CSoC 2025 Intelligence Guild Assignment

Multivariable Linear Regression on California Housing Price Prediction

Suyash Ranjan
Mining(IDD) / 24154022

May 19, 2025

Abstract

This report presents the implementation and comparison of multivariable linear regression using three approaches: (i) pure Python, (ii) NumPy (vectorization), and (iii) scikit-learn. The California Housing Prices dataset is used to predict median house values, and the models are evaluated using standard regression metrics.

1 Introduction

The goal of this project is to understand and implement the core of linear regression from scratch and compare it with optimized and library-based implementations. This helps in building an intuitive understanding of machine learning algorithms.

2 Dataset Description

The dataset used is the **California Housing Prices** dataset, containing features such as latitude, longitude, housing median age, total bedrooms, households, total rooms, population, median income, and ocean proximity and some new features created to reduce the errors. The target variable is `median_house_value`.

3 Data Preprocessing

- Filled missing values using the median (for `total_bedrooms`).
- One-hot encoded the `ocean_proximity` categorical feature.
- Applied Z-score normalization to all numerical features except the target value.
- I created some new features by experimenting with the given features and came to choose some that reduced the errors for me.
- Split the dataset into training and validation sets by 8:2.

4 Model Implementations

4.1 Pure Python

Implemented multivariable linear regression using loops for prediction, cost computation, gradient calculation, and parameter updates (batch gradient descent).

4.2 NumPy Vectorized

Replaced all loops with NumPy operations. Used vectorized matrix multiplication for predictions and gradients, leading to a significant speedup in convergence.

4.3 Scikit-learn

Used `LinearRegression` from the `sklearn.linear_model` module to fit the data and evaluate performance.

5 Evaluation and Results

Evaluation Metrics

- Mean Absolute Error (MAE)
- Root Mean Squared Error (RMSE)
- Coefficient of Determination (R^2)

Comparison Table

Model	**MAE**	**RMSE**	**R^2 Score**	**Convergence Time (s)**
Pure Python (val)	51354.29	72226.36	0.6124	55.91
(train)	50325.86	69877.22	0.6323	55.91
NumPy Vectorized (val)	51354.29	72226.36	0.6124	0.29
(train)	50325.86	69877.22	0.6323	0.29
Scikit-learn (val)	50449.47	70713.66	0.6285	0.01
(train)	49079.12	67624.39	0.6556	0.01

Table 1: Performance comparison of all implementations on validation and training set

