects; each object is separated by a newline (line break). This type of format generates problems during conversion to a dataframe as standard JSON parsers expect a single valid JSON object or an array of objects. NDJSON, on the other hand, contains multiple JSON objects separated by newlines, violating standard JSON syntax. When attempting to read the file with json.load() or pd.read_json() by default, the parser immediately fails on the second object because it finds neither a comma nor a closing parenthesis. This complicates the usability of the data in subsequent steps. so it was necessary to change the formatting to tabular format, the formatting change was done on Colab (Python).

The first step in changing the formatting was to read the files line by line and save them as a list. The list, after reading the 10 files, contains 554,008 elements. The data list was then converted into a Pandas dataframe with the following format: id, date, value, userId, measureType. An example of the following format can be seen in the Figure 3.3

	_id	date	value	userId	measureType
0	{'$oid': '6891c87d21be5504f5d5d90e'}	1754384456	[44.22628]	922998084	PhoneLatitude
1	{'$oid': '6891c87d21be5504f5d5d90f'}	1754384456	[9.52931]	922998084	PhoneLongitude
2	{'$oid': '6891c87d21be5504f5d5d910'}	1754384456	[1.5]	922998084	PhoneAltitude
3	{'$oid': '6891c87d21be5504f5d5d911'}	1754384461	[44.22611]	922998084	PhoneLatitude
4	{'$oid': '6891c87d21be5504f5d5d912'}	1754384461	[9.52914]	922998084	PhoneLongitude
...	...	...	...	...	...
73195	{'$oid': '68921d6cff969188b1d639fd'}	1754406186	[9.5292]	922998084	PhoneLongitude
73196	{'$oid': '68921d6cff969188b1d639fe'}	1754406186	[66.5]	922998084	PhoneAltitude
73197	{'$oid': '68921d6cff969188b1d639ff'}	1754406187	[44.22631]	922998084	PhoneLatitude
73198	{'$oid': '68921d6cff969188b1d63a00'}	1754406187	[9.5292]	922998084	PhoneLongitude
73199	{'$oid': '68921d6cff969188b1d63a01'}	1754406187	[66.5]	922998084	PhoneAltitude

Figure 3.3: An example of the data in a Pandas after the first parsing.

Finally, the second step after converting the list into a dataframe was to use the Pandas pivot_table() function, which is used to reorganize tabular data, creating a spreadsheet-style pivot table. A visual example is shown in Figure 3.4. It allows you to aggregate

numeric values based on one or more indexes and columns. pivot_table() was useful because it transforms large datasets into a more readable tabular structure without modifying the original data. The specific instruction used was: *dataframe.pivot_table(index='date', columns='measureType', values='value', aggfunc='mean').*

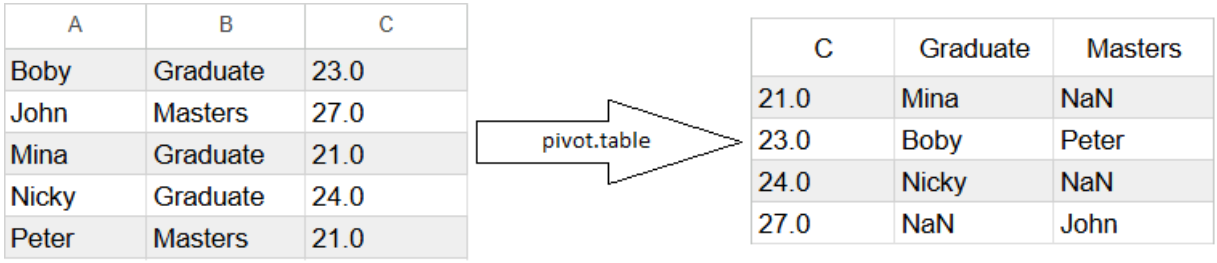


Figure 3.4: The table on the left is the original table, while the table on the right has been pivoted. The values in column B are converted to columns in the right table, and the values in column A are then moved to the appropriate columns in the right table.

The index of the pivoted dataframe is the timestamp (the moment the values were acquired), the various columns are represented by the type of measurements and the values of the columns are the individual measured values. The aggfunc option (aggregation function) was added and set to calculate the average, but it was not used, as there were no values collected at the same time. Once this step was carried out, a dataframe with **39,572** rows of data was obtained, with 14 columns representing the features ['PhoneLatitude', 'PhoneLongitude', 'PhoneAltitude', 'Ecg', 'Respiration', 'BreathFrequency', 'Position', 'R2R', 'HeartRate', 'AccelerationX', 'AccelerationY', 'AccelerationZ', 'StressIndex', 'HrvScore']. An example of the pivoted dataframe is shown in Figure 3.5

measureType date	AccelerationX	AccelerationY	AccelerationZ	BreathFrequency	Ecg	HeartRate	R2R
2025-08-05 09:10:46	0.029844	0.989688	0.237031	14.0	-0.483635	74.0	821.0
2025-08-05 09:10:48	0.030625	0.976250	0.290781	14.0	-0.487760	75.0	800.0
2025-08-05 09:10:51	0.027344	0.983906	0.264531	13.0	-0.509039	75.0	789.0
2025-08-05 09:10:53	0.024688	0.973125	0.302031	15.0	-0.451592	75.0	784.0
2025-08-05 09:10:55	0.032500	0.969063	0.317812	17.0	-0.495523	76.0	779.0

Figure 3.5: An example of the pivoted dataframe.

After the dataset was correctly pivoted, in tabular dataframe format, the actual data analysis was performed by choosing which data columns to keep and which columns to remove. Finally, the relationship between the chosen features was visualized with Weka.

The HrvScore and StressIndex columns have been removed because out of 39,572 rows of data there were only 30 values per column, making these features insufficient. The presence of little data could be generated by the sensor acquisition which could have been done incorrectly.

The date column has been removed as it is not needed for creating the templates in the next step.

The Position, phone longitude, phone latitude and phone altitude columns have been removed because, firstly, they have not been measured uniformly and not all values are always present at the same time. Secondly, the rows of data containing all the values of these four columns are few, approximately 300. Lastly, these four features do not seem relevant for the type of models that are created in the next subsection.

Once the irrelevant columns were removed, the Pandas dropna() function was used to remove the rows containing null values. This is because, since this is real-time data, the models generated from this data will also have to handle real-time data and the presence of null values could cause the models to be trained incorrectly.

After this final filtering, the dataset has **18,422** rows and **8** columns of features: AccelerationX, AccelerationY, AccelerationZ, BreathFrequency, Ecg, HeartRate, R2R, Respiration and a column of labels: 1: Rest, 2: Walking on plane, 3: Stairs, 4: Downhill, 5: Walking uphill. Finally, the Weka data visualization tool was used to create plots of the data to show any relationships between them. The plot of the relationships is shown in Figure 3.6, while the plots of the distribution of each individual feature can be found at the end of this thesis in Figure 5.1.

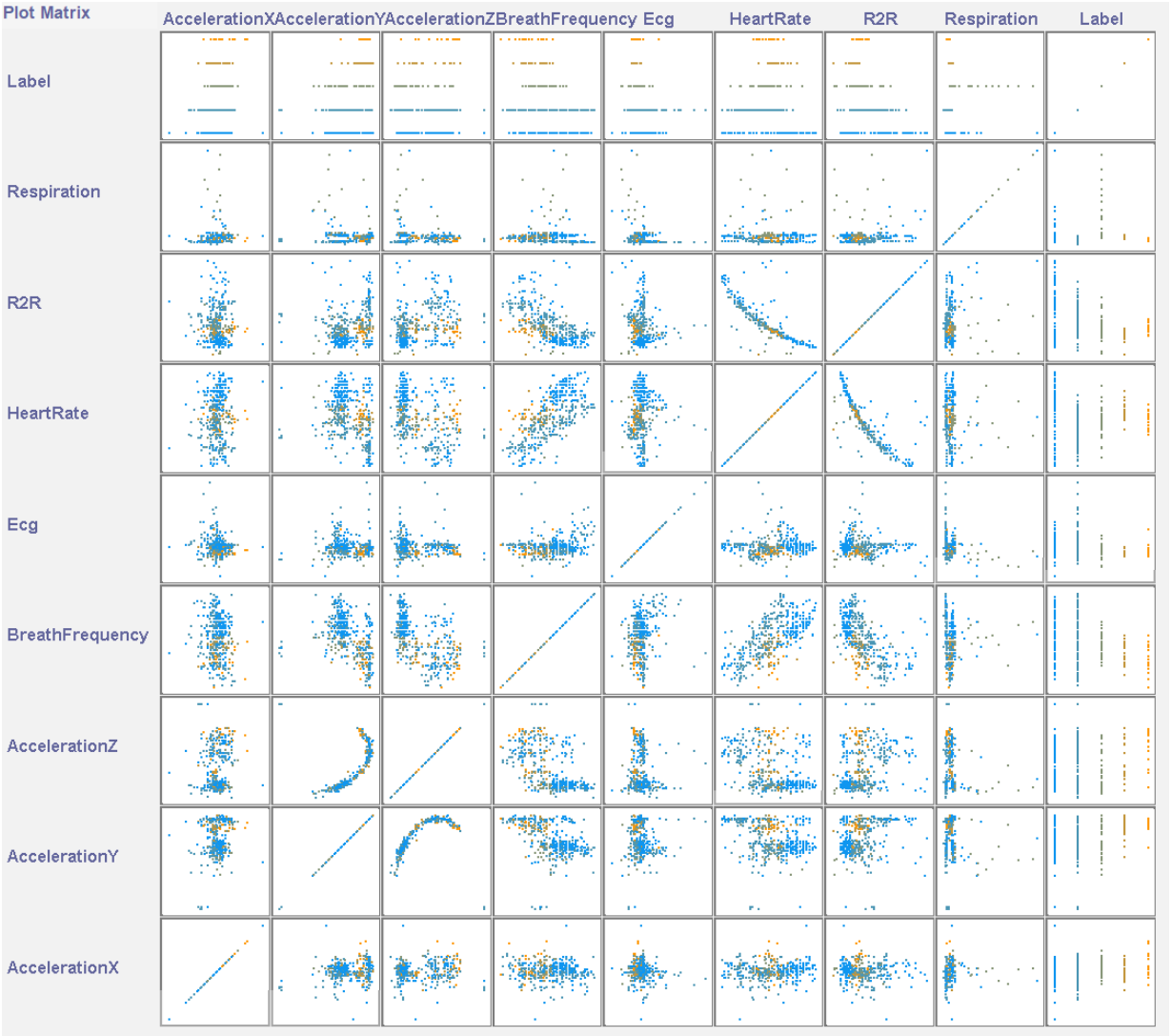


Figure 3.6: A plot of data relationships generated by Weka tool. This plot shows that many features have no relationship with each other, except for the Heart rate and R2R features which show a correlation and the acceleration Y and acceleration Z features which show another correlation.

3.3 AutoWeka

This section explains the work done to automate the selection of the classification model using AutoWeka, the choices made to obtain reliable results. AutoWeka's search operation can be divided into two main phases. In the first phase, AutoWeka constructs a large search space that includes most of the classification and regression algorithms available in WEKA.

In the second phase, AutoWeka uses the SMAC optimization algorithm [HHLB11], which implements a Bayesian optimization approach. SMAC builds a surrogate model (based on random forests) to approximate the relationship between the tested configurations and the achieved performance. It then sequentially selects the most promising configurations, balancing exploration and exploitation through an acquisition criterion, typically Expected Improvement (EI). Summarizing all the steps of the AutoWeka search process, we note that: AutoWeka defines the search space and time constraints using the parameters entered by the user; it selects an initial set of random configurations based on the seed entered by the user, example in Figure 3.7; trains and evaluates the models through cross-validation, using an 80% of the dataset for training and the other 20% to validate; updates the surrogate model with the results obtained by training and validating other models; selects new configurations based on the optimization criterion; AutoWeka repeats the previous steps cyclically until the computational budget is exhausted, which can be given by the memory limit or time limit. Once a certain budget limit is reached, AutoWeka outputs the best configuration found.

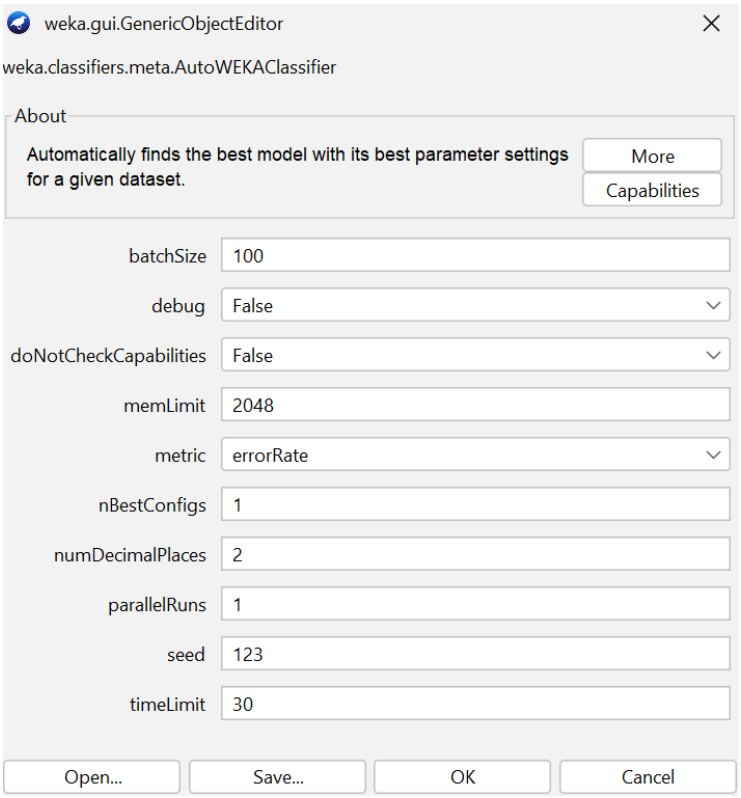


Figure 3.7: This image shows the GUI screen for entering AutoWeka search parameters which are: batchSize; debug; doNotCheckCapabilities; memLimit; metric; nBestConfig; seed and timeLimit.

