VIETNAMESE-GERMAN UNIVERSITY

PROGRAMMING 2 PROJECT REPORT

THE TINY PROJECT

Major: Computer Science

Instructor	**:**	**Prof. Huỳnh Trung Hiếu , PhD**
Student 1	:	Đoàn Quốc Hưng - 10423049
Student 2	:	Nguyễn Thanh An - 10423003
Student 3	:	Lương Gia Bảo - 10423010
Student 4	:	Đoàn Ngọc Bảo - 10423009
Student 5	:	Lê Minh Huy - 10423044

---o0o---

HO CHI MINH CITY, 06 /2025

Table of Contents

Table of Contents 1

CHAPTER 1: INTRODUCTION 2

CHAPTER 2: THEORETICAL BACKGROUND 4

I. Vectors 4

II. Matrices 4

III. Linear Systems 4

IV. Linear Regression 5

CHAPTER 3: METHODOLOGY 6

I. Vector Class 6

II. Matrix Class 7

III. LinearSystem Class 8

IV.PosSymLinSystemClas10

V. GeneralLinSystem Class 11

VI. Main Regression Class 13

CHAPTER 4: TEST CASE AND RESULTS 16

I. Dataset 16

II. Experimental Setup 16

III. Output 17

CHAPTER 5: CONCLUSION 20

CHAPTER 1: INTRODUCTION

In this project, we created a C++ program that solves systems of linear equations and applies that knowledge to build a linear regression model. The core of the program is based on three main classes: Vector, Matrix, and LinearSystem. These classes represent the key mathematical structures and operations needed to solve equations of the form Ax = b, and to use those solutions for prediction tasks through linear regression.

Linear systems and regression are widely used in fields like data science, machine learning, statistics, physics, and engineering. By learning how to build and solve these systems through programming, we gain valuable tools to solve real-world problems effectively.

This project is divided into two parts:

Part A: Building the Core System

We created and handled:

- The Vector, Matrix, and LinearSystem classes
- The program includes key algorithms such as Gaussian Elimination with pivoting, the Moore-Penrose pseudo-inverse, and the Conjugate Gradient method for symmetric positive-definite systems.
- Floating-point precision, memory management, and edge cases like singular or inconsistent matrices.

Part B: Applying Linear Regression

- We used the solver to implement linear regression on real-world data from the UCI Computer Hardware dataset.
- This part included data processing, model training, parameter estimation, and error analysis using Root Mean Square Error (RMSE) to measure accuracy.
- The goal was to predict CPU performance based on multiple input features.

During development, we faced several challenges:

- Ensure the correctness of matrix operations (elimination and back-substitution).
- Deal with numerical stability and precision.
- Design a clean and flexible class structure
- Make the code easy to extend and maintain

Despite the challenges, this project significantly improved our understanding of linear algebra and regression theory. It also gave us practical experience with object-oriented programming in C++. Combining mathematical reasoning with software design deepened our appreciation for how abstract theories can become concrete solutions through code.

Before diving into the code, it's helpful to have a solid theoretical understanding of linear systems, vector and matrix operations, and linear regression principles. This foundation will make it easier to follow our design choices and logic.

CHAPTER 2: THEORETICAL BACKGROUND

I. Vectors

A vector is a one-dimensional list of numbers. It represents both direction and length in space. Typically, a vector is constructed as a dynamic array that can expand in size and supports basic math operations like adding, subtracting, and multiplying .

II. Matrices

A matrix is a table of numbers arranged in rows and columns. It is used to describe linear transformations or systems of linear equations.

Some important operations with matrices include finding the determinant, inverse, and doing multiplication with other matrices or vectors. These operations are especially useful when solving systems of equations or performing linear regression.

III. Linear Systems

A linear system is a group of equations that use the same variables and follow a straight-line relationship. We can write it as:

$$Ax = b$$

where:

+ A is a matrix containing the equation coefficients,
+ x is a vector of unknown values we want to find,
+ b is a vector of results (the right-hand side).

If A is square and non-singular ($\det(A) \neq 0$), then the system has a unique solution. One basic method to solve it is Gaussian Elimination with Partial Pivoting. This method transforms the matrix into a simpler form, and then we work backward to find the solution.

If A is symmetric and positive definite, the Conjugate Gradient Method is often better. It’s an iterative method that works faster for large systems with lots of zeros (sparse matrices).