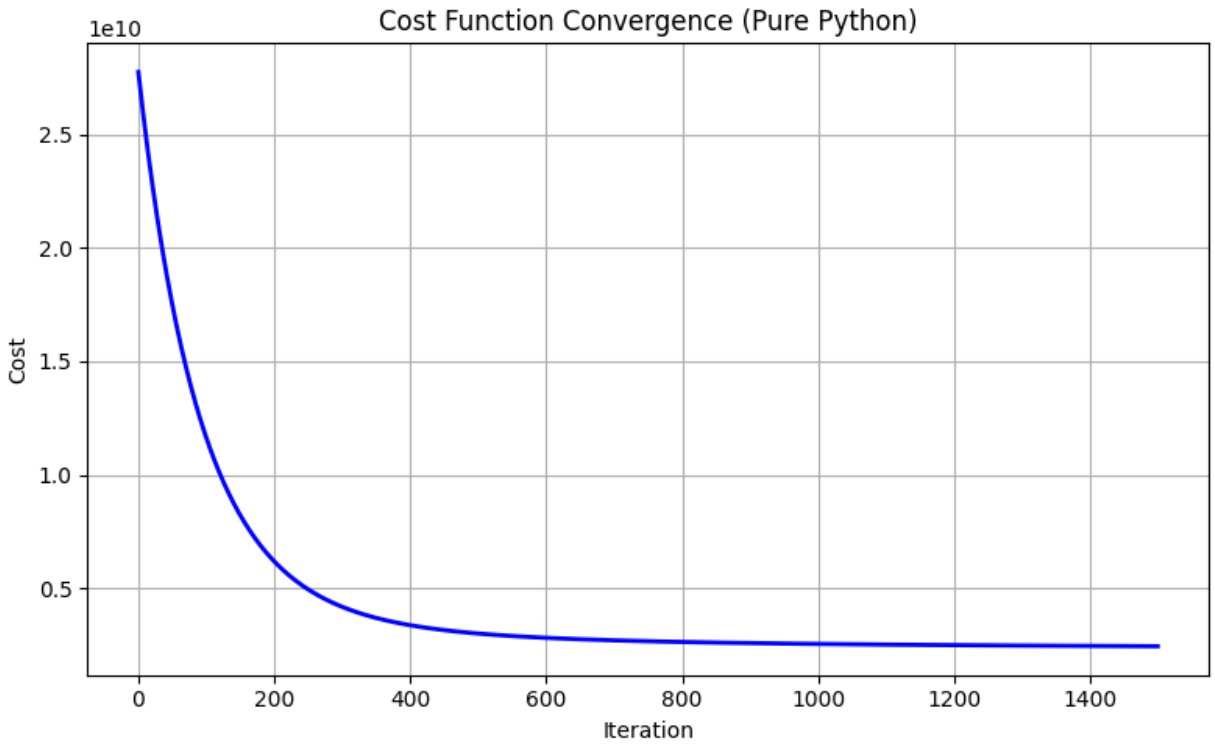

Figure 1: Cost vs iterations

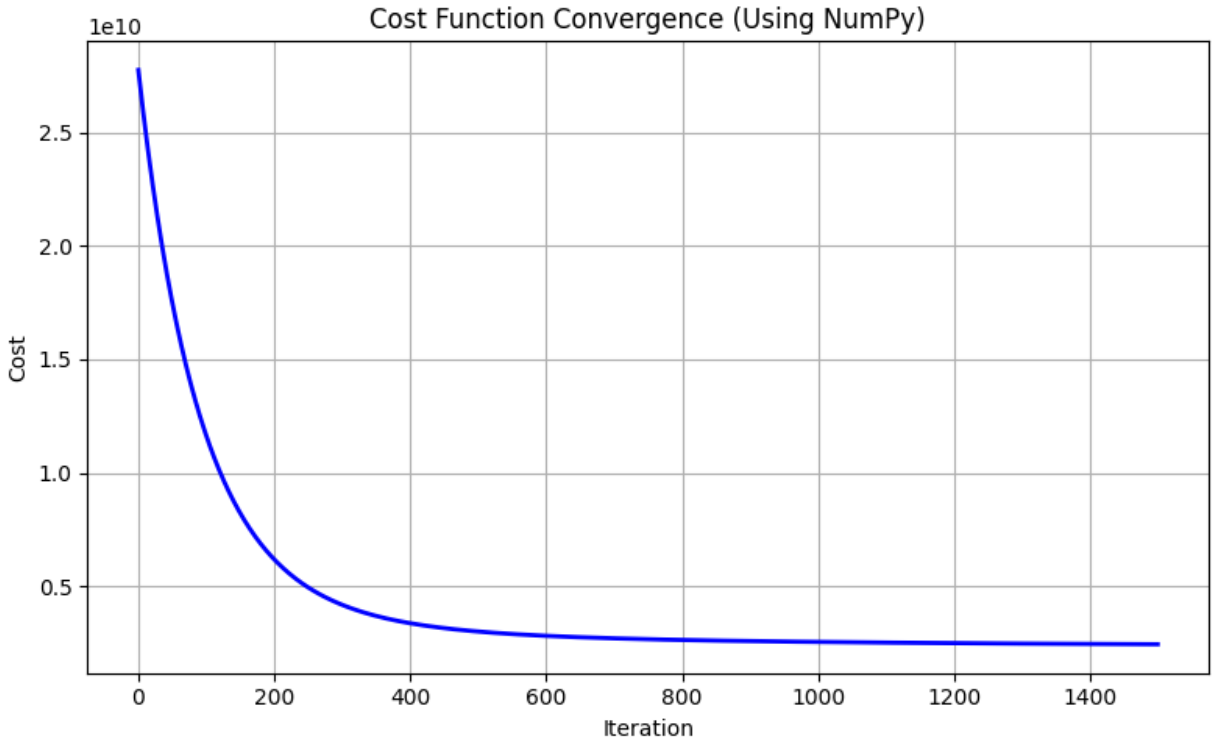

Figure 2: Cost vs iterations

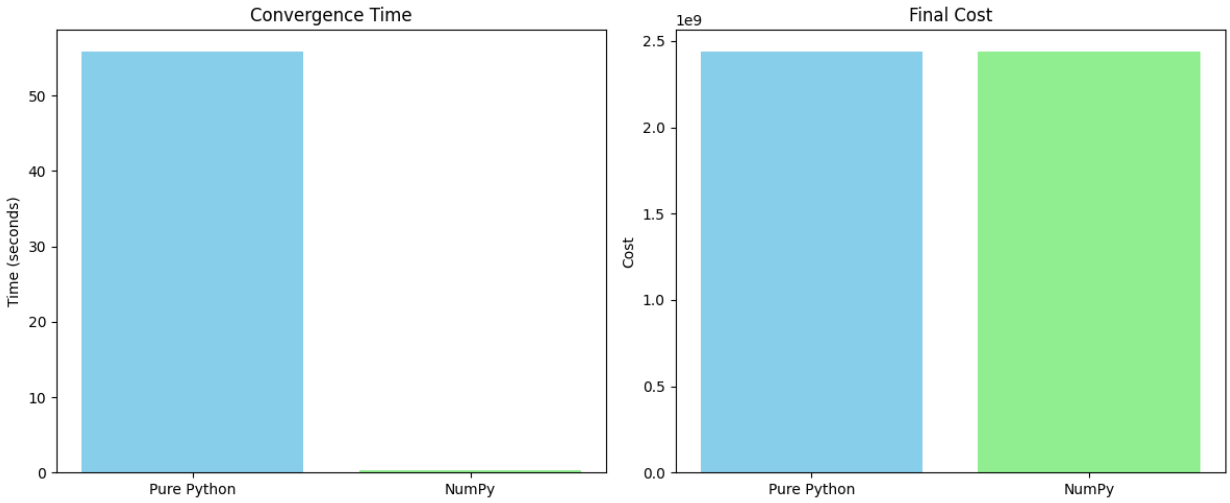

Figure 3: Convergence speed and cost comparison

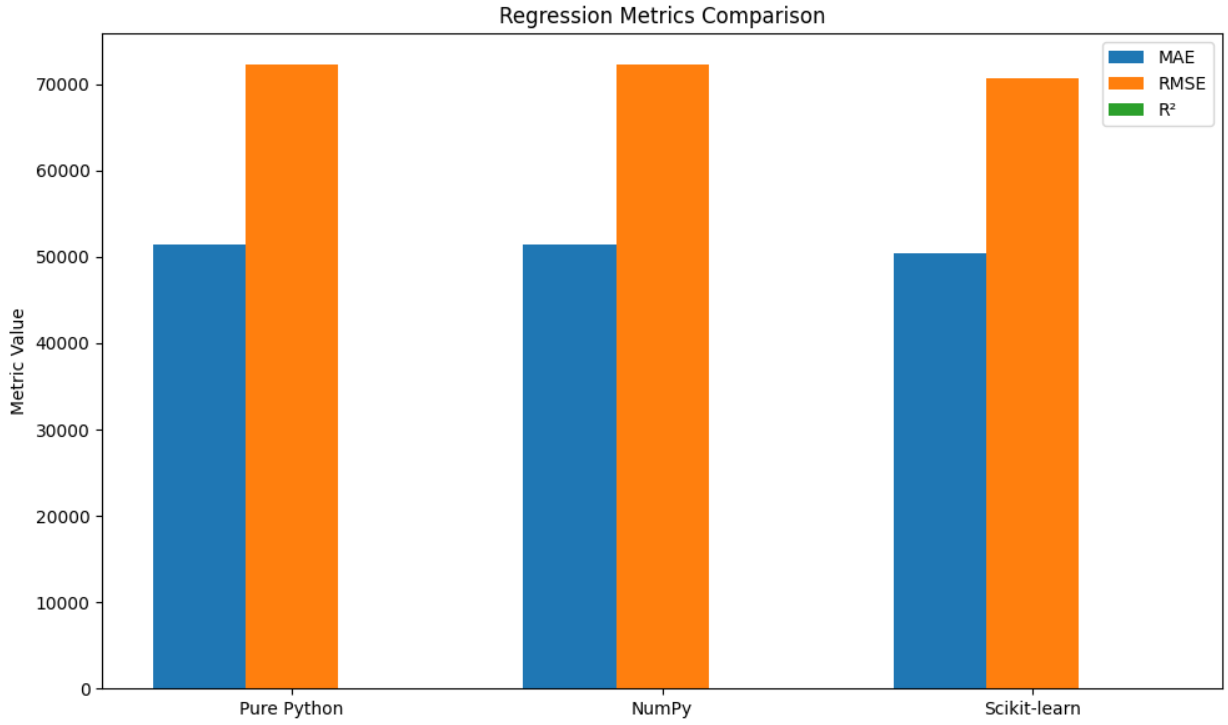

Figure 4: Error Metrics

6 Conclusion

To sum up, we compared different ways of implementing linear regression and found clear differences in how fast they work and how accurately they predict. Each method had its strengths — some were quicker to give results, while others needed more time but were easier to understand.

The changes in performance were mostly due to how the code was written and how well it made use of available tools. We also noticed that things like the choice of starting values and learning rates affected how quickly the models learned.

IMPORTANCE OF INITIAL PARAMETERS - Your initial parameters decide how fast the cost function will converge and may also give different values of minimum cost function in case you are using an another function than squared error. The squared error function cost function is like a bowl and the initial parameters decide where you stand on the bowl - the closer you are to the bottom the faster it will converge but if you are not using squared error cost function then the graph could be anything but a bowl so there could be different minima and hence your initial parameters may get you to any minima.