To search a suitable classification model for the sensor-recorded physiological dataset, as explained in the Section 3.1, 10 distinct searches were conducted using AutoWeka. The decision to perform 10 different searches was made to mitigate the risk of inaccurate results and to see if the results of 10 separate searches lead to the same result. The parameters were set more or less similar for all 10 searches: the batchSize was set to 100 as it is a standard value for many algorithms that make predictions with a specific number of batches; the debug parameter was set to False as it generates additional unnecessary information in the output, slowing down the search times; the doNotCheckCapabilities parameter was set to False, as if it were True, AutoWeka would not check the capabilities of the classifiers before building them, in some cases leading to testing models that are initially worse, slowing down the search time; the memLimit parameter was set to 2048MB to leave an adequate memory size for each single search; the metric parameter was decided to use the errorRate metric as, among all the metrics present, it is the one that best suits the type of work and the dataset, looking for the models that have the lowest error rate and therefore looking for the models that have a greater accuracy in the prediction; The nBestConfig parameter was set to 1 to output only the best model found; The only parameters that were changed for the 10 searches were seed and timeLimit. A different seed was used for each search, as it affects the search initialization. The timeLimit was set to 30 min for 4 searches, 60 min for 4 searches and 120 min for 2 searches. This difference in the search time limit was necessary to understand how the AutoWeka results change as the search time varies.

Once a search is finished, AutoWeka returns as output a txt file having all the necessary information such as: the best model found based on the search, together with the hyper parameters related to the found model; a list of instructions to be able to recreate the same model found in Java using the Weka package. Unfortunately this feature of AutoWeka has not been used since the next step of porting the model to the FPGA must be done in C++.

Moreover, it provides further information on the performance of the model, such as:

- Correctly Classified Instances;
- Incorrectly Classified Instances;
- Kappa statistic;
- Mean absolute error;
- Root mean squared error;
- Relative absolute error;
- Root relative squared error;

- Total Number of Instances;
- Confusion Matrix;
- Detailed Accuracy By Class.

Before analyzing the results of the 10 searches, it's worth mentioning SMAC, even though it falls outside the scope of this thesis and the SMAC score, to better understand the research results and the subsequent decisions. Sequential Model-based Algorithm Configuration (SMAC) is a Bayesian optimization method developed to address complex problems of automatic configuration of machine learning algorithms and models [HHLB11]. Its primary goal is to efficiently identify the optimal combination of an algorithm's hyperparameters, reducing the number of experiments required compared to exhaustive methods like grid or random search [BB12]. SMAC belongs to the family of Model-Based Optimization (MBO) methods, which build a probabilistic surrogate model to approximate the relationship between tested configurations and their observed performance. Instead of evaluating all possible configurations, SMAC learns a predictive function that estimates the expected quality of new configurations and guides the search sequentially, balancing the exploration of untested regions with the exploitation of promising ones [HHLB11] [SLA12]. Precisely because of the probabilistic model construction, it was decided to perform 10 searches, as different initializations can affect the optimizer's results. Unlike other Bayesian optimizers that use Gaussian Processes, SMAC uses random forests as the surrogate model. This allows it to handle large search spaces, categorical parameters and noisy functions more efficiently. [HKV19]. At each iteration, SMAC updates the surrogate model using results from previous evaluations and selects the next configuration to test by maximizing an acquisition function, typically the Expected Improvement (EI), which quantifies the expected gain in performance.

The term SMAC score generally refers to the performance rating the system assigns to a hyperparameter configuration during the optimization process. In practice, the SMAC score represents the estimated (or observed) value of the objective function, such as classification error, accuracy, precision and recall, obtained for a given configuration. SMAC scores range from 0 to 1, with lower values generally indicating better models.

The surrogate model then uses this score to update the predictive distribution and guide the search towards areas of the parameter space with more promising scores. In applications such as AutoWeka [THHLB13], the SMAC score serves as the reference metric for comparing different combinations of algorithms and hyperparameters, providing a quantitative guide for the entire sequential learning process and enabling progressive, adaptive improvement of the tested configurations.

SMAC has been successfully applied in various domains, including automated machine learning (AUTOML), automatic scheduling, combinatorial optimization and solver tuning

for NP-hard problems [HHLB14]. Its efficiency stems from its ability to iteratively learn from previous evaluations and reduce the total computational cost of optimization while maintaining a high probability of converging to optimal configurations.

The AutoWeka results at the end of the 10 searches were not entirely uniform, since not all searches led to the same model: Multinomial Logistic Regression was suggested 7 times, AdaBoost twice and RandomForest once. Specifically, Multinomial Logistic Regression was selected in the longest searches: both 2-hour searches led to Multinomial Logistic Regression, three 1-hour searches led to Multinomial Logistic Regression and finally, two 30-minute searches resulted in Multinomial Logistic Regression. However, the searches that led to the choice of the AdaBoost and Random Forest algorithms were mainly the shortest ones and those with the worst SMAC scores. The results of the 10 searches are found in Table 3.1.

Table 3.1: The results of the 10 AutoWeka seraches,

Suggested algorithm	Time Limit	Error Rate	SMAC score
Random forest	30 min	0,023	0,108
Multinomial Logistic Regression	30 min	0,012	0,098
Multinomial Logistic Regression	30 min	0.016	0,099
AdaBoost	30 min	0,033	0,107
AdaBoost	1 hour	0,0010	0,024
Multinomial Logistic Regression	1 hour	0,0010	0,007
Multinomial Logistic Regression	1 hour	0,0009	0,016
Multinomial Logistic Regression	1 hour	0,0006	0,008
Multinomial Logistic Regression	2 hour	0,0	0,006
Multinomial Logistic Regression	2 hour	0,0	0,0059

Analyzing the results of the 10 searches provided by AutoWeka, Multinomial Logistic Regression was chosen as the classifier algorithm for the physiological dataset, since it presents three valid reasons: first, it was recommended by AutoWeka seven times out of ten; second, the search with the best SMAC Score and the best estimated error rate presented the use of the Multinomial Logistic Regression algorithm as a model; third, Multinomial Logistic Regression is a reasonably robust and lightweight model compared to the other two suggested algorithms and seems more suitable for classifying the provided dataset. Therefore, based on these reasons, Multinomial Logistic Regression was used as a classifier and the models trained in this phase were used in the next step to port them into FPGAs.

3.4 Multinomial Logistic Regression

Multinomial Logistic Regression is a classification technique that extends the logistic regression algorithm to handle multiclass outcomes, given one or more independent variables. Multinomial Logistic Regression is similar to logistic regression, but with the difference that the target variable can have more than two classes, i.e., multiclass. For example, in this work, the labels of the physical activities are the dependent variables "Rest", "Walking on plane" and "Stairs", which are multiclass and the independent variables are represented by the physiological acquisitions "Heart rate", "Respiration" and "ECG". The Multinomial Logistic regression model is used to predict the probabilities of a categorically dependent variable with two or more possible outcome classes. Whereas the logistic regression model is used when the dependent categorical variable has two outcome classes, for example, students can either "Pass" or "Fail" in an exam, or a bank manager can either "Grant" or "Reject" the loan for a person. Multinomial Logistic Regression is also known as multiclass logistic regression, softmax regression, polytomous logistic regression, multinomial logit, maximum entropy (MaxEnt) classifier and conditional maximum entropy model.

When you want to use multinomial logistic regression as the classification algorithm, the data should satisfy some of the assumptions required for multinomial logistic regression to work. In this work, the chosen dataset satisfies them all. The assumptions to be satisfied are:

- the Dependent variable should be either a nominal or an ordinal. A nominal variable is a variable that has two or more categories, but it does not have any meaningful ordering in them. Ordinal variables are variables that can also have two or more categories, but they can be ordered or ranked among themselves. For example, in the dataset used for this work, the labels are Nominal;
- a set of one or more Independent variables had to be continuous, ordinal, or nominal. Continuous variables are numeric variables that can have an infinite number of values within the specified range. Ordinal variables should be treated as either continuous or nominal. In the dataset, all independent variables are continuous, as they are physiological measurements of a person's state acquired by sensors;
- the dependent variables must be mutually exclusive and exhaustive. Mutually exclusive means that when there are two or more categories, no observation falls into more than one category of the dependent variable. The categories are exhaustive, which means that every observation must fall into some category of the dependent variable. In the dataset, the categories of the Labels are mutually exclusive and exhaustive, since a person can only be resting, walking, or doing Stairs in a specific observation;
- Multicollinearity between Independent variables. Multicollinearity occurs when two or more independent variables are highly correlated with each other. This makes it

difficult to determine how much each independent variable contributes to the dependent variable's category. Also, it makes it difficult to understand the importance of different variables. To satisfy this assumption during the preprocessing phase of the dataset, the correlated variables were removed and using the Weka data visualization tool, it was verified that the remaining independent variables are not correlated among them.

The simplest way to explain Multinomial Logistic regression is to view it as k models, one for each class, built as a set of independent binomial logistic regressions. For Example, let us consider three classes in the nominal dependent variable A, B and C. Firstly, it is necessary to build three different models doing one category vs the other categories: Class A vs Class B & C, Class B vs Class A & C and Class C vs Class A & B, obtaining three separate Logistic regression models. In the first model, (Class A vs Class B & C): Class A will be 1 and Class B&C will be 0. In the second model (Class B vs Class A & C): Class B will be 1 and Class A&C will be 0 and in the third model (Class C vs Class A & B): Class C will be 1 and Class A&B will be 0. Next is needed to develop the equation to calculate three Probabilities, P(A), P(B) and P(C), very similar to the logistic regression equation. Predicting the class of any observations, based on the independent input variables, will be the class that has the highest probability. If P(A) > P(B) and P(A) > P(C), then the dependent target class = Class A.

In binary logistic regression, the probability of positive class membership is modeled using the logistic function [HJLS13]. In the multinomial case, the response variable y can assume K distinct values 1, 2,..., K. For each category k, a linear function is defined:

$$z_k = \beta_{k0} + \beta_k^T x$$

and the probability of belonging to class k is given by the softmax function [Mur12]:

$$P(y = k|x) = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}$$

The softmax function generalizes the univariate logistic function, ensuring that the probabilities lie between 0 and 1 and that their sum is equal to 1 [BN06]. The model also assumes that the logarithm of the odds ratio (log-odds) between each class k and a reference class K is a linear function of the predictors [McC19]:

$$log(\frac{P(y = k|x)}{P(y = K|x)}) = \beta_{k0} + \beta_k^T x$$

The choice of the reference class helps avoid parameter identifiability problems, since the probabilities are constrained to sum to 1.

The coefficients β_{kj} represent the marginal effect of the variable x_j on the log-odds of belonging to class k compared to the reference class [Agr17]. A unit increase in x_j, therefore, changes the log-odds by β_{kj}, holding other variables constant. This property makes MLR particularly useful when an interpretable and quantitatively explicit model is desired.

Multinomial logistic regression training consists of estimating the β_k parameters that maximize the likelihood function of the data [HJLS13]:

$$l(\beta) = \sum_{i=1}^{N} \sum_{k=1}^{K} 1(y_i = k) log P(y_i = k|x_i)$$

The learning process is equivalent to maximizing the log-likelihood, or, in comparable terms, minimizing the cross-entropy [BN06][Mur12]: $J(\beta) = -l(\beta)$.

Since the likelihood function does not have a closed analytical solution for K>2, iterative methods are used to optimize the parameters. The main algorithms include:

- **Gradient Descent**, which updates parameters in the opposite direction to the gradient [Mur12];
- **Stochastic Gradient Descent (SGD)**, which improves computational efficiency by using mini-batches of data [PVG+11];
- **Newton-Raphson or IRLS** (Iteratively Reweighted Least Squares), which exploits second-order information [McC19].

The gradient of the loss function with respect to the parameters of class k is:

$$\nabla_{\beta_k} J(\beta) = -\sum_{i=1}^{N} [1(y_i = k) - P(y_i = k|x_i)] x_i$$

The iterative process continues until convergence, that is, until the variation in the cost function or parameters is negligible.

3.5 MLR implementation

Unfortunately, the Multinomial Logistic Regression model trained by AutoWeka cannot be used in the porting step, as it is written in Java and AutoWeka does not provide the model's coefficients and biases to recreate the predictor in C++. The need to use C++ to write the model stems from the fact that the tool for porting the model to the FPGA only supports C++, making it necessary to retrain the Multinomial Logistic Regression model in an

environment that allows extraction of the model's coefficients and biases. To perform the new training of the Multinomial logistic regression, it was decided to use the sklearn library in Python, since it is one of the few libraries that provides an optimized implementation of Multinomial Logistic regression and allows extracting the model's coefficients and biases simply.

Using sklearn's logistcregression function and the physiological dataset, three similar models were trained. The decision to train three models was made in anticipation of the next step: porting the models to the FPGA board, as models of different complexities are necessary to test the FPGA board's performance as complexity changes. The first MLG model was trained on all features of the physiological dataset, aiming to predict the type of activity a person is performing during a given observation. The features are: heart rate, breath frequency, respiration, ECG, R2R, acceleration X, acceleration Y and acceleration Z. At the end of the training, the following coefficients and biases were found:

HR	BF	RES	ECG	R2R	acc X	acc Y	acc Z	biases
-0.1307	0.2265	-0.0559	0.22214	2.0489	1.1138	0.2243	4.89936e-05	-277.77
-0.4122	-0.3357	-0.3839	0.1693	1.2017	-0.740	-0.0524	-0.00018	164.22
0.5987	-0.2875	0.5608	0.1859	0.1236	-0.0744	-0.0295	7.75729e-05	-15.489
-0.2175	0.3562	-0.02534	-0.292	-4.805	-0.01215	-0.079	2.60834e-05	32.194
0.1618	0.0406	-0.0956	-0.284	1.430	-0.287	-0.0624	-8.27049e-05	96.846

Table 3.2: In this table there are the coefficients and the biases found for the MLR model with 8 features. In each row there are coefficients for each class, in the columns there are coefficients for each feature and in the last column there are biases for each class.