IV. Linear Regression

In Part B of the project, we build a linear regression model. Linear regression is a method used to model the relationship between a target value and several input features. It assumes that the target can be predicted as a linear combination of the features. This approach is commonly used for prediction tasks, such as estimating CPU performance based on hardware specifications.

$$PRP=x_1*MYCT+ x_2*MMIN+ x_3*MMAX+ x_4*CACH+ x_5*CHMIN+ x_6*CHMAX$$

The quality of the model is often measured using Root Mean Square Error (RMSE), which shows how close the predicted values are to the actual ones.

CHAPTER 3: METHODOLOGY

I. Vector Class

The Vector Class is created to work with vectors. In this class, we will store, manage, and do basic math with vectors.

The class has 2 main variables:

+ *mSize:* the size of the vector
+ *mData:* the data stored in the vector (a dynamic array of type double)

First, we have the constructor ***Vector(int size)***, which creates a vector with the given size and sets all elements to 0.0. This gives a clear starting point for other calculations. We also use ***assert*** to make sure that the size of vector is always positive

The copy constructor ***Vector(const Vector& other)*** is used to make a deep copy of another vector. It creates new memory and copies each value one by one, so the two vectors don't share the same memory.

The destructor ***~Vector()*** deletes the memory used by the vector. This helps prevent memory leaks.

The assignment ***operator operator=*** allows us to assign one vector to another. It first deletes the old data, then copies the size and values from the other vector.

The function ***getSize()*** simply returns the number of elements in the vector.

To access elements in the vector, we have two ways:

+ ***operator[]***: get or change elements starting from index 0.
+ ***operator()***: use an index starting from index 1.

Both functions use ***assert*** to make sure the index is not out of range.

The Vector class also supports math operations between vectors, and between a vector and a single number (a scalar). For example:

+ ***operator+*** adds two vectors.

+ ***operator-*** subtracts one vector from another.
+ ***operator**** (between two vectors) calculates the dot product and returns a single number.

For setting values, ***manualAssign()*** lets the user input values from the keyboard, while ***assign(double* val)*** copies values from a given array.

The ***display()*** function shows the vector elements in the console, making it easier to read.

Overall, this class helps us work with vectors easily. We can create vectors, set their values, access elements, and do math with them.

II. Matrix Class

The Matrix Class is created to work with matrices. This class stores matrix data, performs memory management, and supports various mathematical operations such as addition, multiplication, transposition, and inversion.

The class has 3 main variables:

+ *mNumRows:* the number of rows in the matrix
+ *mNumCols:* the number of columns in the matrix
+ *mData:* a 2D dynamic array that stores the matrix elements

The constructor ***Matrix(int rows, int cols)*** creates a matrix of given dimensions and allocates memory dynamically using a helper function ***allocateMemory()***. It uses assert to make sure rows and columns are positive.

Rows() and ***Cols()*** return the number of rows and columns respectively.

There are some functions which are used to perform calculations on matrices. Each operation returns a new Matrix object as the result:

+ ***operator+*** adds two matrices of the same size.
+ ***operator-*** subtracts one matrix from another of the same size.
+ ***operator* (Matrix-Matrix)*** multiplies two matrices if their inner dimensions match.
+ ***operator* (Matrix-Scalar)*** multiplies all elements by a constant value.

The ***transpose()*** function returns a new matrix that is the transpose of the original matrix

The ***determinant()*** function works recursively using Laplace expansion.

+ For 1x1 and 2x2 matrices, it uses direct formulas.
+ For larger matrices, it computes minors and cofactors.
+ It assumes the matrix is square (n x n) and uses assert to check this condition.

The ***inverse()*** function computes the matrix inverse using the Gauss-Jordan elimination method:

+ It builds an augmented matrix **[A | I]** and reduces it to **[I | A^{-1}].**
+ It assumes the matrix is square and invertible (non-zero determinant).
+ It uses assert to check for non-zero pivots.

The ***pseudoInverse()*** function handles both overdetermined and underdetermined systems:

If the matrix has more rows than columns, it uses the formula:

$$A^+ = (A^TA)^{-1} A^T$$

If it has more columns than rows, it uses:

$$A^+ = A^T(AA^T)^{-1}$$

manualAssign() allows the user to input matrix values manually

display() prints the matrix on the console. The output format is a bracketed grid of numbers.

