Internship Report

IP SVM

Autour : Yong LI

17 septembre 2023

Table des matières

1 Introduction to Support Vector Machines (SVMs)

In the landscape of machine learning and pattern recognition, Support Vector Machines (SVMs) stand as a foundational pillar, renowned for their prowess in classification and regression tasks. Born from the realm of statistical learning theory, SVMs offer a powerful framework that transcends traditional classification methods, robustly handling complex and high-dimensional data.

2 Training the SVM Model with Python

2.1 Data Preparation and Preprocessing for SVM Training

In the realm of Support Vector Machines (SVMs), the foundation of accurate and insightful analysis lies in the data that fuels the model. Before delving into the intricacies of SVM algorithms, it's imperative to address the critical phase of data preparation. This section sheds light on the process of organizing data within an Excel table and introduces the pivotal role that the "pandas" package plays in extracting and managing data from Excel.

2.1.1 Function used

Here are some essential functions that are used to import and organize data using the pandas library in Python :

- **read_excel() :** The read_excel() function, enables us to effortlessly import data from Excel files into pandas DataFrames.
- **DataFrames :** DataFrames stand as pandas' core data structure, providing an intuitive way to organize and manipulate data.
- **drop() :** The drop() enables us to eliminate rows or columns containing null values.
- **loc[] :** The loc[] function is a label-based indexer, designed to locate and select data based on specified row and column labels within a DataFrame.
- **head() :** The head() function is designed to offer a concise preview of the initial rows within a DataFrame.

2.1.2 Implementing SVM using Python and Scikit-Learn

In this section, we'll walk through a basic implementation of SVM using the popular Python library, Scikit-Learn.

Required Libraries :

- **sklearn.model_selection** This module provides tools for model selection and evaluation. In our case, it is used for splitting datasets into training and testing sets. train_test_split is function that we used, it's used to split a dataset into two subsets : a training set and a testing (or validation) set. The model is trained on the training set and evaluated on the testing set to assess its performance.
- **sklearn.svm** This module contains the implementation of Support Vector Machines (SVMs), **SVC :** The SVC (Support Vector Classification) class within this module is used to create an SVM classifier and train an SVM model for classification tasks.

- **sklearn.metrics** This module includes various metrics and scoring functions for evaluating machine learning models. We used to measure the performance of classifiers and regressors. **accuracy_score :** This function within the sklearn.metrics module computes the accuracy of a classification model. It compares the predicted labels to the true labels and calculates the ratio of correct predictions to the total number of predictions.

Function used :

- **train_test_split() :** The train_test_split function is used to split the dataset into two or more subsets for training and testing machine learning models.
- **svm.SVC() :** The svm.SVC() stands for Support Vector Classification. It's a class that represents a Support Vector Machine (SVM) classifier.

  Key Parameters when using 'rbf' kernel function :

  · **C :** This is the regularization parameter, which controls the trade-off between maximizing the margin (distance between decision boundary and support vectors) and minimizing classification error. Smaller values of C make the margin wider but might allow misclassifications, while larger values of C reduce the margin but aim for correct classifications.

  · **kernel :** The kernel function used to transform the input data into a higher-dimensional space. Our choice is'rbf' (radial basis function).

  · **gamma :** The kernel coefficient, which determines the shape of the decision boundary. Smaller values make the decision boundary smoother, while larger values make it more complex. When creating the model we set the γ to 1.388 with $\sigma = 0.6$
- **fit(X, y) :** Trains the SVM classifier on the input features X and corresponding labels y.
- **predict(X) :** Predicts the labels for a new set of input features X using the trained classifier.

2.1.3 Collecting trained SVM model

We want to extract specific information from an SVM model created in Python for the purpose of implementing the model test part in VHDL, We can access various attributes and parameters of the trained SVM model. Here's how we extracted the information :

The decision function :

$$g(x) = \sum_{n=1}^{N} \alpha_n y_n K(x_i, x) + b$$

The parameters to be collected include '$\alpha \cdot y$' and 'b' values for each support vector, as well as the complete set of support vectors.

Function used :

- **Support Vectors :** We can access the support vectors (data points from the training set that are used to define the decision boundary) using the support_vectors_ attribute.
- ω **:** The dual_coef_ attribute holds the product $\alpha \cdot y$ in the decision function.
- **Intercept (Constant) in Decision Function :** The intercept_ attribute provides the independent term (constant) in the decision function.