The second model was trained on only five features from the dataset: heart rate, breath frequency, respiration, ECG and R2R. Excluding the Acceleration features was primarily to create a lighter model, as the discarded features did not significantly affect the predictive model's performance. The found coefficients and biases are shown in Table 3.3.

HR	BF	RES	ECG	R2R	biases
0.220051	2.04891	1.11017	0.223619	0.000117199	-276.6576
0.174333	1.23153	-0.734954	-0.0521767	-0.000118635	163.0590
0.181139	0.130759	-0.0764211	-0.0295454	0.00014559	-15.4323
-0.291914	-4.81771	-0.0132671	-0.0797118	9.41345e-05	32.5635
-0.28361	1.40651	-0.285523	-0.062185	-1.42141e-05	96.4673

Table 3.3: In this table are presented the coefficients and the biases found for the model with 5 features. In each row there are coefficients for each class, in the columns there are coefficients for each feature and in the last column there are biases for each class.

Finally, the third model was trained on only three features of the dataset: heart rate, breath frequency and respiration. Excluding the Acceleration and ECG features, the decision to train this model was made solely to obtain a very lightweight predictive model and to see how it affected the FPGA board's performance. This last model has poorer prediction capabilities than the previous two, making it a worse predictive model than the two more complex ones. At the end of training, the coefficients and biases listed in Table 3.4 were found.

HR	BF	RES	biases
0.0509846	0.391408	0.0798768	-89.31607443
0.212564	-0.169964	-0.00882437	19.25098301
0.0878693	-0.139796	-0.0182498	23.97523702
-0.174342	-0.0178114	-0.0283784	21.33804296
-0.177076	-0.0638371	-0.0244242	24.75181145

Table 3.4: In this table are presented the coefficients and the biases found for the model with 3 features. In each row there are coefficients for each class, in the columns there are coefficients for each feature and in the last column there are biases for each class.

All three prediction models were trained on 80% of the dataset and evaluated on the remaining 20% using standard sklearn functions such as fit() and score(). The validation results can be found in Table 3.5.

MLR	Score
8 features	0.9901
5 features	0.9902
3 features	0.8217

Table 3.5: In this table are presented the validation scores of the three Multinomial Logistic Regression models

Once the coefficients and biases for the three models trained in Python were obtained, three predictive models were created in separate C++ files. The code was written with the understanding that the Vitis HLS porting tool does not recognize non-standard C++ libraries, so no external functions were used to create the models. The code for the three models is very similar; an example code:

```
#include <array>
#include <iostream>
#include <cmath>
#include <cstddef>
#include <algorithm>
#include <chrono>
```

```
7
8  // Predict 5 outputs from 3 inputs.
9  int predict_model(const std::array<double,5>& x) {
10     double coeffs[5][5] = {
11         {2.20051328e-01,  2.04890955e+00,  1.11016533e+00,  2.23618979e
       -01, 1.17199286e-04},
12         {1.74333332e-01,  1.23152727e+00, -7.34954500e-01, -5.21766994e
       -02, -1.18635132e-04},
13         { 1.81139182e-01,  1.30759400e-01, -7.64210852e-02, -2.95453966e
       -02, 1.45589518e-04},
14         {-2.91914025e-01, -4.81770505e+00, -1.32671175e-02, -7.97117767e
       -02, 9.41345455e-05},
15         {-2.83609800e-01,  1.40650893e+00, -2.85522588e-01, -6.21850304e
       -02, -1.42141364e-05}
16     };
17     double biases[5] = {
18         -276.65764851,
19          163.05908344,
20          -15.43233559,
21          32.56356667,
22          96.46733399
23     };
24
25     std::array<double,5> y;
26     for (std::size_t i = 0; i < 5; ++i) {
27         y[i] = coeffs[i][0]*x[0] + coeffs[i][1]*x[1] + coeffs[i][2]*x[2]
       + coeffs[i][3]*x[3] + coeffs[i][4]*x[4] + biases[i];
28     }
29
30     double Z = 0.0;
31     for (std::size_t i = 0; i < 5; ++i) {
32         y[i] = std::exp(y[i]);
33         Z += y[i];
34     }
35
36     for (std::size_t i = 0; i < 5; ++i) {
37         y[i] /= Z;
38     }
39     auto it = std::max_element(y.begin(), y.end());
40     auto idx = std::distance(y.begin(), it);
41
42     return idx+1;
43 }
```

Listing 3.1: Code example

The variable coeffs stores all the coefficients of the predictive model. Each of the 5 rows contains the coefficients for each category and each coefficient in the row represents a feature. The variable biases contains the bias values for each category. The variable x,

passed as an attribute, includes the feature values of a single observation. Lines 25-28 of the code multiply the input feature values with the corresponding feature coefficients and finally add the bias, representing the linear function $z_k = \beta_{k0} + \beta_k^T x$ for each category k. Lines 30-38 calculate the probabilities of belonging to category k. Lines 39-40 search for the maximum element of an array, as the probability of belonging to a class k is the most significant probability compared to the remaining K categories. Once the maximum element of the array is found, the array index is returned, as it represents the category.

MLG	Averge speed(ns)
8 features	433
5 features	415
3 features	298

Table 3.6: In this table are presented the average prediction speeds in nanoseconds of the three Multinomial Logistic Regression models written in C++.

After writing the three predictive models in C++, tests were run to find the average prediction speed of each model. The tests were performed to understand the performance of the three models executed sequentially within the CPU, as these speed tests are necessary for comparison with their optimized versions after the porting step. The tests were performed using the entire dataset of 18,422 observations, measuring the prediction times for each observation's category and finally averaging all the estimated times. Since the three models already have high performance, the measurements were taken in nanoseconds. The test results are found in Table 3.6.

3.6 What if AutoWeka hadn't been used?

Without using AutoWeka to automatically select the classification algorithm applied to the physiological dataset, the choice would have fallen on the Multiclass Perceptron. This model is characterized by a relatively simple structure and a limited number of parameters, which facilitates its interpretability and computational management. Furthermore, since it does not produce probabilistic estimates but bases the decision on linear scores and the argmax operator, it is less sensitive to probability calibration issues. Its limited computational complexity makes it particularly suitable for application contexts that require real-time classification, especially when efficiency and processing speed are constraints.

Multiclass Perceptron is a linear classification algorithm designed for supervised problems where the label belongs to a finite set of K classes. Given a dataset of pairs (x_i, y_i), with $x_i \in R^d$ and $y_i \in \{1, ..., K\}$, the model associates a weight vector with each class, organized in a matrix $W \in R^{K \times d}$. The prediction is obtained by calculating, for each class

k, a linear score $f_k(x) = w_k \cdot x$ and selecting the class with the highest score using the argmax operator. The learning process is online: for each example, the predicted class is compared with the actual one and, in case of an error, the weight vector of the correct class is updated by increasing it by ηx_i and that of the incorrectly predicted class by decreasing it by the same amount, where η is the learning rate. This mechanism increases the margin between the classes involved in the error. The algorithm converges in a finite number of steps if the data are linearly separable, otherwise it may not stabilize.

Chapter 4

Porting the models on FPGA using Vitis HLS

This chapter explores and explains the work done to port the three Multinomial Logistic Regression models found in the previous chapter to the AMD/Xilinx Artix™ UltraScale+ XCAU20P-2SFVB784I FPGA board. The porting process was carried out using the Vitis tools to synthesize RTL (Register Transfer Level) code from C++ code and Vivado to transfer the RTL code into the simulator. Before explaining the work done, it is necessary to spend a few words on the FPGA board used, Vitis and Vivado.

4.1 FPGA

The XCAU20P-2SFVB784I FPGA board, a member of the AMD/Xilinx Artix UltraScale+ family, is classified as a mid-to-high-end device. It is optimized for applications that require both cost efficiency and high performance [Xil25a]. The XCAU20P-2SFVB784I designation specifies the model and its primary construction features. The XCAU prefix denotes membership in the Artix UltraScale+ family. The 20P acronym indicates the device size, with approximately 20,000 effective logic cells. The -2 suffix indicates the speed grade, which represents the timing performance class. Generally, a lower number indicates better timing performance. SFVB784I describes the package type, a 784-pin BGA and the industrial operating temperature range from -40 °C to +100 °C [Xil25c]. The Artix UltraScale+ series constitutes the low-power, low-cost segment of the UltraScale+ platform. It is intended for applications that require a balance among logic density, power efficiency and processing power. Although positioned below the Kintex and Virtex series, it shares significant architectural features with them, such as enhanced DSP blocks, a distributed clock network and high-speed serial transceivers [Xil24].

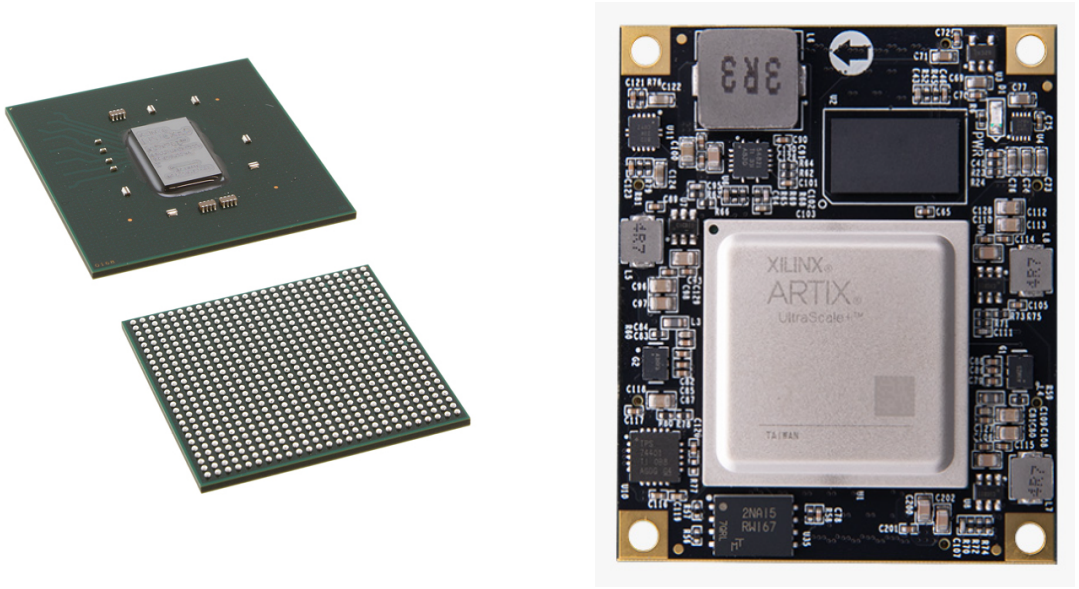


Figure 4.1: A In this figure, there are images of the FPGA board on the right and an image of the FPGA outside the case on the left.

The XCAU20P device contains approximately 238,000 programmable logic elements (LEs), which are organized into Configurable Logic Blocks (CLBs) composed of Look-Up Tables (LUTs) and associated Flip-Flops. This structure supports the synthesis of high-performance combinatorial and sequential logic, configurable through an externally stored configuration bitstream. The device also includes about 13,625 adaptive logic blocks (ALMs or LABs), providing sufficient processing capacity for projects of small to medium complexity. Onboard memory comprises both Block RAM (BRAM) and distributed memory, with an estimated total capacity of 3.3 Mbit [Ele24]. This combination of memory resources enables the implementation of data buffers, pipelines and state memories directly on the logic die, thereby reducing access latency relative to external memory solutions. BRAMs are typically organized into 18 KB or 36 KB blocks, support single-port, dual-port and true dual-port configurations and include integrated Error Correction Code (ECC) capabilities. The device also incorporates numerous DSP slices, specialized arithmetic blocks for multiply-accumulate operations. These are essential for applications in signal processing, machine learning and computer vision. Although the exact number of DSP slices depends on the internal configuration, the UltraScale+ family supports both fixed-point and floating-point operations and is optimized for high-performance pipelines [Xil25c]. Connectivity is a critical feature of modern FPGAs. The XCAU20P-2SFVB784I supports up to 240 user I/O lines, configurable for various logic standards, including LVCMOS, LVDS, SSTL and HSTL, with voltage levels from 1.2 to 3.3 V.[Xil24].

The "-2" suffix in the model number designates the speed class, or speed grade. The -2 speed grade offers an optimal balance between performance and power consumption, making the device suitable for embedded computing, intelligent sensors and real-time control systems [Xil25c].

The selection of the FPGA in question is particularly consistent with the application context outlined in Section 3. Its compact size favors its use in operational scenarios characterized by limited accessibility or stringent logistical constraints, in which the use of larger equipment would be impractical or inefficient. Another reason behind this choice is the -2 speed rating, which allows for the efficient management of applications with real-time processing requirements. This feature ensures high computational capacity while maintaining significantly lower energy consumption than traditional CPU-based solutions. This combination of performance and energy efficiency makes the platform particularly suitable for embedded systems intended to operate in dynamic and distributed environments.

4.2 Vitis HLS

AMD Vitis is a unified software platform for embedded systems design, including System-on-Chip (SoC), Field-Programmable Gate Array (FPGA) and AI Engine (AIE) architectures. Developed by Xilinx, now part of AMD, Vitis democratizes access to adaptable hardware by allowing software engineers, system architects and researchers to utilize hardware acceleration without requiring expertise in low-level Register Transfer Level (RTL) design [New19].

The Register Transfer Level represents a fundamental level of abstraction in digital circuit design, in which the behavior of a synchronous system is described in terms of data transfer between registers and the logical/arithmetic operations applied to that data at successive clock cycles. Unlike lower levels of description, RTL does not focus on transistors or individual logic gates, but rather on the organization of data flow between memory elements and the transformations performed by the intervening combinational logic.