In conclusion, this Matrix class allows us to build, display, and perform algebraic operations on matrices. It uses dynamic memory, supports matrix-matrix and matrix-scalar operations, and includes tools for computing transpose, determinant, inverse, and pseudoinverse.

III. LinearSystem Class

The **LinearSystem** class is designed to represent and solve a system of linear equations using **Gaussian Elimination**. It makes use of the previously defined Matrix and Vector classes to express the system in the standard form **Ax = b**.

This class contains the following variables:

+ *mSize:* the size of the system (number of equations/variables).
+ *mpA:* a pointer to a dynamically allocated **Matrix** object representing the **coefficient matrix A.**
+ *mpb:* a pointer to a dynamically allocated **Vector** object representing the **right-hand side vector b.**
+ *augMatrix:* a 2D dynamic array to store the augmented matrix **[A|b]** for computation.

First we have the constructor ***LinearSystem(Matrix A, Vector b)***, which initializes the linear system using matrix A and vector b. It checks for size compatibility and returns an error if the dimensions don't match. It also allocates memory for the augmented matrix and copies the values of A and b into it.

The destructor ***~LinearSystem()*** deallocates all dynamically allocated memory (augMatrix, mpA, and mpb) to prevent memory leaks.

We have deleted copy operations to delete the copy constructor and assignment operator to prevent copies or misuse of dynamic memory

+ ***LinearSystem(const LinearSystem&) = delete***
+ ***LinearSystem& operator=(const LinearSystem&) = delete***

This class also provides several functions for performing matrix operations and displaying the matrix on the console.

- ***int getSize():*** returns the number of equations (size of the system).
- ***void assign(Matrix* pA, Vector* pb):*** copies the data from matrix A and vector b into the augmented matrix augMatrix.
- ***void swap_row(int i, int j):*** swaps two rows of the augmented matrix.
- ***nt forwardElim():***
    + performs forward elimination to convert the augmented matrix into an upper triangular form.
    + includes partial pivoting by swapping rows when necessary.

  + returns the row index if a zero pivot (singular matrix) is found, otherwise returns -1.
- ***Vector backSub():***
  + performs back substitution on the triangular matrix obtained from forwardElim() to find the solution vector x.
  + returns a Vector containing the solution.
- ***Vector solve():***
  + This is the main function to solve the system using Gaussian Elimination.
  + It first performs forward elimination. If the matrix is singular, it returns an error and indicates that the equation has either no solution or infinitely many solutions
  + Otherwise, it calls the function backSub() to return the solution
- ***void display():*** prints the augmented matrix in console

IV. PosSymLinSystem Class

The PosSymLinSystem class is designed to solve linear systems where the matrix **A** is **symmetric** and **positive definite**, using the **Conjugate Gradient (CG)** method.
It inherits from the LinearSystem class and uses the Matrix and Vector classes to store and work with matrices and vectors.
The constructor ***PosSymLinSystem(Matrix A, Vector b):***

- Calls the parent constructor to store matrix A and vector b.
- Checks whether A is symmetric:
  + Loops through all off-diagonal elements A(i, j) and A(j, i).
  + Ensures they are equal within a small tolerance (1e-12).
  + If the matrix is not symmetric, it returns an invalid_argument exception. This symmetry check ensures the CG algorithm will work correctly, as it relies on matrix symmetry.

Vector solve(): *This method applies the Conjugate Gradient algorithm to solve the equation Ax = b. The steps involved:*

Initialize:

- Set solution vector x to all zeros.
- Compute initial residual r = b - A*x → simplifies to r = b.
- Set initial search direction p = r.
- Compute rr_old = r^T * r.

Iteration (up to n times, where n is matrix size):

- Compute Ap = A * p using nested loops.
- Compute step size alpha = $\frac{r^T * \mathrm{r}}{p^T * Ap}$
- Update the solution: x = x + alpha * p
- Compute new residual: r_new = r – alpha * Ap
- Check for convergence: if the norm of r_new is smaller than a small tolerance (1e-10), return x.
- Compute beta = $\frac{r\_new^T * \mathrm{r\_new}}{r^T * r}$
- Update search direction: p = r_new + beta * p
- Set r = r_new and update rr_old

Overall, The PosSymLinSystem class checks matrix symmetry and solves efficiently through iteration, making it ideal for large, sparse systems.