PRE-PROCESSING OF DATA - I didn't feel any data to be useless as such so first i included all of them, applied one-hot encoding for ocean proximity and then applied Z-scale to all the features so that each feature has the same scaling. After this i started experimenting with different features like dropping a feature or dividing two features to get a new feature and tried to see if it reduces the errors. What worked the best for me i chose to go with at last.

Learning rate - It decides how better your initialised parameters will reach to the final values that at the time of convergence. If too small your convergence time will increase a lot and if too large it will not converge instead of the cost function deacreasing, it will start increasing.

Accuracies - The errors should be low on both training and validation sets and also Both training and validation sets should be closer as it shows the model is neither overfit nor underfit. Although the error seems off the chart i believe this model is not suited for a multivariable linear regression at all instead a polynomial regression would have suited better.

WHY IS NUMPY FASTER THAN PURE PYTHON IMPLEMENTATION -

- Pure Python uses loops (for) to go through data element by element, which is slow.
- NumPy uses vectorized operations, which means it applies operations to entire arrays at once, without loops in Python.
- NumPy is built on top of C and Fortran, which are compiled languages and run much faster than interpreted Python.
- NumPy can take advantage of low level CPU features like SIMD, MULTITHREADING, BLAS/LAPACK libraries (used in scientific computing for fast matrix operations)

WHY IS SCIKIT-LEARN FASTEST -

- Scikit-learn's core has various libraried working under the hood like BLAS (Basic Linear Algebra Subprograms), LAPACK (Linear Algebra Package), Intel MKL / OpenBLAS (hardware-optimized versions of BLAS).

- When you are calling a function like model.fit You're indirectly invoking highly optimized C/Fortran routines for matrix operations, which are much faster than any Python loop or even raw NumPy.
- Algorithms like LinearRegression, LogisticRegression, SVM, etc., are implemented using compiled backends (sometimes even Cython or C++).These backends are heavily optimized for performance — both in speed and memory usage.
- Scikit-learn can take advantage of multi-core CPUs.

I have used LLMs like chatgpt for utilizing a little bit panda (for implementing it's functions for data pre-processing) and matplotlib (for the plottings) and to understand the optimizations happening inside scikit-learn and also to help me writing this pdf using latex on overleaf.