3 Implementing the SVM Model with VHDL

3.1 Introduction to Hardware Implementation of Machine Learning Models

The increasing demand for high-performance and energy-efficient computing solutions has led to a growing interest in exploring hardware implementations of machine learning models. Hardware acceleration, through Field-Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs), offers a promising avenue to address these challenges. By directly translating machine learning algorithms into dedicated hardware circuits, it becomes possible to achieve substantial speedup and energy savings. This section of the report delves into the transition from a software-based Support Vector Machine (SVM) model trained using Python to a hardware implementation in VHDL. We explore the foundational concepts of hardware design, discuss the rationale behind this shift, and examine the key considerations associated with deploying machine learning models in hardware.

3.2 Mapping SVM Algorithm to VHDL Architecture

3.2.1 Overview of the Mapping Process :

As we understand, the basic content of the SVM model in hardware language is a mathematical expression and then a computation of his results. The derivation of a collation formula has been introduced in the previous section, so we are actually implementing computational processing at the hardware level with multiple plus and subtract multiplication operations plus a power operation, so then we'll introduce you to the general idea by combining text and pictures.

The mathematical expression implemented :

$$g(x) = \sum_{n=1}^{N} \alpha_n y_n e^{-\frac{||X_n - X||^2}{2\sigma^2}} + b$$

The computation :

$$g(x) > 0 \ ?$$

3.2.2 Algorithm Decomposition :

Input values of the model : For all the data to be processed we need to have an X value which in our case is an 8-dimensional vector

Post-Training Model Characteristics : After successful training, the SVM model exhibits notable properties that significantly influence its subsequent hardware implementation. The model identifies approximately 20 support vectors, denoted as N=25 for this study. Each support vector is characterized by several attributes, the foremost being the X value—a multidimensional vector spanning 8 dimensions. Additionally, the support vector carries the y value, signifying its classification, as well as the associated index, termed 'alpha'. To facilitate the hardware design, we establish a fixed value for sigma, set at 0.6, which remains consistent across all computations. The b variable, another essential component for SVM decision boundary calculation, is precomputed through Python and directly integrated into the hardware architecture.

So the simplified formula is :

$$g(x) = \sum_{n=1}^{25} \alpha_n y_n e^{-\gamma||X_n - X||^2} + b$$

3.2.3 Data Representation and Conversion :

In this subsection, we outline our strategy for maintaining precision while performing SVM calculations within the VHDL hardware architecture. Given that the input data possesses a precision of 0.01, we've opted to perform all internal calculations using a higher precision of 0.001. This choice ensures that we retain accuracy throughout the computations. To represent the data efficiently and consistently, we employ a fixed-point format, leveraging the std_logic_vector data type.

Fixed-Point Representation : Our fixed-point representation allocates 10 bits to the fractional part of the number, providing a precision of approximately 0.001 per increment. This design choice guarantees that the precision of the calculated values remains intact, preventing information loss due to truncation. Furthermore, we dynamically adjust the integer part of the fixed-point representation during calculations. By optimizing the integer part bit-width based on the range of values encountered, we minimize the number of required bits, contributing to efficient use of hardware resources.

Benefits of Consistent Precision : The adherence to a consistent precision of 0.001 across internal calculations ensures that no rounding errors or truncation effects compromise the SVM algorithm's accuracy. Maintaining uniform precision throughout various modules within the hardware design enables seamless data flow and consistent results. Moreover, as detailed in the subsequent sections, our approach to managing input and output value bit-widths further contributes to a holistic and optimized hardware design.

Summary : In summary, our data representation strategy underscores our commitment to maintaining precision during SVM calculations in the VHDL hardware architecture. The use of a fixed-point format, with a 10-bit fractional part and dynamically allocated integer part, guarantees accuracy and prevents information loss. This foundational approach serves as a cornerstone for achieving reliable and efficient hardware acceleration of the SVM algorithm.

3.3 Hierarchical Architecture Design for SVM Hardware Implementation

The successful translation of the Support Vector Machine (SVM) algorithm into VHDL architecture demands not only an intricate understanding of the algorithm itself but also a well-structured hierarchy that facilitates seamless interaction between various modules. In this section, we delve into the intricacies of the hierarchical architecture design for our SVM hardware implementation. By organizing the design into well-defined modules and sub-modules, we create a coherent framework that encapsulates the SVM algorithm's core computations, control logic, and data flow. This hierarchical approach enables efficient resource utilization, ease of debugging, and modularity—a fundamental principle in the realm of hardware design. We now embark on an exploration of the architectural hierarchy, uncovering the layers that make up this intricate ecosystem of SVM execution within the VHDL framework.