V. GeneralLinSystem Class

The **GeneralLinSystem** Class is created to work with general linear systems of the form Ax = b, where A can be a non-square matrix. This class stores the matrix and right-hand-side vector, manages memory, and provides multiple methods to solve the system, including pseudoinverse and Tikhonov regularization.

The class has 4 main variables:

+ mRows: the number of rows in matrix A
+ mCols: the number of columns in matrix A

+ mpA: a pointer to a dynamically allocated Matrix object representing A
+ mpb: a pointer to a dynamically allocated Vector object representing b

The constructor ***GeneralLinSystem(const Matrix& A, const Vector& b)*** takes a matrix A and a vector b and sets up the system Ax = b. It verifies that the number of rows in A equals the size of vector b using an exception check. It then dynamically allocates and copies A and b using the new operator.

The destructor ***~GeneralLinSystem()*** releases the memory allocated for matrix A and vector b by using delete. This avoids memory leaks.

Rows() returns the number of rows in matrix A.

Cols() returns the number of columns in matrix A.

The solvePseudoInverse() function solves Ax = b using the Moore–Penrose pseudoinverse, which provides a least-squares solution for inconsistent or underdetermined systems. Steps:

+ Compute the pseudoinverse A^+ = A.pseudoInverse() — a matrix of size $(n \times m)$.
+ Multiply $A^+ * b$ to compute the result vector x of length n.
+ The multiplication is done manually using double for loops and 1-based indexing.
+ The method assumes that the pseudoinverse function is implemented correctly and that the matrix involved in inversion has non-zero determinant

The ***solveTikhonov(double lambda)*** function solves Ax = b using Tikhonov regularization. This method is useful for non-square matrices. Steps:

+ Compute the transpose of A: A^T.
+ Compute $A^T * A$, resulting in a square $(n \times n)$ matrix.
+ Add λI (lambda times the identity matrix) to the result to stabilize the inverse.
+ Invert the matrix $(A^T A + \lambda I)^{-1}$.
+ Compute the vector $A^T * b$.
+ Multiply the inverse matrix by this vector to get the result x.

Overall, the GeneralLinSystem class manages memory safely and provides flexible, numerically robust solutions to real-world linear systems.

VI. Main Regression Class

The ***main_regression.cpp*** file is created to perform linear regression on a dataset from the file "machine.data". This program loads data, splits it into training and testing sets, builds the regression model using the normal equation, and finally computes prediction errors using RMSE.

We use a real-world dataset from the UCI Machine Learning Repository. The model assumes that performance (PRP) can be predicted by a linear combination of the features:

$$PRP=x_1*MYCT+ x_2*MMIN+ x_3*MMAX+ x_4*CACH+ x_5*CHMIN+ x_6*CHMAX$$

The program includes several key parts and helper functions. For instance, the ***function loadData(...)*** reads a dataset from a CSV file named "machine.data". This function stores:

+ dataX: a vector<vector<double>> with shape 209×6, where each row contains six numeric features (columns 2 to 7 in the file).
+ dataY: a vector<double> with 209 values, where each value is the target output PRP (column 8 in the file).

Each line is split by commas using a stringstream, and the numeric values are converted using atof. If any line does not contain exactly 10 fields, a warning is printed, and the line is skipped.

The function ***trainTestSplit(...)*** splits the data indices [0..N-1] into two groups:

+ trainIdx: 80% of the data used for training.
+ testIdx: 20% of the data used for testing.

From the training indices, two structures are built:

+ Matrix X_train: a matrix of size (nTrain × 6) containing feature values.

+ Vector y_train: a vector of size nTrain containing the PRP values.

Each feature vector is inserted using X_train(ii+1, j+1) and each PRP value using y_train(ii+1) (1-based indexing, as used in the custom Matrix/Vector classes).

To solve the normal equation for linear regression:

$$w = (X^T X)^{-1} X^T y$$

The program does the following:

- Computes the transpose of X_train → X_t
- Calculates XtX = Xt * X_train
- Computes the vector Xty manually: each entry i is the dot product of row i of Xt with y_train
- Then, it creates a **GeneralLinSystem** object with XtX and Xty and solves for w using solvePseudoInverse().

The model is evaluated on the training set:

+ Predicted values are calculated using the dot product of w and each feature vector.
+ RMSE (Root Mean Square Error) is computed using the helper function computeRMSE(), which calculates the square root of the average squared difference between true and predicted values.
+ RMSE value shows how close the predicted values are to the actual ones, giving the model's error on the training data.