A typical RTL model is composed of three main elements. Registers are the memory elements responsible for holding data between one clock edge and the next, generally implemented using flip-flops. Combinational logic includes the set of logical and arithmetic functions that process data between the outputs of the source registers and the inputs of the destination registers. Finally, the clock is the synchronization signal that determines when the registers update their contents, a simple example is shown in Code 2.1. RTL descriptions are expressed using hardware description languages (HDLs), such as Verilog and VHDL, which formally define the system's registers, data paths and timing constraints. RTL represents a point of convergence between functional design and physical circuit implementation for several reasons. First, it provides a level of abstraction that

allows focusing on the behavior and clock-driven timing, avoiding direct management of transistor-level details. Second, it enables automation of the design process, as synthesis tools can automatically translate RTL descriptions into physical circuits. Finally, RTL is the reference standard for most modern ASIC and FPGA designs. [RDJ96]

Writing RTL code, however, is complex, as it requires a paradigm shift compared to traditional software programming and imposes physical, temporal and methodological constraints that are not immediately apparent in the source code. The main difficulty stems from the intrinsically parallel nature of hardware: while in software instructions are executed sequentially, in RTL all descriptions coexist and operate simultaneously. An RTL instruction does not represent an action executed at a specific instant, but rather the description of a constantly active hardware component. This can lead to conceptual errors, such as assuming a nonexistent execution order or the improper use of constructs mistakenly interpreted as sequential, when in reality they describe concurrent circuits. Consequently, the designer must think in terms of circuits and architectures, rather than algorithms.

A further challenge is that a single RTL instruction can translate into tens or hundreds of logic gates. A seemingly minor error can introduce critical paths that limit clock frequency, or cause fanout, routing congestion, or increased power consumption. These effects are not immediately visible in the RTL code and often emerge only in the later stages of synthesis and physical implementation.

Unlike software, time in RTL is an explicit design dimension. The developer must therefore precisely control when the registers sample data, how much combinatorial logic is allowed between two consecutive clock edges and how signals such as reset, enable and stall conditions are handled. Furthermore, not all HDL constructs are synthesizable: some structures may be correct in simulation but not translate to real hardware. Typical examples include non-statically determinable loops or improper assignments within sequential or combinatorial blocks. The challenge lies in producing code that is both semantically correct in simulation and unambiguously interpretable by synthesis tools.

Debugging RTL code represents another significant challenge. A minimal error can lead to profoundly different behavior, functional bugs can only become apparent after millions of simulation cycles and some problems emerge only at runtime on real hardware. For this reason, rigorous verification is essential and often the verification code, including testbench, assertions and coverage metrics, exceeds the size and complexity of the RTL code itself. The challenge lies not only in writing the RTL code, but in producing testable code that makes the debugging phase manageable and effective.

Finally, RTL code must simultaneously satisfy multiple, often conflicting, constraints, such as performance in terms of clock frequency, area, power consumption, reusability and maintainability. Optimizing a single dimension often tends to degrade others, making RTL

design a process of continuously negotiating tradeoffs at the architectural and microarchitectural levels.[PN92]

To mitigate the inherent difficulties of writing pure RTL code, tools like Vitis HLS offer a synthesis-based alternative from high-level languages such as C and C++. These tools allow for the automatic generation of RTL code from algorithmic descriptions, partially automating some of the issues described above and reducing the complexity of the design process.

Vitis is organized as a modular platform stack comprising distinct layers that interact to form a comprehensive development environment. The stack begins with the Target Platform (hardware), followed by the Core Development Kit, accelerable libraries and specialized tools such as Vitis AI. Each layer functions as an independent module, enabling flexibility and customization [New19].

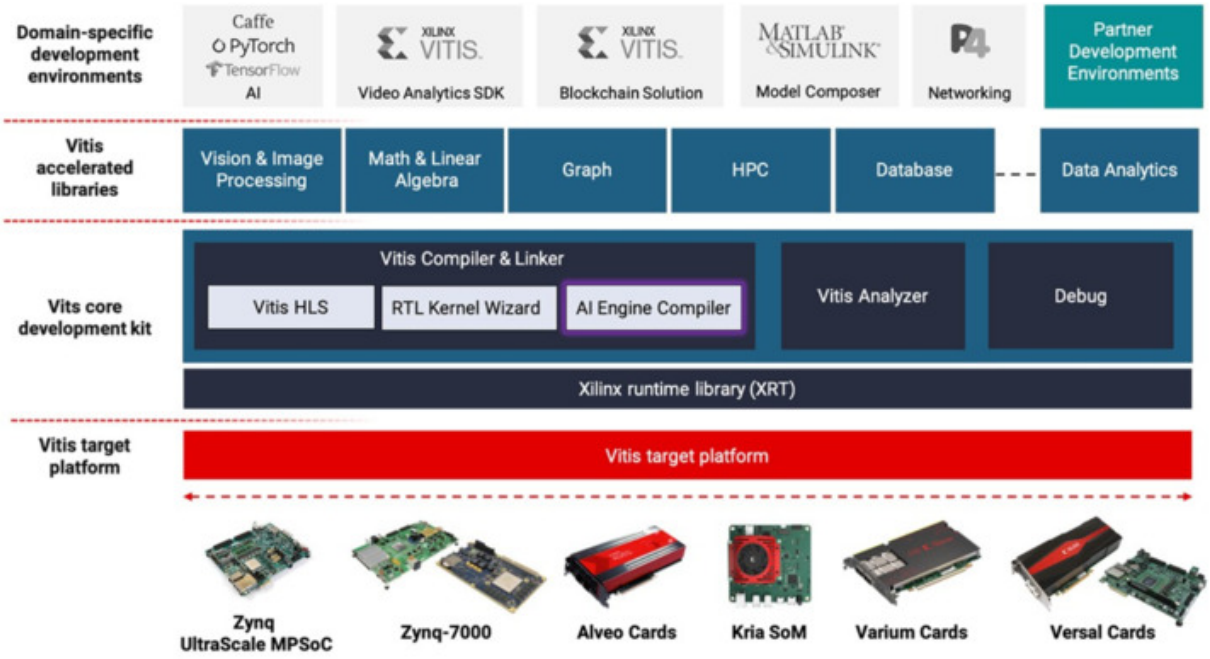


Figure 4.2: Vitis modular platform stack.

- Target Platform: this layer contains the hardware description of the target device, typically exported from Vivado as a Xilinx Support Archive (.XSA) file. It encompasses programmable logic (PL), interfaces and physical resources [New19].
- Core Development Kit: this component includes the open-source Xilinx runtime, compilers, analysis and debugging tools and core libraries that manage data transfer between software and hardware domains [Xil22a].

- Accelerable Libraries: Vitis offers more than 400 optimized, open-source functions for application domains including linear algebra (BLAS), computer vision, data compression, quantitative finance, databases and artificial intelligence [New19].

Vitis Embedded is a component of the platform dedicated to software development for processors integrated into SoCs, such as ARM processors in Xilinx or AMD platforms. It enables the development of C/C++ applications that execute on the host processor and interface with accelerated kernels through the Vitis runtime APIs [Xil25b]. This workflow is particularly advantageous for embedded systems, where some computations are managed by software while others are offloaded to hardware accelerators.
A central feature of Vitis is Vitis High-Level Synthesis (HLS), which allows hardware kernels to be described in C/C++ and synthesized into RTL code for integration into FPGA or SoC designs [Xil25b]. This approach enables developers to design accelerators without manually creating complex RTL logic, thereby increasing productivity and facilitating algorithmic reuse. For Versal architectures featuring AI Engine (AIE), Vitis incorporates dedicated compilers and simulators. This integration enables the development of digital signal processing and performance-intensive functions on AIEs using a unified toolchain [Xil25b].
Vitis libraries offer pre-optimized functions for FPGAs and AIEs, including libraries for digital signal processing (Vitis DSP), computer vision (Vitis Vision) and numerical solvers (Vitis Solver). These resources significantly reduce development time by enabling the reuse of optimized blocks [Xil25b]. Vitis supports multiple design flows tailored to specific application requirements. In this thesis, the model-based design flow was selected. Using Vitis Model Composer, designers begin with Simulink models, explore the design space and generate accelerated IPs without manual HLS or RTL coding [Xil25b].

A primary advantage of employing Vitis and FPGAs within a model-based approach is the democratization of software acceleration through dedicated hardware. Vitis is designed to enable software developers, including those without RTL expertise, to utilize reconfigurable hardware such as FPGAs, SoCs and AIEs. This approach lowers the barrier to entry and broadens access to FPGA technology [New19]. By supporting high-level languages such as C/C++, accelerated libraries and established design patterns, Vitis substantially reduces development and testing time. Developers can utilize familiar APIs and tools, facilitating integration into existing workflows. Despite its abstraction, Vitis delivers high performance. Kernels generated through HLS can be extensively optimized and parallelized. The included libraries are engineered to maximize FPGA resource utilization, resulting in substantial acceleration compared to software-only implementations [Jou19]. Vitis offers a unified integrated development environment (Vitis Unified IDE) that consolidates tools for embedded development, HLS, AIE and performance analysis (Vitis Analyzer) within a single interface. This integration streamlines hardware-software co-design.

The typical flow with Vitis HLS can be schematized as follows, according to the official

guide [Xil22b].

1. First, the designer writes an algorithm or function description in C/C++, intended to be synthesized in hardware.
2. Next, functional simulations are performed in C/C++ to validate the logical behavior without hardware implementation.
3. Then, the HLS tool is invoked to translate the high-level description into RTL (Verilog or VHDL), taking into account constraints (target clock, area, latency) and employing scheduling, binding and pipelining strategies [Xil22b].
4. After this, co-simulation (C/RTL) is performed to compare the behavior of the generated hardware kernel with the original behavior [Xil22b].
5. Finally, the resulting Intellectual Property (IP) is integrated into the traditional FPGA flow (for example, with Vivado Design Suite for physical synthesis, implementation and timing closure) and then deployed on the target hardware [Xil20].

This flow thus reduces the gap between software-algorithmic design and hardware implementation, favoring co-designed hardware/software models.

4.3 Porting

4.3.0.0.1 The first step in the Vitis HLS workflow is writing the models in C++ code. While writing the three models in C++, several key details were considered to facilitate their synthesis. Dynamic memory was not used because HLS requires static-sized data structures or arrays with compile-time sizes. I/O functions such as printf and scanf were avoided because they lack hardware equivalents. The use of recursion and complex pointers has been limited because the HLS compiler must statically analyze data flow. A clear separation has been made between arithmetic calculations and control logic operations, enabling the HLS tool to better optimize pipelines and scheduling.

4.3.0.0.2 The second step in the Vitis HLS workflow consists of using the Code Analyzer to identify major issues in the code. To best utilize the Code Analyzer and all other processes in the Vitis workflow, a Testbench was written. The Testbench's task is to verify the correctness of the model outputs and that they remain correct throughout all model synthesis phases. The testbench was constructed using 1,000 elements predicted by the CPU-based models and saved to a file. During the execution of the test benches, Vitis will then check whether the synthesized models' outputs match those saved in the file. The

choice to use 1,000 elements was made because the subsequent steps in the synthesis flow run the testbench multiple times and having more elements significantly slows down each phase.

The first run of the Code Analyzer provided useful information about the model code, identifying two critical points within the C++ code that prevent complete optimization. The first problem is in the part of the code that multiplies the input values by the coefficients represented in the following code.

```
1     std::array<double,5> y;
2     for (std::size_t i = 0; i < 5; ++i) {
3         y[i] = coeffs[i][0]*x[0] + coeffs[i][1]*x[1] + coeffs[i][2]*x[2]
    + coeffs[i][3]*x[3] + coeffs[i][4]*x[4] + biases[i];
4     }
```

Listing 4.1: Part of the code where the first problem is found

The problem with this portion of the code is that some variables, in this case x and coeffs, are accessed by multiple instructions at the same time and the hardware does not have an adequate implementation to support multiple accesses in a single clock cycle. This problem does not allow this loop to be accelerated and can be solved mainly in two ways. The first way is using the **#pragma HLS ARRAY_PARTITION variable=x complete dim=1** which splits the arrays into separate memory banks to allow parallel accesses. The other way to solve this problem is to unroll the loop manually and make the instructions sequential, this ensures direct optimization by Vitis as the sequential code is automatically optimized by Vitis.

```
1     std::array<double,5> y;
2     y[0] = coeffs[0][0]*x[0] + ... + coeffs[0][4]*x[4] + biases[0];
3     y[1] = coeffs[1][0]*x[0] + ... + coeffs[1][4]*x[4] + biases[1];
4     y[2] = coeffs[2][0]*x[0] + ... + coeffs[2][4]*x[4] + biases[2];
5     y[3] = coeffs[3][0]*x[0] + ... + coeffs[3][4]*x[4] + biases[3];
6     y[4] = coeffs[4][0]*x[0] + ... + coeffs[4][4]*x[4] + biases[4];
```

Listing 4.2: Solution for the first problem

It was decided to use the second solution because the loop is small and performs only 5 iterations; the number of iterations cannot increase, since they correspond to the number of classes the model predicts and partitioning the arrays into multiple blocks results in greater memory consumption.

The second problem is in the code that calculates the probabilities for each class, as shown below.

```
1     double Z = 0.0;
2     for (std::size_t i = 0; i < 5; ++i) {
3         y[i] = std::exp(y[i]);
4         Z += y[i];
```

```
5       }
```

Listing 4.3: Part of the code where the second problem is found

The problem generated by this portion of the code is the presence of a cyclic dependency within the loop, caused by the variable y being written and read in a single loop. To solve this problem, there is only one solution: remove the dependency. In this case, removing the dependency is simple because the calculation of Z is not strictly necessary within this loop and can be performed after the loop ends.

```
1       for (std::size_t i = 0; i < 5; ++i) {
2           y[i] = std::exp(y[i]);
3       }
4       double Z = y[0] + y[1] + y[2] + y[3] + y[4];
```

Listing 4.4: Solution for the second problem

Once the two problems presented by the Code Analyzer were resolved, the code was optimized in terms of transition interval. The Transaction Interval (TI) is the minimal delay between two executions of a process. The overall region performance is limited by the slowest process, common optimization technique is to accelerate the processes with the largest TIs.

