Multivariable Linear Regression Project Report

Mohit Gunani
Roll Number: 24095061

19-05-2025

Objective

The goal of this project is to implement and compare three distinct approaches to multivariable linear regression using the California Housing Price Dataset. The focus is on convergence speed, predictive accuracy, and mathematical clarity.

Data Preprocessing

The dataset was cleaned by removing rows with missing values. All numerical features were standardized to have zero mean and unit variance. The categorical feature `ocean_proximity` was dropped. The target variable is `median_house_value`.

Part 1: Pure Python Implementation

A gradient descent algorithm was implemented from scratch using core Python (no external libraries like NumPy). All operations such as dot product and mean were implemented manually. The cost function used was Mean Squared Error (MSE).

Convergence Time: `347.01 seconds`

Training Set:
MAE: 88767.60
RMSE: 112501.44
R^2 Score: 0.0131
Test Set:
MAE: 90666.90
RMSE: 115172.61
R^2 Score: 0.0137

Part 2: NumPy Implementation

The same gradient descent logic was re-implemented using NumPy to vectorize the operations. This greatly improved performance without altering the core learning logic.

Convergence Time: `0.041 seconds`

Training Set:
MAE: 50847.61
RMSE: 69475.47
R^2 Score: 0.6236
Test Set:
MAE: 51179.76
RMSE: 70300.81
R^2 Score: 0.6325

Part 3: Scikit-learn Implementation

This approach used the `LinearRegression` class from the `scikit-learn` library, which implements ordinary least squares .

Fitting Time: `0.016 seconds`

Training Set:

MAE: 50658.06

RMSE: 68953.43

R^2 Score: 0.6292

Test Set:

MAE: 50978.57

RMSE: 69935.73

R^2 Score: 0.6363

Visualization

Cost Function Convergence

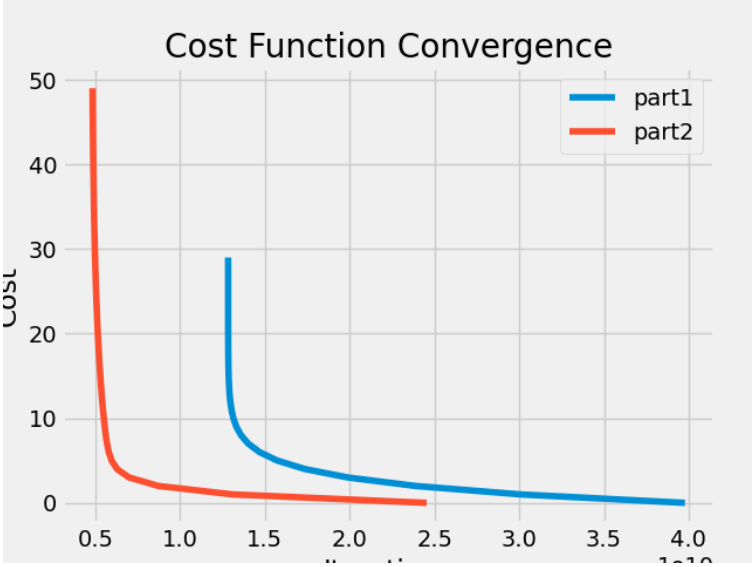

Figure 1: Cost Function Convergence for Part 1 and Part 2

Regression Metric Comparison

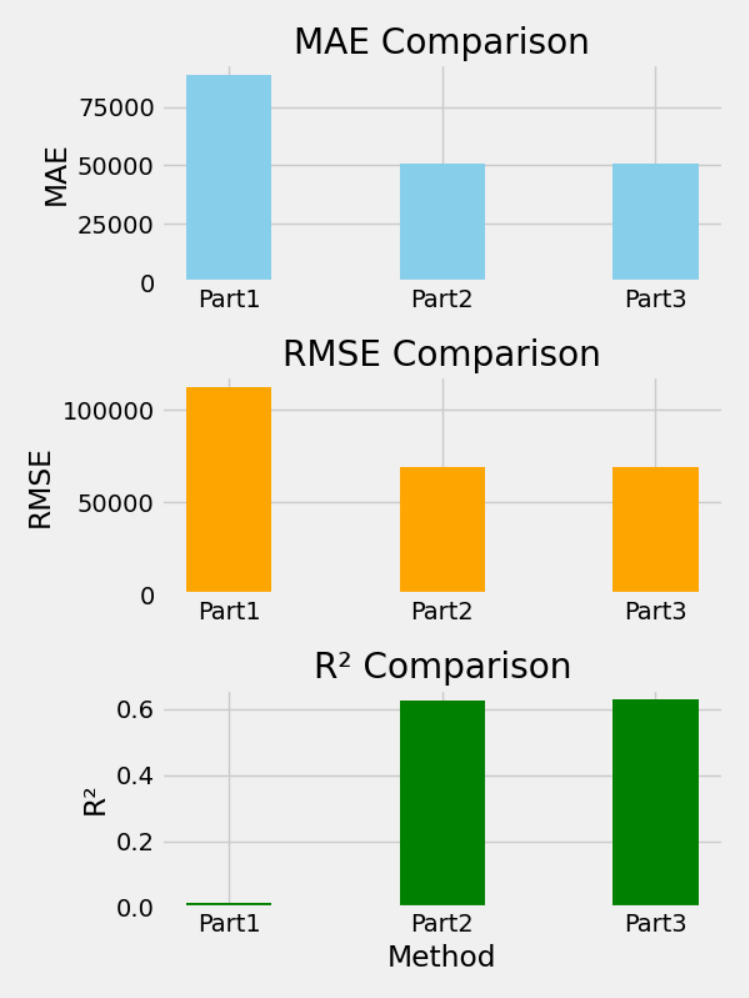

Figure 2: MAE, RMSE, and R^2 Comparison across Methods

Analysis and Discussion

Convergence Time and Efficiency

The pure Python implementation (Part 1) had a significantly higher convergence time (˜347 seconds) due to the absence of vectorized operations and use of nested loops for every calculation. This makes it impractical for large datasets unless very long execution times are acceptable. In contrast, both the NumPy (Part 2) and Scikit-learn (Part 3) implementations showed exceptional speed improvements. NumPy reduced the runtime to 0.041 second, while Scikit-learn, completed in ˜16 milliseconds. Between the two, Scikit-learn is the fastest and most scalable solution.