The same process is applied to the test data:

+ Predictions are made using the learned coefficients w.
+ RMSE is computed and printed to evaluate generalization performance.

When outputting, the program prints:

+ Total number of rows read from the file
+ Training and testing set sizes
+ The regression coefficients w
+ RMSE on both training and testing sets

As a result, the program provides a complete pipeline for simple linear regression by reading and processing raw data and randomly splitting the dataset into training and testing sets.

CHAPTER 4: TEST CASE AND RESULTS

I. Dataset

We use the Computer Hardware Data Set (machine.data) from the UCI Machine Learning Repository. This dataset consists of 209 records, each describing various hardware features of different computer systems, with the target variable being PRP (Published Relative Performance).

We extract 6 numerical features for regression:

- MYCT (Machine Cycle Time in nanoseconds)
- MMIN (Minimum main memory in kilobytes)
- MMAX (Maximum main memory in kilobytes)
- CACH (Cache memory in kilobytes)
- CHMIN (Minimum channels)
- CHMAX (Maximum channels)

The target variable is PRP (Published Relative Performance).

II. Experimental Setup

The dataset is split into:

+ Training set: 80% (167 samples)
+ Testing set: 20% (42 samples)

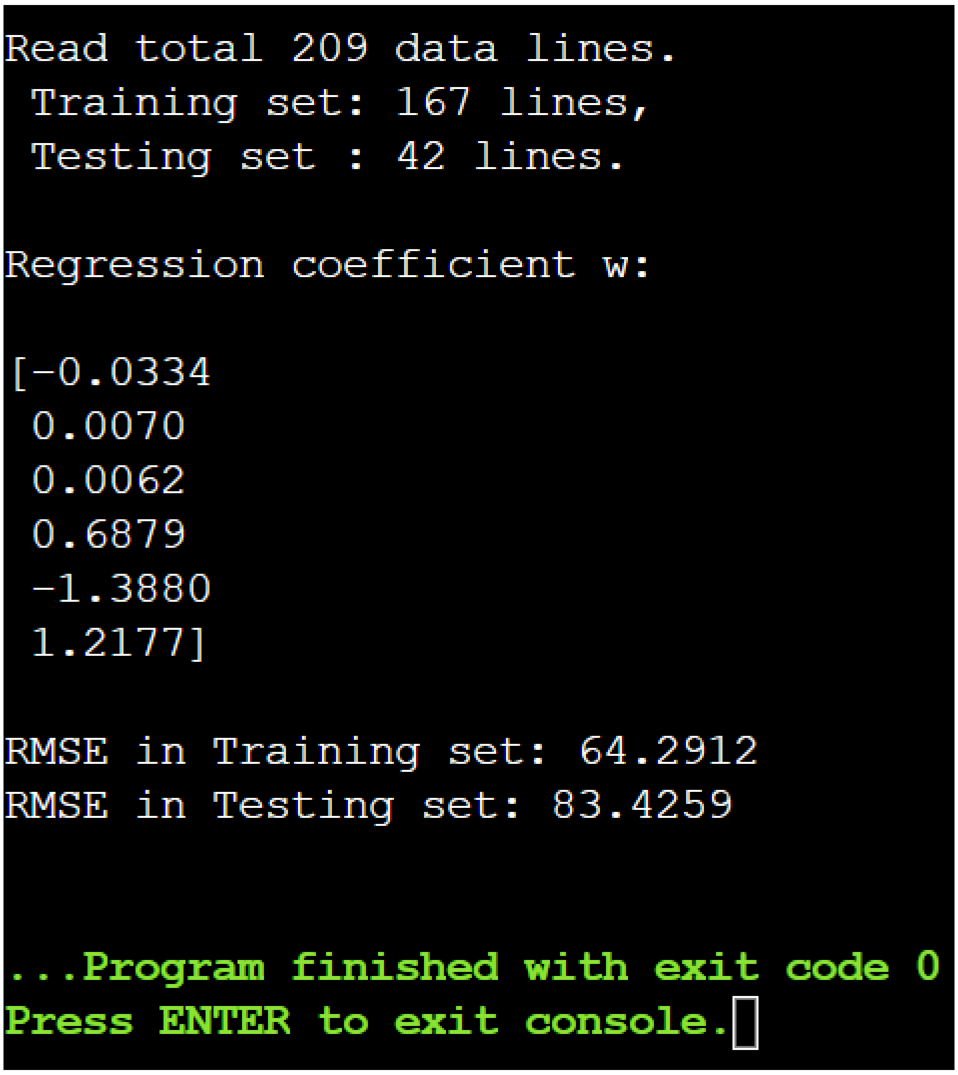
We perform multivariate linear regression using the normal equation:

$$w = (X^T X)^{-1} X^T y$$

Matrix operations are implemented via custom Matrix and Vector classes, and solved using a GeneralLinSystem solver.