The Vitis Code Analyzer has divided the code into seven processes.
The first process initializes the coefficients, bias and the y variable where the predictions will be stored. It then performs the multiplications between the coefficients and the input values. This process has a transition interval of 54 clock cycles (CC).
The second process takes care of the first part of calculating the probabilities of the predictions by raising the variable y to a power. This process has a transition interval of 297 CC.
The third process takes care of calculating the sum of all the values of the variable y and saving them in the variable Z, which will be used to calculate the individual probabilities. This process has a transition interval of 54 CC.
The fourth process is responsible for calculating the individual probabilities by dividing the values of the variable y by the value of the variable Z, this process has a transition interval of 302 CC.
The fifth process is responsible for finding the maximum element in the probability array y, this process has a transition interval of 43 CC.
The sixth process searches for the index of the maximum element of the array y, this process has a transition interval of 4 CC.
The seventh and final process returns the found index, this process has a transition interval of 1 CC.

The Code Analyzer divides the code into processes to convert a sequential software description into a hardware-oriented representation in which concurrency, dataflow and perfor-

mance constraints are made explicit. This decomposition is essential for checking dataflow legality, identifying performance bottlenecks and designing high-throughput HLS architectures. The Code Analyzer also allows sequential processes in the graph to be interactively merged or split to explore alternative architectures. For each modified graph, updated performance estimates are generated, providing a blueprint for manually refactoring the source code. In this case, no better architectures have been found that minimize the transition interval than the one proposed by the Code Analyzer.

Analyzing the information provided by the Code Analyzer, the critical points in the code are processes 2 and 4, which have the slowest transition intervals. To speed up process 4, it was necessary to use the **#pragma HLS PIPELINE II=1**, which introduces a pipeline into the loop. II (Initiation interval) controls every how many cycles a new iteration can begin. Ideally, II = 1 equals maximum throughput. After inserting the pipeline pragma into process 4, its transition interval decreased from 302 to 67. The optimization of process 2 was performed similarly using the pipeline pragma, reducing its transition interval from 297 to 66. After applying these optimizations, the processes have transition intervals that are roughly similar and the slowest has a transition interval of 67. Other changes were tested iteratively but did not improve overall optimization and were therefore not applied.

4.3.0.0.3 The third step in the Vitis HLS workflow involves translating the C++ code into low-level (RTL) code, taking into account the target device, target clock and flow target type. The target device in question is the xcau20p-sfvb784-2-i FPGA, described in Section 4.1. The flow target has been set to Vivado, as it will be used to manage the hardware implementation. After synthesis completes, Vitis HLS automatically generates synthesis reports to help you understand and analyze the implementation's performance. Examples of these reports include the Synthesis Summary report, Schedule Viewer, Function Call Graph and Dataflow Viewer. These reports can be found in the Flow Navigator in the Vitis HLS IDE.

- **Schedule Viewer**: shows each operation and control step of the function and the clock cycle that it executes in.
- **Function Call Graph Viewer**: displays your full design after C Synthesis or C/RTL Co-simulation to show the throughput of the design in terms of latency and II.

In addition to the various graphs and viewers described above, the Vitis HLS tool provides additional views to expand on the information available for analysis of the model's design.

- **Module Hierarchy**: shows the resources and latency contribution for each block in the RTL hierarchy. It also indicates any II or timing violations. In case of timing

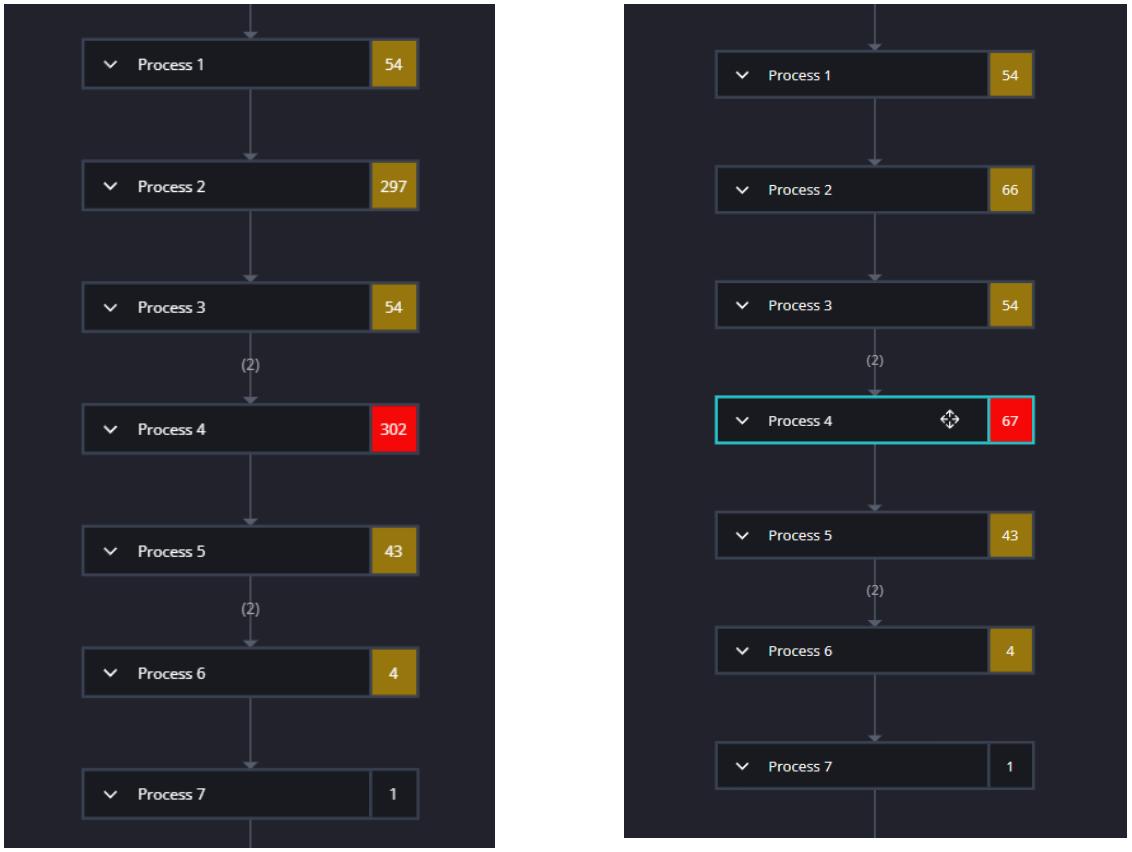


Figure 4.3: A graph of the processes with their TIs generated by Vitis HLS on the left. A graph of the processes with their TIs generated by Vitis HLS after the optimization.

violations, the hierarchy window will also show the total negative slack observed in a specific module.

- **Performance Profile**: shows details on the performance of the block currently selected in the Module Hierarchy view. Performance is measured in terms of latency and the initiation interval and includes details on whether the block was pipelined.
- **Resource Profile**: shows the resources used at the selected level of hierarchy and shows the control state of the operations used.
- **Properties view**: shows the properties of the currently selected control step or operation in the Schedule Viewer.

The Schedule Viewer provides a detailed view of the synthesized RTL, showing each operation and control step of the function, along with the clock cycle in which it executes. The Schedule Viewer helps identify loop dependencies that prevent parallelism, timing violations and data dependencies.

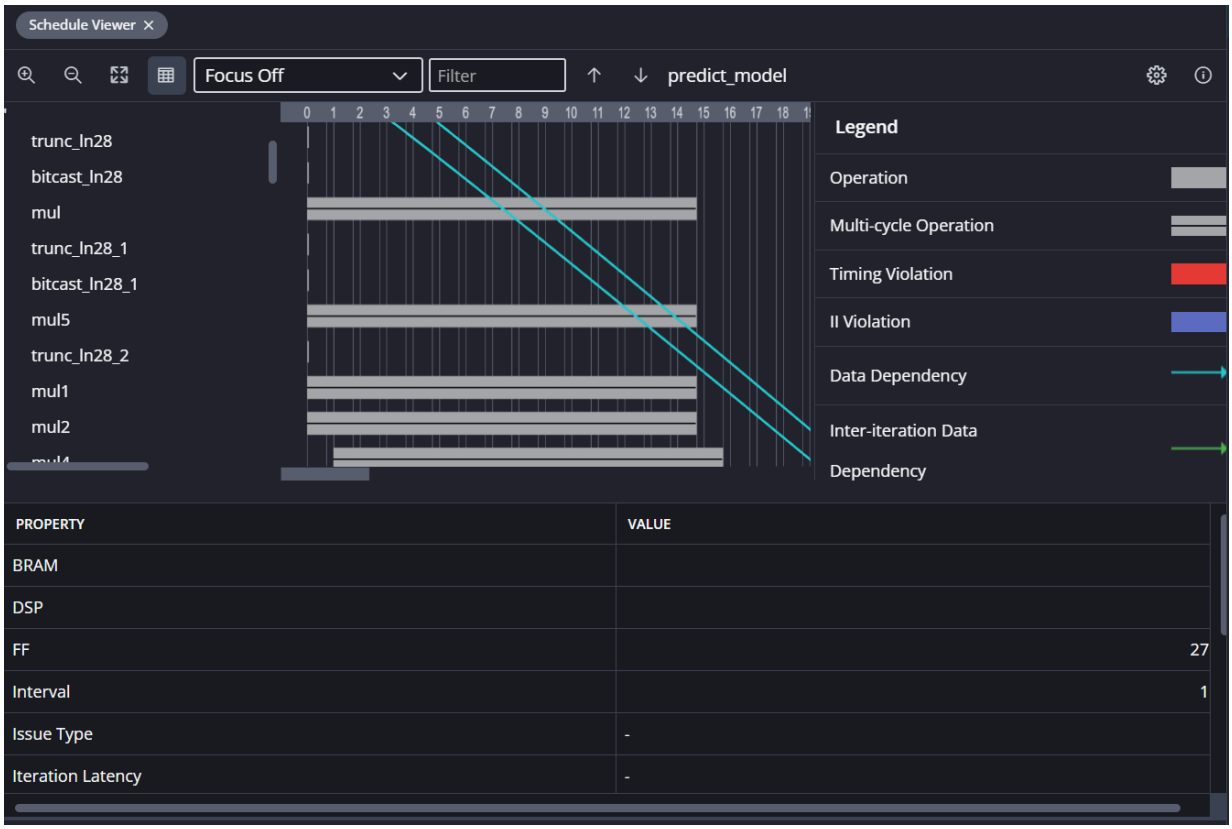


Figure 4.4: Vitis HLS Schedule viewer for the 3 feature MLR model.

The left vertical axis in Figure 4.4 lists the operations and loops in the RTL hierarchy. Operations are in topological order, meaning that an operation on line n can be driven only by operations from a previous line and will drive only an operation in a later line. During the first RTL synthesis iteration, a timing violation occurred because some operations could not complete within the default clock cycle. This issue was resolved by adjusting the clock cycle timing: increasing it to 3 ns for the 3-feature model, 4 ns for the 5-feature model and 5 ns for the 8-cycle model. After making these adjustments, no further violations were detected. The top horizontal axis shows the clock cycles in consecutive order. Each operation is shown as a gray box. The box is horizontally sized according to the operation's delay as a percentage of the total clock cycle. For function calls, the provided cycle information corresponds to the operation latency. Multi-cycle operations are shown as gray boxes with a horizontal line through their centers. The Schedule Viewer

also displays general operator data dependencies as solid blue lines. As shown in Figure 4.4, when selecting an operation, you can see solid blue arrows highlighting the specific operator dependencies. This type of display enables detailed analysis of data dependencies.

At the bottom of the Schedule Viewer, as shown in Figure 4.4, is the Properties view that displays the properties of a currently selected object in the Schedule Viewer. This lets you see details of the specific function, loop, or operation that is selected in the Schedule Viewer. The types of elements that can be selected and the properties displayed are Functions or Loops with specifications of:

- Initiation Interval (II): the number of clock cycles before the function or loop can accept new input data.
- Loop Iteration Latency: the number of clock cycles it takes to complete one iteration of the loop.
- Latency: the number of clock cycles required for the function to compute all output values, or for the loop to complete all iterations.
- Pipelined: indicates that the function or loop is pipelined in the RTL design.
- Slack: the timing slack for the function or loop.
- Tripcount: the number of iterations a loop completes.
- Resource Utilization: displays the number of BRAM, DSP, LUT, or FF used to implement the function or loop; for example, we can see in Figure 4.4 that the FFs used are 27.

The Function Call Graph Viewer, accessible via the Flow Navigator, visualizes the complete RTL code design following C Synthesis or C/RTL Co-simulation. This viewer is intended to present the design's throughput in terms of latency and initiation interval (II). It assists in identifying the critical path and bottlenecks within the design that should be addressed to enhance throughput. Additionally, it highlights paths where throughput may be unbalanced, leading to FIFO stalls or deadlock.

In this context, the design hierarchy presented in the user interface may differ from the source code as a result of HLS optimizations, such as converting loops into function pipelines. Functions that are inlined are not displayed in the call graph because they are no longer distinct entities in the synthesized code. When multiple instances of a function are generated, each unique instance is represented in the call graph. This representation enables identification of the functions that contribute to the latency and initiation interval of a calling function.

NAME	II	LATENCY	BRAM	URAM
predict_model	249	248	0	
predict_model_Pipeline_VITIS_LOOP_35_1	6	47	0	
VITIS_LOOP_35_1	1	45		
predict_model_Pipeline_VITIS_LOOP_44_2	6	67	0	
VITIS_LOOP_44_2	1	65		

Figure 4.5: Vitis HLS Schedule viewer for the 3 feature MLR model. In particular we see that loop 35 has an II of 1 and a latency of 45.