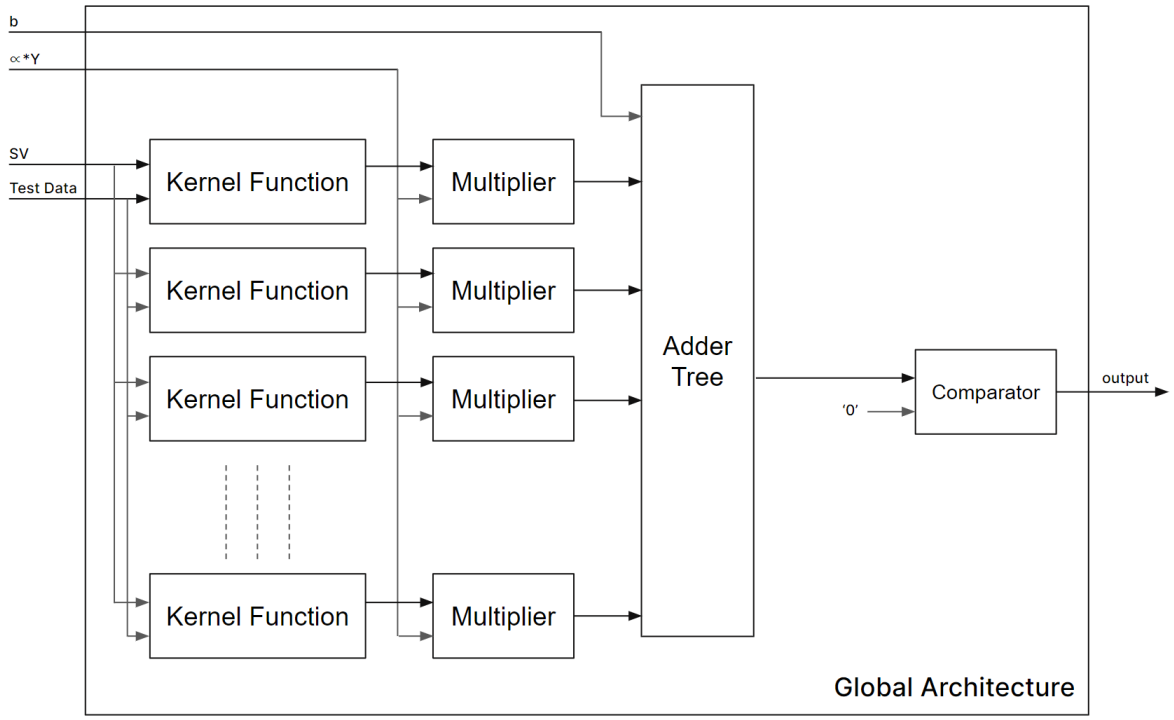

FIGURE 1 – Global Architecture

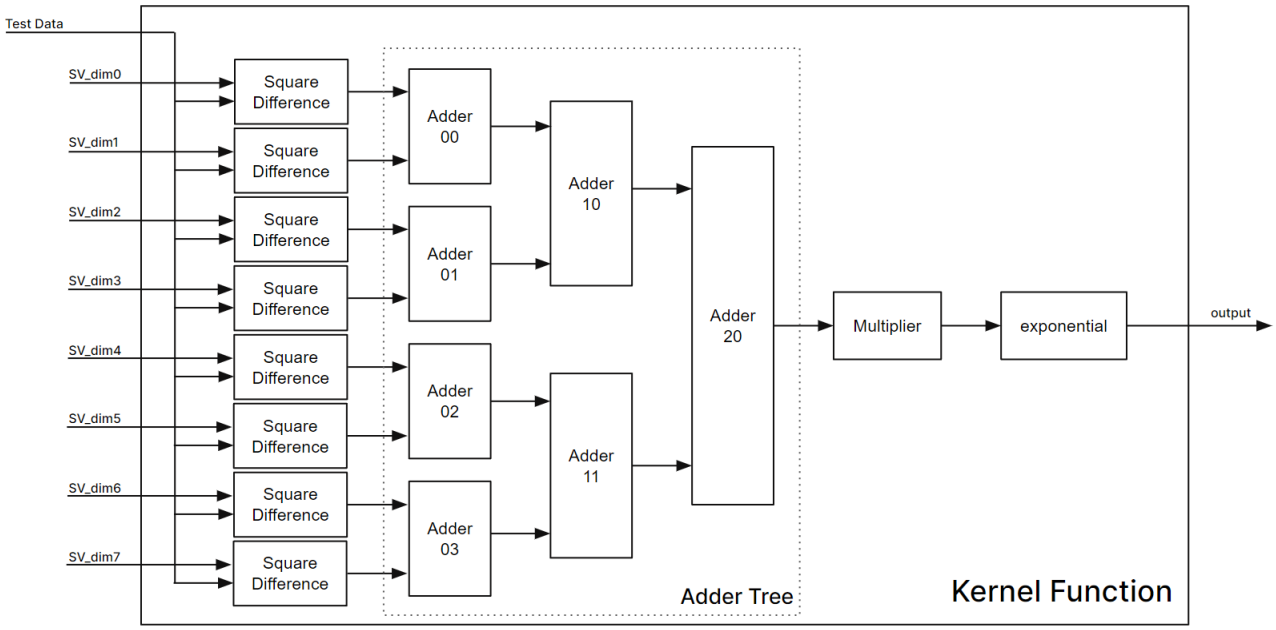

FIGURE 2 – Kernel Function

So, there are several models or components that work together :

- Square Difference Block
- Multiplier Block
- Adder Block
- Exponential Block

3.3.1 Square Difference Block :

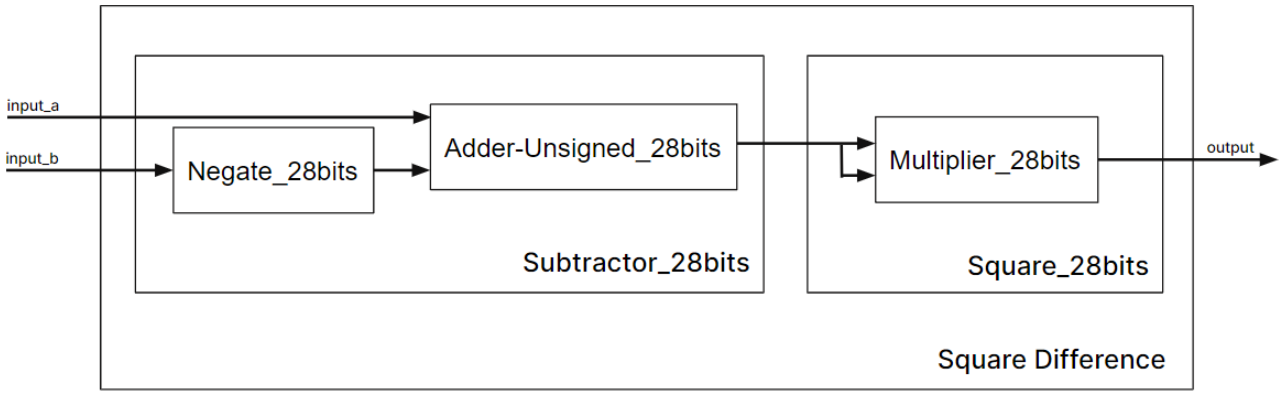

FIGURE 3 – Square Difference Block

At the lowest stratum of our hierarchical architecture resides the Square Difference block—an elemental construct enabling the computation of squared differences pivotal to the SVM algorithm. This block encapsulates a sequence of fundamental operations, commencing with the "Negate_28bits" operator. By employing this operator, we ingeniously invert the sign of a value, an operation germane to calculating the difference between two values. Aided by the "Adder_Unsigned_28bits" operator, we effectuate the numerical difference, priming the canvas for subsequent calculations. The "Multiplier" block affords us the squared difference of two values.

Bit-width Selection (Balancing Precision and Resource Efficiency) : The meticulous selection of bit-widths for our operators reflects a harmonious balance between precision and resource constraints. Adhering to a 28-bit framework is guided by the discernment that the highest input value remains within the 50,000 range. This choice precludes overflow concerns, ensuring integrity throughout our calculations. Subsequently, the output of our "Multiplier" block, spanning 44 bits, substantiates the smallest bit-width that accommodates its results. Notably, while the fraction part retains a steadfast 10-bit composition, the integer segment demonstrates adaptive evolution. Transitioning from an 18-bit signed value to a 34-bit unsigned representation.