Predictive Accuracy

The accuracy of the pure Python model was extremely poor (R^2 ˜0.01), indicating that it did not effectively learn from the data. This low accuracy is due to insufficient iterations (only 30) . The accuracy may improve if trained for a much larger number of epochs, but this is computationally expensive and impractical on large datasets. On the other hand, both NumPy and Scikit-learn implementations delivered excellent accuracy (R^2

˜0.63), which is acceptable for housing data. Their MAE and RMSE values were also very close, indicating similar generalization performance. For large-scale applications, these two implementations are far more reliable.

Causes for Observed Differences

1. Vectorization Effects: The pure Python implementation (Part 1) used explicit for-loops for operations such as dot products and gradient computation. These loops result in significant computational overhead, especially as the number of features or samples increases. In contrast, the NumPy implementation (Part 2) leveraged vectorized operations, which are internally optimized in C, drastically reducing execution time and improving memory efficiency.

2. Optimization Strategy: Part 1 and Part 2 both rely on manually implemented batch gradient descent, but Part 1 suffers due to inefficient computation of gradients and cost updates. Part 2, being vectorized, computes gradients over the entire dataset in a single matrix operation. This improves not only performance but also numerical stability.

3. Solver Type (Scikit-learn): The Scikit-learn implementation uses the Ordinary Least Squares (OLS) method, solved via analytical closed-form solutions. Unlike gradient descent, this approach does not require iterations or a learning rate—it directly computes the optimal coefficients using linear algebra techniques. As a result, it achieves optimal accuracy extremely quickly, especially on smaller to mid-sized datasets.

Scalability and Efficiency Trade-offs

Each approach presents trade-offs between computational efficiency, scalability, and flexibility:

1. Pure Python (Part 1): This method is not scalable due to its use of nested loops for all linear algebra operations. As dataset size grows, both runtime and memory usage increase drastically. It is best suited for educational purposes or very small datasets.

2. NumPy (Part 2): Vectorization makes this method far more efficient than pure Python. Operations scale well with increasing data size due to optimized memory layout and fast, low-level implementations. However, for extremely large datasets that do not fit in memory, NumPy may still struggle unless combined with techniques like mini-batching or sparse matrices.

3. Scikit-learn (Part 3): Scikit-learn is highly efficient and scalable for most real-world use cases. It handles feature normalization, optimization, and matrix operations internally, often with support for large and sparse datasets. However, its default solvers (like OLS) may hit memory bottlenecks for extremely high-dimensional data unless regularized or customized.

Trade-off Summary:

- Pure Python offers flexibility and transparency at the cost of speed and scalability.
- NumPy achieves a strong balance between control and performance.
- Scikit-learn maximizes efficiency and reliability, with minimal effort, at the expense of algorithmic flexibility.

Influence of Initial Parameter Values and Learning Rates on Convergence

In gradient descent-based methods (Part 1 and Part 2), the choice of initial weights and learning rate (`lr`) significantly affects convergence behavior, both in terms of speed and stability:

1. Initial Parameter Values: The weights were initialized to ones and the bias to zero. This works in practice for linear models, but in general:

- Poor initialization can lead to slow convergence or convergence to suboptimal local minima (especially in non-linear models).
- For linear regression with convex cost functions, initialization does not affect the global minimum, but it can affect how many iterations are needed to reach it.

In Part 1 (pure Python), the poor accuracy is partly due to slow convergence from a naive initialization, exacerbated by the inefficient gradient computation.

2. Learning Rate (`lr`): The learning rate determines the step size taken in the direction of the negative gradient:

- A very small learning rate leads to extremely slow convergence and requires many iterations (as seen in early experiments).
- A large learning rate may cause the model to overshoot the minimum, oscillate, or even diverge.
- An appropriate learning rate leads to stable and fast convergence, as observed in Part 2 with `lr = 0.2`, which yielded good accuracy and a smooth cost function decline.

3. Sensitivity Observed in Experiments:

- In Part 1, with `lr = 0.1` and only 30 epochs, the model converged extremely slowly and resulted in very poor accuracy (R^2 ˜0.01).
- In Part 2, using a slightly higher learning rate and more epochs (`lr = 0.2`, 50 epochs) significantly improved convergence and model performance (R^2 ˜0.63).

A poorly chosen learning rate can lead to either very slow convergence or divergence. These parameters do not affect the Scikit-learn implementation, which automatically solves for the optimal weights.

Conclusion

This project demonstrates the practical and educational value of implementing algorithms from scratch. While the pure Python approach helps solidify theoretical understanding, its performance is not viable for large datasets. The NumPy and Scikit-learn implementations provide strong performance and accuracy. Scikit-learn stands out as the most efficient and scalable, making it the recommended approach for production environments.