The graph, as illustrated in Figure 4.5, represents functions as rectangular boxes and loops as oval boxes. Each box displays initiation interval, latency and either resource or timing data, depending on the selected view. Prior to completion of C/RTL cosimulation, the performance and resource metrics presented in the graph are derived from the C Synthesis phase and are therefore estimates provided by the HLS tool.

4.3.0.0.4 The fourth step in the Vitis HLS workflow is to test the generated RTL code and ensure it behaves identically to the C++ source code. This process is done using the C/RTL co-simulation tool and the testbench. The C/RTL co-simulation is located in the Flow Navigator and the Run co-simulation command allows you to verify the RTL synthesis results. The Co-simulation Dialog box is opened as shown in the following Figure 4.6. This dialog box provides several options for configuring co-simulation. The available settings are described below.
Simulator: allows the selection of an HDL simulator from those available in the Vivado Design Suite. The Vivado simulator is used as the default and is retained, since Vivado is

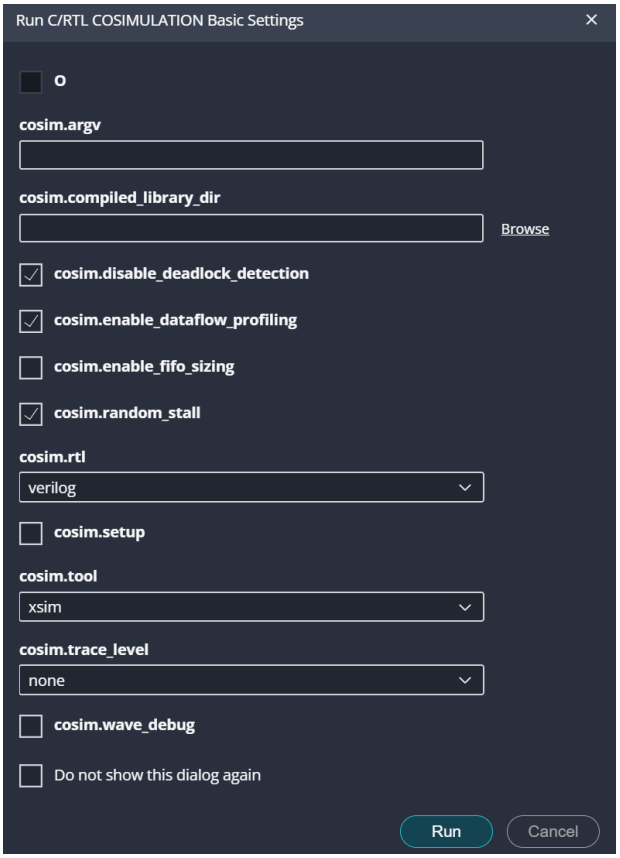


Figure 4.6: Co-simulation dialog box.

the tool employed to implement the RTL code on the FPGA board.
Language: specifies the output language for simulation, with Verilog or VHDL as available options. Verilog is selected because it is the native language supported by Vivado.
Optimizing Compile: Enables compilation optimizations to improve simulation runtime performance, at the cost of increased compilation time.
Wave Debug: enables the visualization of waveforms for all processes in the RTL simulation. When this option is activated, the simulator graphical user interface is launched, allowing inspection of dataflow activity through the generated waveforms.
Deadlock Detection: disables automatic deadlock detection and opens the Cosim Deadlock Viewer during co-simulation.
Dynamic Deadlock Prevention: prevents deadlocks by enabling automatic tuning of FIFO channel sizes for dataflow profiling during co-simulation.

The C/RTL verification process consists of three phases:

1. the C simulation is executed and the inputs to the top-level function, or the Design-

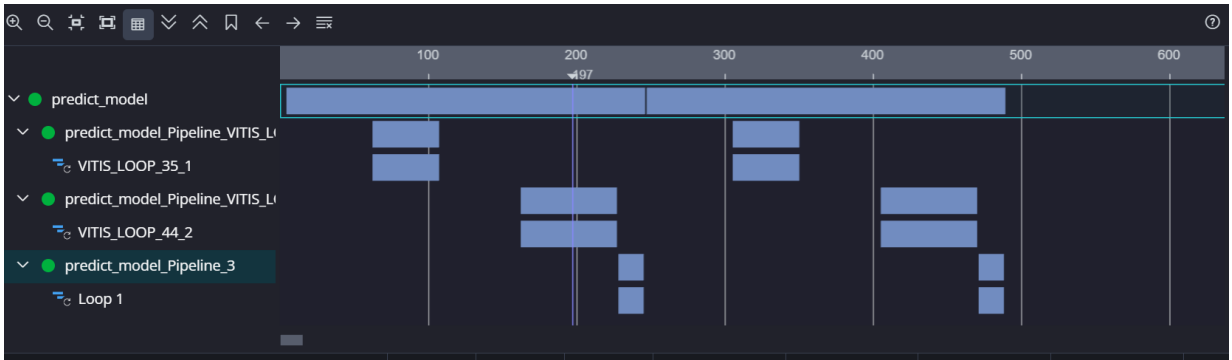
UnderTest (DUT), are saved as “input vectors.”;

2. the “input vectors” are used in an RTL simulation using the RTL created by Vitis HLS in Vivado simulator. The outputs from the RTL, or results of simulation, are saved as “output vectors.”;

3. the “output vectors” from the RTL simulation are returned to the main() function of the C test bench to verify the results are correct. The C test bench performs verification of the results, in some cases by comparing to known good results.

After the co-simulation completes, Vitis HLS automatically generates co-simulation reports, a timeline trace and a Function call graph. These results may differ from values reported after HLS synthesis, which are based on the absolute shortest and longest paths through the design. The results from the C/RTL co-simulation show the actual latency and II values for the given simulation data set.

The Timeline Trace viewer as shown in Figure 4.7 displays the runtime profile of the functions in the implemented design. It is especially useful to see the behavior of dataflow regions after Co-simulation, as there is no need to launch the Vivado logic simulator to view the timeline. The Timeline Trace viewer displays multiple iterations through the sub-functions of a dataflow region, shows where the functions start and end, the upper part of the Figure 4.7. The viewer provides basic tools for viewing the timeline, such as



NAME	AVG II	MAX II	MIN II	AVG LATENCY	MAX LATENCY	MIN LATENCY	STALLING TIME	FIFO E
AESL_inst_predict_model	281	491	243	242	242	242	0.00%	0.00%
grp_predict_model_Pipeline_VITIS_LOOP_35_1_fu_157	281	491	243	45	45	45	0.00%	0.00%
grp_predict_model_Pipeline_VITIS_LOOP_44_2_fu_162	281	491	243	65	65	65	0.00%	0.00%
grp_predict_model_Pipeline_3_fu_168	281	491	243	17	17	17	0.00%	0.00%
VITIS_LOOP_35_1	281	491	243	46	46	46	-	-
VITIS_LOOP_44_2	281	491	243	66	66	66	-	-
Loop 1	281	491	243	18	18	18	-	-

Figure 4.7: Timeline trace of the 3 feature MLR model on 1000 instances.

adding markers, stepping from one marker to the next and measuring the time between

markers. The report also shows the loop internal II and latency, The table at the bottom of the Figure 4.7 shows dataflow path status, including performance, total time, stalling time and percentage.

After completing the four Vitis workflow steps and utilizing the analyses above, the Vitis HLS software provides the estimated results for the three Multinomial Logistic regression models, shown in Table 4.1. From the results of the Table 4.1, we can see that: the estimated clock speed before synthesis is similar for the 3 and 5 feature models, with the 8 feature model having a slower clock speed; however, the more accurate clock speed estimate made after synthesis of the three models shows faster clock speeds for all three models; the number of low-level operations increases with the model's complexity. Furthermore, it can be noted that the MLR model with 5 features is the one that performs the worst, since to obtain the same clock speed as the model with 3 features, it uses more resources than the model with 8 features and having a greater number of low-level operations than the model with 3 features makes the execution time of the model with 5 features slower.

	3 feature model	5 feature model	8 feature model
Estimated Clock speed	2,835 ns	2,811 ns	4,763 ns
Number of low level operations	56	78	106
Avg II	281	313	189
Avg Latency	242	312	188
Total execution time	281125 ns	312999 ns	388999 ns
Estimated clock speed post synthesis	1,844 ns	1,874 ns	2,591 ns
LUT usage	9105	11104	11036
FF usage	15731	20763	11488
DSP usage	83	111	121
BRAM usage	4	4	4

Table 4.1: Table of results after the 4th step of Vitis workflow

4.4 Results

The fifth step in the Vitis HLS workflow is the implementation phase, performed using Vivado Design Suite Hub, which includes a simulator for the target FPGA. The simulator was necessary because, for various reasons, it was not possible to perform tests on the physical FPGA board. The Vitis HLS tool is limited in the estimations it can provide about the RTL design it generates. It can project resource utilization and timing of the end result, but these are just projections. To get a better view of the RTL design, you

can actually run Vivado synthesis and place and route on the generated RTL design and review the actual results of timing and resource utilization in the simulator.

The Implementation Report contains the results of the Synthesis and Place-and-Route. The report is structured into several sections, each serving a distinct purpose:

- **General Information**: provides general information related to the design and implementation;
- **Run Constraints and Options**: reports the constraints and options that were set for the RTL Synthesis run and/or the Place & Route run. This shows you the constraints set and/or modified for the run;
- **Resource Usage and Final Timing**: the Resource Usage and Final Timing sections provide a quick summary of the resources and timing achieved by either the RTL Synthesis or Place & Route run. These sections provide a high-level overview of resource utilization and whether timing goals were met;
- **Resources**: a detailed per-module split-up of resources is shown in this table. In addition, the tables can also show the original variable and source location information from the source code. If a particular resource was the result of a user-specified pragma, then this can also be shown in the table. This allows you to relate your C code to the synthesized RTL implementation. Inspecting this report is very beneficial because it is generated after Vivado has synthesized the design; therefore, functional blocks such as DSPs and other logic units have been instantiated in the circuit;
- **Timing Paths**: the Timing Paths reports show the timing-critical paths that result in the worst slack in the design. By default, the tool will show the top 10 worst negative slack paths. Each path in the table provides detailed information about the combination path between one flip-flop and another. Breaking these long combinational paths will be required to address the timing issues.

Analyzing the reports generated after running the simulator in Vivado, the results shown in Table 4.2 were taken and comparing the results in this table with the results found in the 4 steps in the Table 4.1, it can be noted that the Vitis cosimulation tool makes very optimistic predictions of execution times and resource usage, because almost all of the predictions made by the cosimulation tool are better than those found by the Vivado simulator. The Vivado target device simulator shows more precisely that the clock speed for each of the three models is 0,9 ns slower than the cosimulation tool predicts. Furthermore, the simulator shows that the resources used are 1% greater than those predicted by the 4-step cosimulator. Furthermore, using the simulator, it was possible to measure each model's execution time more accurately, as it provides the total number of clock cycles for an entire execution. Using the number of clock cycles and the clock speed for each model,

	3 feature model	5 feature model	8 feature model
Final Clock speed	2,301 ns	2,677 ns	3,455 ns
Number of clock cycles	118	141	114
Execution time	271,518 ns	377,457 ns	393,870 ns
LUT usage	9492	10134	11420
LUT percentage	10,44 %	10,46 %	10,48 %
FF usage	15923	16737	11595
FF percentage	7,30%	7,67%	5,32%
DSP usage	83	111	121
DSP percentage	9,22 %	12,33 %	13,44 %
BRAM usage	2	2	2
BRAM percentage	1 %	1 %	1 %

Table 4.2: Table of results after the simulation on Vivado

we calculated each model's execution time. The 3-feature model predicts in 271,518 ns, the 5-feature model predicts in 377,457 ns and the 8-feature model predicts in 393,870 ns. The reports generated by Vivado also show additional useful information, such as the actual resource consumption within the target device, the device's power consumption for each implementation, in Figure 5.3 and a low-level operations diagram showing how they are connected in Figure 5.2. Vivado also allows viewing the schematic of the current implementation on the FPGA board, showing the occupied cells and their connections in Figure 5.4.

Table 4.3 shows the execution times associated with each phase of the Vitis workflow. Analysis of the table shows that the most time-consuming steps are the fourth and fifth. The fourth step is longer because it includes the validation phase of the RTL model and

	3 feature model	5 feature model	8 feature model
C simulation	7 s	13 s	16 s
C synthesis	32 s	36 s	45s
C/RTL cosimulation	55m 28s	24m 51s	24m 38s
Implementation	21m 58s	23m 11s	49m 52s

Table 4.3: Table of execution times of each Vitis step.

the C++ model. This verification process is conducted using a dedicated testbench; consequently, as the number of test cases included in the testbench increases, the time required to complete the validation phase increases proportionally. The fifth step is also particularly time-consuming, as during the implementation phase, Vitis optimizes the logical paths to meet timing requirements. This process involves finding the most efficient configurations, taking into account all the constraints imposed by the target device, resulting in increased

computational complexity and execution times.

Finally, comparing the three Multinomial Logistic Regression models in the unoptimized CPU version with their counterparts in the FPGA, it is clear that execution on the FPGA results in an average 10% faster execution time. Specifically, the optimized MLR model with 3 features runs 27 ns faster than the CPU version, the model with 5 features runs 30 ns faster and the model with 8 features runs 36 ns faster. The 10% speed gain seems small compared to the amount of work performed.
This perception stems from the fact that CPU-based models are already very fast; in this case, their speed is guaranteed by their simplicity. The simplicity of MLR models also plays an important role in the FPGA versions. Since the models do not utilize all the board's resources, multiple instances of the same model can run in parallel, efficiently using all available resources to achieve greater execution speed. For example, for the MLR model with 3 features synthesized on the FPGA, Table 4.2 shows that the model uses only 10.44% of the available LUTs, leaving 89.66% unused. The same applies to the other resources of the FF: only 7.3% is used, the DSP only 9.22% is used and the BRAM only 1% is used. Based on these data, it is clear that the board's resources are underutilized. By using parallel instances of the same model within the board, resource utilization can be maximized and greater computational speedup achieved, as more data can be processed simultaneously. However, this approach increases power consumption on the FPGA board and requires multiple data streams to be processed simultaneously.