3.3.2 sub-modules : Negate_28bits Block :

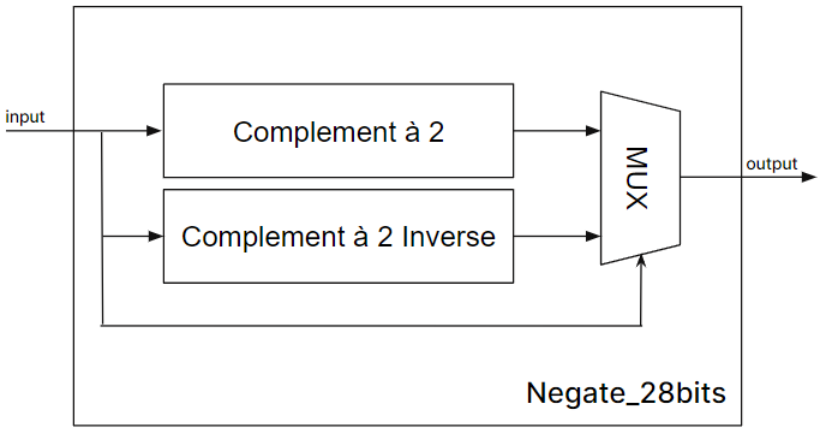

FIGURE 4 – Negate_28bits Block

The bedrock "Negate_28bits" block stands as the linchpin for dynamic sign transformation—a pivotal function necessary for SVM operations. Within its construct resides an adaptive duality, choosing between two operators : complement à 2 or its inverse. The decision-making pivot hinges on the most significant bit of the input. The "Negate_28bits" block's prowess in traversing the subtle realm of binary values serves as a cornerstone, setting the stage for subsequent calculations.

3.3.3 sub-modules : Multiplier_28bits

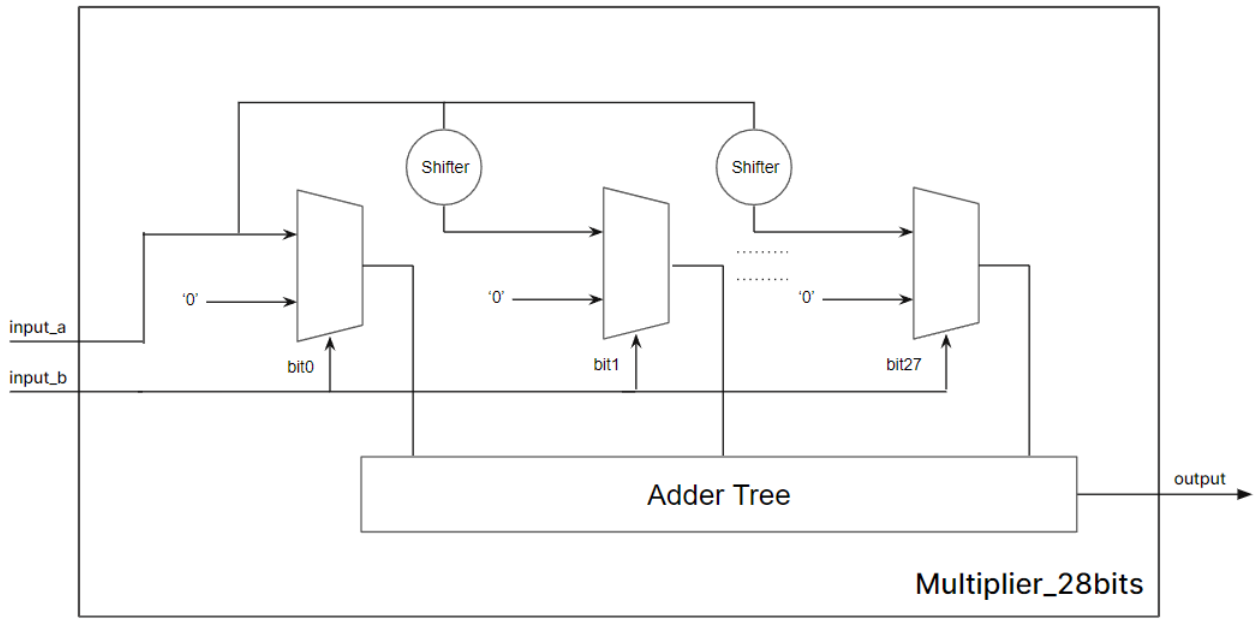

FIGURE 5 – Multiplier_28bits Block

The bedrock of our hierarchy, the "Multiplier" block, encapsulates the very essence of binary product transformation—an operation deeply rooted in the fundamentals of multiplication. Guided by the binary truth that

multiplication by a single binary digit yields either 0 or the original value, the "Multiplier" block works by doing shift and addition.