We evaluate performance using RMSE (Root Mean Square Error) on both training and testing sets.

III. Output



```
Read total 209 data lines.
 Training set: 167 lines,
 Testing set : 42 lines.

Regression coefficient w:

[-0.0334
 0.0070
 0.0062
 0.6879
 -1.3880
 1.2177]

RMSE in Training set: 64.2912
RMSE in Testing set: 83.4259


...Program finished with exit code 0
Press ENTER to exit console.
```

The final output of our linear regression model consists of two parts:

1. The **learned regression coefficients (weights)**

Weights: [-0.0334, 0.0070, 0.0062, 0.6879, -1.3880, 1.2177]

These weights represent the **parameters** of the linear regression model. Each weight corresponds to the **influence of one feature** (input variable) on the predicted CPU performance. The general form of the prediction function is:

$$\hat{y} = w_0x_0 + w_1x_1 + \ldots + w_5x_5$$

Where:

- w_i is the coefficient for feature x_i
- $\hat{y}$ is the predicted output (CPU performance)

Weight	Interpretation
-0.0334	Slight negative impact: This feature has a very weak negative correlation with the target. May be near-zero influence.
0.0070	Slight positive impact: Very minimal influence, may be close to noise level.
0.0062	Another small positive effect: Suggests limited explanatory power.
0.6879	Strong positive contribution: This feature has a noticeable direct relationship with the CPU performance.
-1.3880	Strong negative impact: Likely a critical feature, but its value has an inverse effect on performance.
1.2177	Another strong positive weight, highly predictive in nature.

2. The **Root Mean Square Error (RMSE)** on the training and testing datasets.

- Training RMSE: 64.29
- Testing RMSE: 83.43

The RMSE measures the average deviation between the predicted values and the actual values. It is defined as: $RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}$

Overall Evaluation of Model Performance

The comparison between the training and testing RMSE values provides insight into how well our linear regression model generalizes to unseen data.

+ With a **training RMSE of 64.29**, the model appears to fit the training data reasonably well, suggesting it has successfully captured meaningful patterns during training.
+ However, the **testing RMSE of 83.43**, which is noticeably higher, reveals a **generalization gap** — a common phenomenon in machine learning where the model performs better on the data it has seen than on new data.

This gap indicates that the model may have slightly **overfit** the training data, meaning it may have learned some specific details that do not generalize well. While the increase in error is not extreme, it suggests there is room for improvement in terms of model robustness and predictive accuracy.

Possible solutions include applying **regularization**, refining the **feature selection or preprocessing**, or exploring **alternative modeling techniques** beyond linear regression if the relationships in the data are more complex.

In summary, the model shows a decent level of predictive capability but still exhibits typical signs of limited generalization, which is important to consider for real-world deployment.

CHAPTER 5: CONCLUSION

In conclusion, this project gave us valuable knowledge in object-oriented programming, especially in designing classes and working with key concepts such as inheritance and polymorphism. These principles are fundamental in coding particularly in software development, as they help create well-structured, maintainable, and scalable code.

We also gained important experience in applying mathematical concepts, especially linear systems and regression models, through C++ programming. By building the Vector, Matrix, and LinearSystem classes, we not only improved our understanding of mathematical operations but also learned how to implement them efficiently in code.

Throughout the project, we faced several challenges such as handling floating-point precision errors, designing class structures, and working with real-world data. These difficulties not only allowed us to improve our technical skills, but also teamwork, critical thinking, and debugging strategies. Although our model achieved reasonable results in predicting CPU performance, there is still room to make it better—especially by improving how it handles numbers and by adding support for more advanced regression methods.

In the end, this project helped us understand both the theory and practice of programming and math. It showed us how to turn ideas into working software, which is a valuable skill for our future studies and careers.

The complete source code and related materials for this project are available on GitHub:

HungExplorer/Tiny-Project