	3 feature model	5 feature model	8 feature model
Execution time (CPU)	298,12 ns	415,36 ns	433,42 ns
Execution time (FPGA)	271,518 ns	377,457 ns	393,87 ns
Speed up (FPGA)	8.98%	9.13%	9.16%
Power consumption (FPGA)	0.963 W	1.236 W	1.557 W
Execution time (FPGA with multiple instances)	30,16 ns	47,182 ns	49,234 ns
Speed up (FPGA with multiple instances)	80.82%	73.04%	73.28%
Power consumption (FPGA with multiple instances)	8.667 W	9.88 W	10.89 W

Table 4.4: Table of results after the simulation on Vivado

The use of the FPGA, in addition to ensuring a significant increase in model performance, allows for greater energy efficiency compared to CPU-based counterparts. The CPU used to evaluate the non-optimized models is an AMD Ryzen 7 4800H, which reaches a power consumption of between 60 and 65 W under maximum load. However, this value does not represent the actual power consumption attributable to model execution, since not all processor resources are dedicated exclusively to this activity. In contrast, as reported in

Table 4.4, the FPGA exhibits significantly lower power consumption. In particular, single model executions require an average power consumption between 1 and 1.5 W, while executions with multiple instances processed in parallel show an average power consumption of around 9–10 W.
This reduction in power consumption compared to a CPU is attributable to several factors: the FPGA allows the algorithm to be implemented directly in dedicated hardware, operates at lower clock frequencies, selectively powers only the components actually involved in processing and significantly reduces access to external memories. These features overall contribute to the high energy efficiency of the FPGA architecture.

Chapter 5

Conclusion and Future development

The pipeline proposed in this thesis, based on AutoWeka and Vitis HLS, represents an effective approach for democratizing access to machine learning models and computationally intensive hardware architectures. Specifically, AutoWeka enables even users without specialized machine learning training to develop models quickly and easily, while maintaining satisfactory performance levels. At the same time, for expert users, AutoWeka represents a valuable tool, significantly accelerating the model exploration and selection process by optimizing the search based on the characteristics of the dataset. Furthermore, the ability to include a broad set of algorithms in the search space can lead to the identification of solutions that the expert might not have considered or been aware of.

From a hardware implementation perspective, Vitis HLS emerges as a powerful development environment, capable of substantially reducing the gap between software and hardware. It allows developers to exploit the high computational capabilities of FPGAs without requiring in-depth knowledge of RTL implementations. In this context, the use of Vitis HLS mitigates the intrinsic complexity of FPGA systems and enables systematic architectural exploration of hardware designs, an activity that is difficult to achieve using traditional RTL approaches.

In the case study considered in this thesis, which concerns the sports sector, the proposed pipeline has demonstrated its effectiveness. From an automation perspective, AutoWeka was able to identify a suitable MLR model for the classification of the physiological dataset. The selected model is able to efficiently classify dynamic data acquired from sensors, ensuring response times compatible with real-time processing requirements. Multinomial logistic regression, is particularly efficient thanks to its low computational complexity, a feature that contributes significantly to meeting real-time constraints.

In terms of model acceleration, the use of FPGA has proven to be particularly effective in scaling the computational capabilities of an already intrinsically fast model. Executing

the model on the FPGA is on average 9% faster than executing it on a CPU. Furthermore, the ability to instantiate multiple copies of the model in parallel, using a single board, allows for a reduction in average execution times of approximately 75% compared to the CPU counterpart. Finally, the use of the FPGA also ensures a significant improvement in energy efficiency, with a consumption equal to approximately one-sixth of that required by the CPU, confirming the validity of the proposed approach in terms of both performance and energy.

Finally, among possible future developments, with a view to further simplifying access to computationally powerful machine learning models, the integration of an additional automation tool dedicated to the dataset preprocessing phase is envisioned. Such a module would automate key operations such as data cleaning, normalization, feature selection, and missing value management, further reducing manual intervention and improving process reproducibility. Furthermore, given the rapid evolution of research in the field of AutoML, another area of future work involves the potential replacement of AutoWeka with more advanced tools, which do not yet exist. In particular, the adoption of frameworks capable of not only exploring the space of traditional machine learning models but also automatically searching for neural network architectures would allow the proposed pipeline to be extended to more complex models, broadening its applicability and performance potential.

Bibliography

[Agr17] Alan Agresti. *Statistical methods for the social sciences*. Pearson, 2017.

[BB12] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *The journal of machine learning research*, 13(1):281–305, 2012.

[BN06] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[BPT24] Armando Biscontini, E Popovici, and A Temko. Machine learning for fpga electronic design automation. *IEEE Access*, 2024.

[CBHK02] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[CCA+11] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36, 2011.

[CG15] Xinlei Chen and Abhinav Gupta. Webly supervised learning of convolutional networks. In *Proceedings of the IEEE international conference on computer vision*, pages 1431–1439, 2015.

[CGMT09] Philippe Coussy, Daniel D Gajski, Michael Meredith, and Andres Takach. An introduction to high-level synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, 2009.

[CH02] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (csuR)*, 34(2):171–210, 2002.

[CIKW16] Xu Chu, Ihab F Ilyas, Sanjay Krishnan, and Jiannan Wang. Data cleaning: Overview and emerging challenges. In *Proceedings of the 2016 international conference on management of data*, pages 2201–2206, 2016.

[CMI+15] Xu Chu, John Morcos, Ihab F Ilyas, Mourad Ouzzani, Paolo Papotti, Nan Tang, and Yin Ye. Katara: A data cleaning system powered by knowledge bases and crowdsourcing. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1247–1261, 2015.

[CSSM20] Philip Colangelo, Oren Segal, Alex Speicher, and Martin Margala. Automl for multilayer perceptron and fpga co-design. In *2020 IEEE 33rd International System-on-Chip Conference (SOCC)*, pages 265–266. IEEE, 2020.

[CZ06] Jason Cong and Zhiru Zhang. An efficient and versatile scheduling algorithm based on sdc formulation. In *Proceedings of the 43rd annual Design Automation Conference*, pages 433–438, 2006.

[CZM+19] Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. Autoaugment: Learning augmentation strategies from data. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 113–123, 2019.

[Ele24] DigiKey Electronics. Fpga product summary and specifications., 2024. URL: `https://www.digikey.com/en/products/detail/amd/XCAU20P-2SFVB784I/15861198`.

[EMH19] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.

[FAHS10] Mohamed Farouk Abdel Hady and Friedhelm Schwenker. Combining committee-based semi-supervised learning and active learning. *Journal of Computer Science and Technology*, 25(4):681–698, 2010.

[FCC+21] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. Bambu: an open-source research framework for the high-level synthesis of complex applications. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1327–1330. IEEE, 2021.

[FKE+15] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. *Advances in neural information processing systems*, 28, 2015.

[Gam04] João Gama. Functional trees. *Machine learning*, 55(3):219–250, 2004.

[GBC+15] Isabelle Guyon, Kristin Bennett, Gavin Cawley, Hugo Jair Escalante, Sergio Escalera, Tin Kam Ho, Núria Macià, Bisakha Ray, Mehreen Saeed, Alexander Statnikov, et al. Design of the 2015 chalearn automl challenge. In *2015 International joint conference on neural networks (IJCNN)*, pages 1–8. IEEE, 2015.

[HHLB11] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International conference on learning and intelligent optimization*, pages 507–523. Springer, 2011.

[HHLB14] Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. An efficient approach for assessing hyperparameter importance. In *International conference on machine learning*, pages 754–762. PMLR, 2014.

[HJLS13] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. *Applied logistic regression*. John Wiley & Sons, 2013.

[HKV19] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. *Automated machine learning: methods, systems, challenges*. Springer Nature, 2019.

[HZC21] Xin He, Kaiyong Zhao, and Xiaowen Chu. Automl: A survey of the state-of-the-art. *Knowledge-based systems*, 212:106622, 2021.

[IA17] Ozan Irsoy and Ethem Alpaydın. Unsupervised feature extraction with autoencoder trees. *Neurocomputing*, 258:63–73, 2017.

[Jen23] Felix Jentzsch. Hardware-aware automl for exploration of custom fpga accelerators for radioml. In *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*, pages 359–360, 2023. doi: 10.1109/FPL60245.2023.00066.

[Jou19] COTS Journal. Is xilinx's vitis a game changer? this unified software platform unlocks a new design experience for developers, 2019. URL: https://www.cotsjournalonline.com/is-xilinxs-vitis-a-game-changer-this-unified-software-platform-unlocks-a-n

[KBE14] Brent Komer, James Bergstra, and Chris Eliasmith. Hyperopt-sklearn: Automatic hyperparameter configuration for scikit-learn. In *Scipy*, pages 32–37, 2014.

[KFGW17] Sanjay Krishnan, Michael J Franklin, Ken Goldberg, and Eugene Wu. Boostclean: Automated error detection and repair for machine learning. *arXiv preprint arXiv:1711.01299*, 2017.

[KHD+15] Arun Sai Krishnan, Xiping Hu, Jun-qi Deng, Li Zhou, Edith C-H Ngai, Xitong Li, Victor CM Leung, and Yu-kwong Kwok. Towards in time music mood-mapping for drivers: A novel approach. In *Proceedings of the 5th ACM Symposium on Development and Analysis of Intelligent Vehicular Networks and Applications*, pages 59–66, 2015.

[KW19] Sanjay Krishnan and Eugene Wu. Alphaclean: Automatic generation of data cleaning pipelines. *arXiv preprint arXiv:1904.11827*, 2019.

[LHW+20] Yonggang Li, Guosheng Hu, Yongtao Wang, Timothy Hospedales, Neil M Robertson, and Yongxin Yang. Differentiable automatic data augmentation. In *European conference on computer vision*, pages 580–595. Springer, 2020.

[LM12] Huan Liu and Hiroshi Motoda. *Feature selection for knowledge discovery and data mining*, volume 454. Springer science & business media, 2012.

[LP+20] Erin LeDell, Sebastien Poirier, et al. H2o automl: Scalable automatic machine learning. In *Proceedings of the AutoML Workshop at ICML*, volume 2020, page 24, 2020.

[McC19] Peter McCullagh. *Generalized linear models*. Routledge, 2019.

[MCSK17] Qinxue Meng, Daniel Catchpoole, David Skillicom, and Paul J Kennedy. Relational autoencoder for feature extraction. In *2017 International joint conference on neural networks (IJCNN)*, pages 364–371. IEEE, 2017.

[ML02] Hiroshi Motoda and Huan Liu. Feature selection, extraction and construction. *Communication of IICM*, 5(2):67–72, 2002.

[Mur12] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[NAM20] Alireza Naghizadeh, Mohammadsajad Abavisani, and Dimitris N Metaxas. Greedy autoaugment. *Pattern Recognition Letters*, 138:624–630, 2020.

[New19] PR Newswire. a unified software platform unlocking a new design experience for all developers, 2019. URL: `https://www.prnewswire.com/news-releases/xilinx-announces-vitis--a-unified-software-platform-unlocking-a-new-design-html`.

[NP19] Thiloshon Nagarajah and Guhanathan Poravi. A review on automated machine learning (automl) systems. In *2019 IEEE 5th International Conference for Convergence in Technology (I2CT)*, pages 1–6. IEEE, 2019.

[OBUM16] Randal S Olson, Nathan Bartley, Ryan J Urbanowicz, and Jason H Moore. Evaluation of a tree-based pipeline optimization tool for automating data science. In *Proceedings of the genetic and evolutionary computation conference 2016*, pages 485–492, 2016.

[OM16] Randal S Olson and Jason H Moore. Tpot: A tree-based pipeline optimization tool for automating machine learning. In *Workshop on automatic machine learning*, pages 66–74. PMLR, 2016.

[PGZ+18] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient neural architecture search via parameters sharing. In *International conference on machine learning*, pages 4095–4104. PMLR, 2018.

[PN92] Alberto Palacios PAWLOVSKY and Sachio NAITO. Verification of register transfer level (rtl) designs. *IEICE TRANSACTIONS on Information and Systems*, 75(6):785–791, 1992.

[PVG+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[RAHL19] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789, 2019.

[RDJ96] Anand Raghunathan, Sujit Dey, and Niraj K Jha. Register-transfer level estimation techniques for switching activity and power consumption. In *Proceedings of International Conference on Computer Aided Design*, pages 158–165. IEEE, 1996.

[REGSV02] Jonathan Rose, Abbas El Gamal, and Alberto Sangiovanni-Vincentelli. Architecture of field-programmable gate arrays. *Proceedings of the IEEE*, 81(7):1013–1029, 2002.

[RMKR24] Jayesh Rane, SK Mallick, O Kaya, and NL Rane. Automated machine learning (automl) in industry 4.0, 5.0, and society 5.0: Applications, opportunities, challenges, and future directions. *Future Research Opportunities for Artificial Intelligence in Industry*, 4:181–206, 2024.

[SLA12] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.

[Son09] Parikshit Sondhi. Feature construction methods: a survey. *sifaka. cs. uiuc. edu*, 69:70–71, 2009.

[SW19] Benjamin Carrion Schafer and Zi Wang. High-level synthesis design space exploration: Past, present, and future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2628–2639, 2019.

[THHLB13] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 847–855, 2013.

[Tri18] Stephen M Steve Trimberger. Three ages of fpgas: a retrospective on the first thirty years of fpga technology: this paper reflects on how moore's law has driven the design of fpgas through three epochs: the age of invention, the age of expansion, and the age of accumulation. *IEEE Solid-State Circuits Magazine*, 10(2):16–29, 2018.