3.3.4 sub-modules : Square_28bits Block :

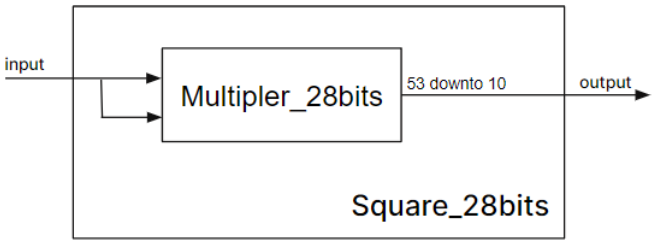

FIGURE 6 – Square_28bits Block

The "Square_28bits" block is achieved through a strategic integration with the "Multiplier_28bits" module. However, when we shift our focus towards precision and data representation, as well as the balance between resource optimization and computational accuracy. Leveraging fixed-point representation, we harmonize the representation of integer and fraction parts. In a stride towards resource efficiency, we truncate the fraction part to a ten-bit segment, reflecting a precision of 0.001. This astute pruning aligns with our SVM's specific needs, discarding excess precision while maintaining the essence of the data.

3.3.5 Kernel Function Block :

Bit-width Selection The input values come in different sizes, ranging from the larger value ±50000 to the smaller values (±10 and ±2). To accommodate the size of these data and not have an overly complex structure, all adder components within the block were chosen to have a bit width of 44 bits with 10 bits allocated for the fractional part, ensuring that the sum of squared differences does not exceed the capacity of the adders. The output data of the multiplier block is still selected to be 44 bits wide, and the decimal part is allocated 10 bits. This is also because when multiplied by gamma (1.388 when sigma is 0.6), it will not exceed 44 bits. The output of the exponent operator (exp) is a value between 0 and 1, so the output is 2 bits wide for the integer part and 10 bits for the fractional part. These thoughtful bit-width decisions are designed to maintain the accuracy and efficiency of the SVM hardware implementation, ensuring numerical stability and precision throughout the process.

3.3.6 sub-modules : Exponential Block :

The primary function of the exponential block is to compute the value of exp(-x) based on the input value, where the input consists of 34 bits for its integer part. However, upon analyzing the results of exp(-x), it becomes evident that when the input exceeds 7.8 (equivalent to approximately 8000 with the 10 bits allocated for the fractional part), the output becomes smaller than our precision threshold of 0.001 used during calculations. Managing approximately 8000 such cases manually, involving decimal-to-binary conversions and intricate calculations, presents a substantial coding challenge with a higher likelihood of errors. Given my familiarity with the C programming language, I've opted for a more efficient approach by implementing this particular block in C.

This decision has simplified the task into a straightforward "when" case, reducing complexity while maintaining accuracy in the hardware design.

Library used in C :

- stdio.h, stdlib.h : To write into text file and some basic function.
- math.h : To calculate the exponential value.

Function created in C : We'll focus on discussing the functions created in C for our specific purpose. These functions serve two main roles : one for converting between binary and decimal values, and the other for writing values to a file. Specifically, for each input value 'x,' we utilize a for loop to generate code of the form "exp(-x) <= when input = x," and then we write this code into the file.

4 Verification and Testing of the SVM Model

4.1 In python

With the scikit-learn library, we can effortlessly construct our SVM model. Testing the model is equally straightforward, thanks to the prior separation of data into training and testing sets. To assess the model's performance on the testing data, we simply employ the accuracy_score function. This function provides us with the classification accuracy, a valuable metric for evaluating the model's effectiveness

4.2 In VHDL

I have meticulously developed individual testbench files for each VHDL block to rigorously verify the functionality of each component. These testbenches serve to ensure that each block performs its intended function correctly. Furthermore, you can access all the testbenches and review the synthesis results within the Vivado project, providing a comprehensive view of the verification and synthesis processes.