[UDP+20] Ecenur Ustun, Chenhui Deng, Debjit Pal, Zhijing Li, and Zhiru Zhang. Accurate operation delay prediction for fpga hls using graph neural networks. In *Proceedings of the 39th international conference on computer-aided design*, pages 1–9, 2020.

[VDJ98] Haleh Vafaie and Kenneth De Jong. Evolutionary feature space transformation. In *Feature Extraction, Construction and Selection: a data mining perspective*, pages 307–323. Springer, 1998.

[WFH+05] Ian H Witten, Eibe Frank, Mark A Hall, Christopher J Pal, and Mining Data. Practical machine learning tools and techniques. In *Data mining*, volume 2, pages 403–413. Elsevier Amsterdam, The Netherlands, 2005.

[Xil20] AMD Xilinx. An introduction to vitis hls. Technical report, Amd Xilinx, 10 2020. URL: `https://www.xilinx.com/content/dam/xilinx/publications/presentations/c_D1_05_Introduction_to_Vitis_HLS.pdf`.

[Xil22a] AMD Xilinx. Designing ip subsystems using ip integrator. Technical report, Amd Xilinx, 10 2022. URL: `https://www.xilinx.com/support/documents/sw_manuals/xilinx2022_2/ug994-vivado-ip-subsystems.pdf`.

[Xil22b] AMD Xilinx. Vitis high-level synthesis user guide. Technical report, Amd Xilinx, 10 2022. URL: `https://www.xilinx.com/support/documents/sw_manuals/xilinx2022_2/ug1399-vitis-hls.pdf`.

[Xil24] AMD Xilinx. Power design manager user guide. Technical report, Amd Xilinx, 05 2024. URL: https://docs.amd.com/r/2024.1-English/ug1556-power-design-manager/Introduction.

[Xil25a] AMD Xilinx. Ultrascale architecture and product data sheet: Overview. Technical report, Amd Xilinx, 11 2025. URL: https://docs.amd.com/v/u/en-US/ds890-ultrascale-overview.

[Xil25b] AMD Xilinx. The vitis software platform development environment, 2025. URL: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis.html.

[Xil25c] AMD Xilinx. Vivado design suite user guide: Design flows overview. Technical report, Amd Xilinx, 11 2025. URL: https://docs.amd.com/r/en-US/ug892-vivado-design-flows-overview.

[Yar95] David Yarowsky. Unsupervised word sense disambiguation rivaling supervised methods. In *33rd annual meeting of the association for computational linguistics*, pages 189–196, 1995.

[YYH+25] Şenda Yıldırım, Ahmet Deniz Yücekaya, Mustafa Hekimoğlu, Meltem Ucal, Mehmet Nafiz Aydin, and İrem Kalafat. Ai-driven predictive maintenance for workforce and service optimization in the automotive sector. *Applied Sciences*, 15(11):6282, 2025.

[ZG04] Yan Zhou and Sally Goldman. Democratic co-learning. In *16th IEEE international conference on tools with artificial intelligence*, pages 594–602. IEEE, 2004.

[Zhe98] Zijian Zheng. A comparison of constructing different types of new feature for decision tree learning. In *Feature extraction, construction and selection: A data mining perspective*, pages 239–255. Springer, 1998.

[ZL16] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

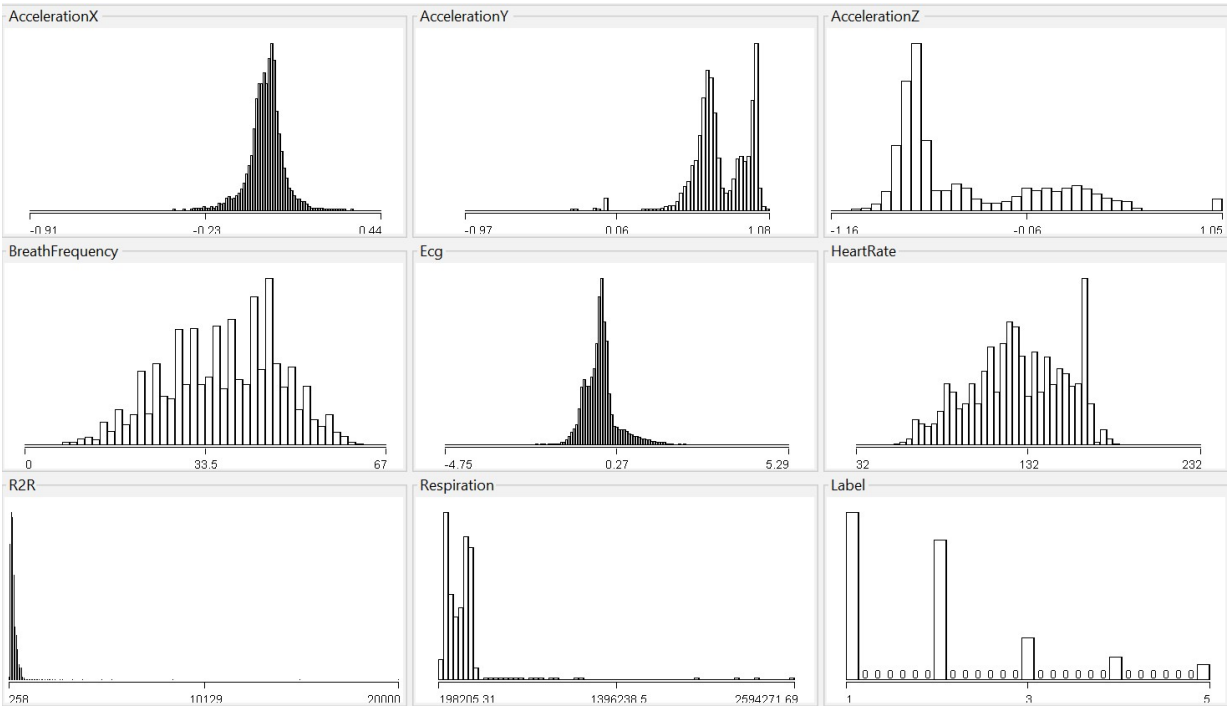


Figure 5.1: Plot showing the distribution of individual features.

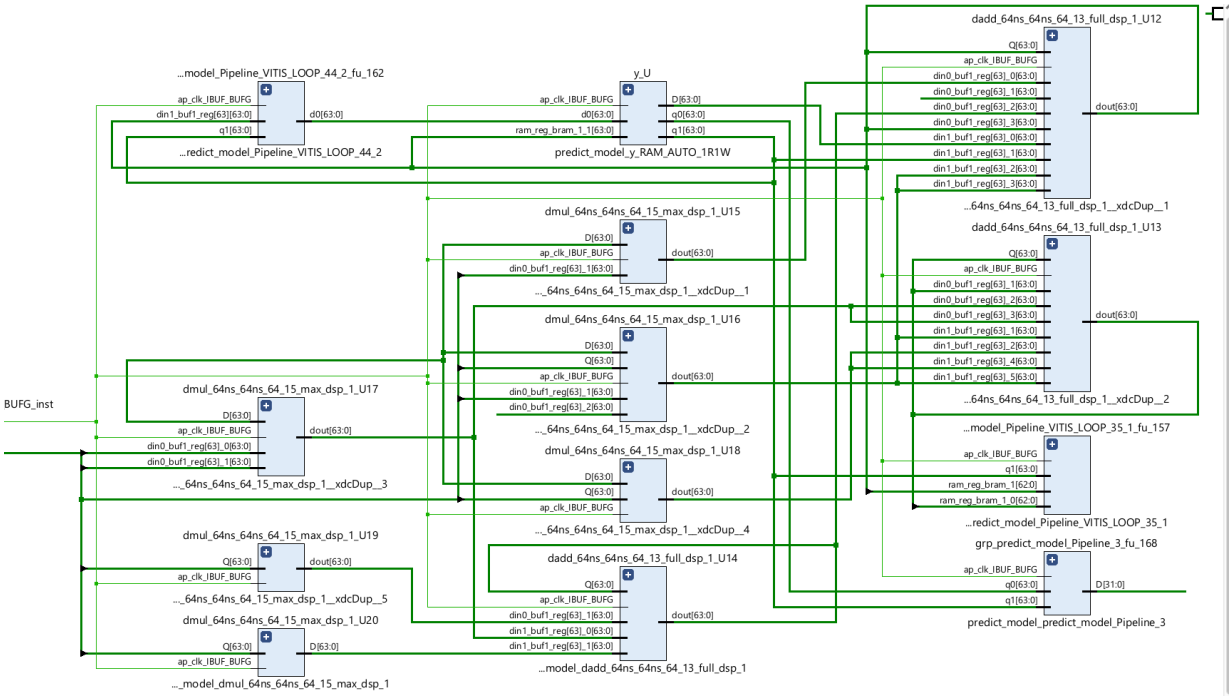


Figure 5.2: This image shows the flowchart of the low-level operations used within the FPGA and how they relate to each other. The image represents the operations used by the MLR model with 3 features.

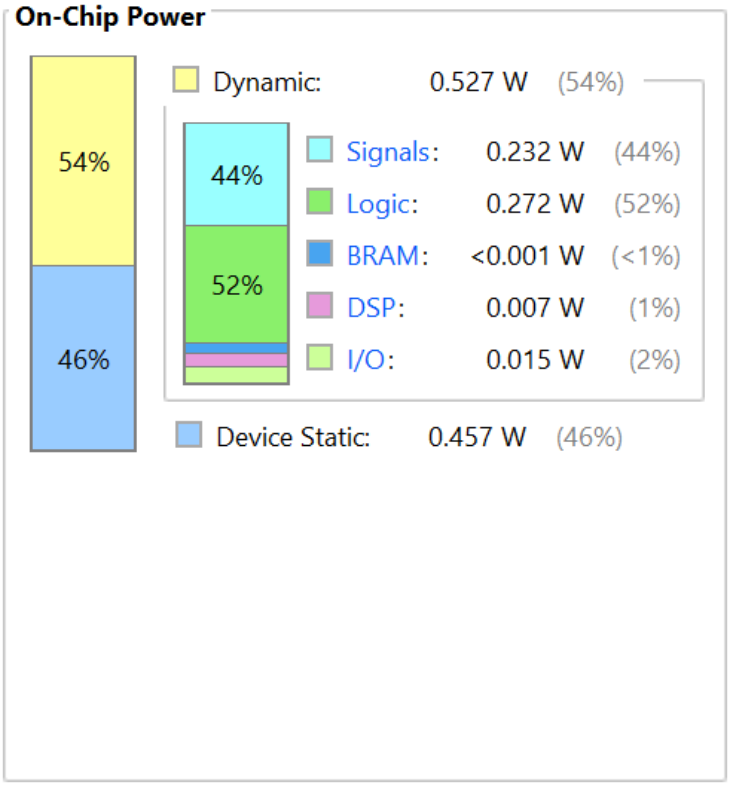
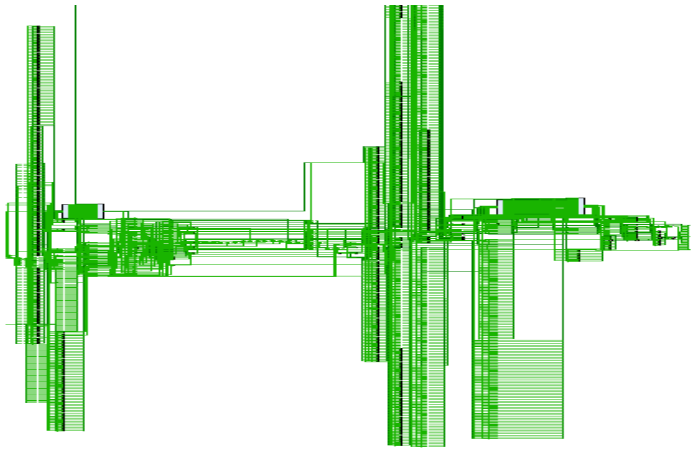
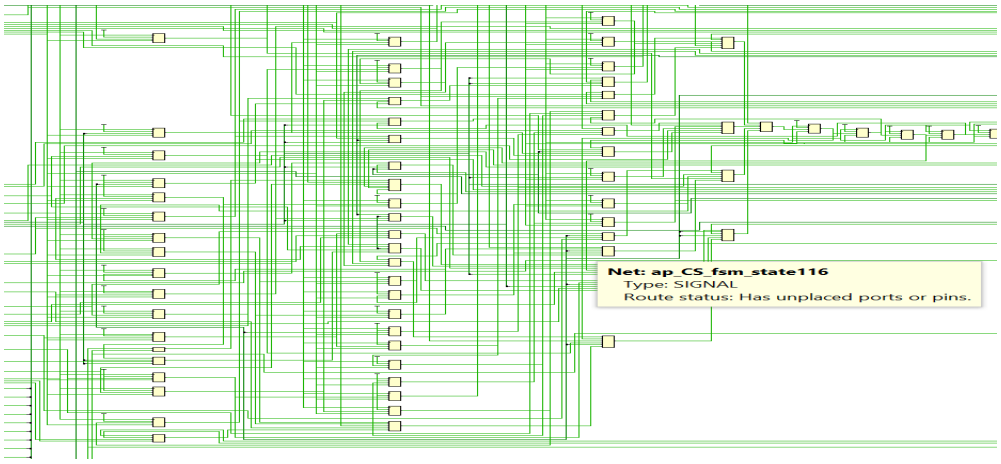


Figure 5.3: This image shows the power consumption diagram that occurs while using the FPGA board while running the MLR model with 3 features. Power consumption is divided into static and dynamic. Static power consumption occurs from the moment the board is powered on, while dynamic power consumption occurs only while the model is running.



(a) Schematic of the actual FPGA board implementation of the MLR model with 3 features.



(b) Schematic of the actual FPGA board implementation of the MLR model with 3 features, zoomed to show the actual connection net between data locations

Figure 5.4: Schematic of the actual FPGA board implementation.