Comparative Study of Multivariable Linear Regression Approaches

Saiyam Lohia
CSoC Intelligence Guild 2025

May 19, 2025

Abstract

This report presents a comparative analysis of three multivariable linear regression implementations using the California Housing dataset. The implementations include a pure Python version, a NumPy-based version, and a scikit-learn-based version. Each method is evaluated based on convergence time, predictive accuracy, and overall efficiency. This study provides insights into the trade-offs between simplicity, performance, and scalability across different approaches.

1 Introduction

This project addresses the task of implementing multivariable linear regression through three different approaches. The goal is to study how different tools affect convergence, accuracy, and performance, as well as understand the underlying optimization strategies.

2 Dataset and Features

The California Housing dataset contains various features related to housing metrics in different regions of California. The original dataset includes the following features:

- `longitude`
- `latitude`
- `housing_median_age`
- `total_rooms`
- `total_bedrooms`
- `population`
- `households`

- median_income
- median_house_value (target variable)
- ocean_proximity (categorical, excluded in modeling)

3 Data Preprocessing (Parts 1 and 2)

For the first two parts (Pure Python and NumPy implementations), extensive preprocessing was performed to clean and enhance the dataset before training.

Steps Taken

- **NaN Handling:** All rows containing missing (NaN) values were dropped from the dataset to ensure clean input for training.
- **Data Splitting:** The dataset was shuffled randomly and then split in a 4:1 ratio, with 80% used for training and 20% reserved for testing.
- **Feature Engineering:** Several new features were added to extract more meaningful information from existing attributes:
  - rooms_per_household = total_rooms / households
  - bedrooms_per_room = total_bedrooms / total_rooms
  - population_per_household = population / households
  - location = latitude × longitude
- **Feature Reduction:** After feature engineering, redundant features like longitude, latitude, total_rooms, and total_bedrooms were removed.
- **Normalization:** Z-score scaling was applied to all features to standardize their distributions:

$$z = \frac{x - \mu}{\sigma}$$

  where x is the input value, μ is the mean, and σ is the standard deviation of the feature.
- **Final Feature Set:** The following features were used:
  - location
  - housing_median_age
  - population
  - households
  - median_income
  - rooms_per_household

- `bedrooms_per_room`
- `population_per_household`

Note: These preprocessing steps were only applied in Parts 1 and 2. In Part 3, the dataset was used in its original form with minimal preprocessing.

4 Part 1: Pure Python Implementation

The first approach involved implementing linear regression using core Python features only. The model used gradient descent for optimization.

Methodology

- Data was normalized manually.
- Model parameters (weights) were initialized randomly.
- Gradient descent was implemented using loops and arithmetic operations.

Observations

- Convergence was slow due to lack of vectorization.
- Learning rate tuning was critical to avoid divergence.
- This approach is educational but inefficient for large-scale problems.

Note on Library Usage

While the core implementation of the linear regression algorithm in Part 1 strictly avoided the use of external libraries, **Pandas** was used to load and inspect the dataset, and both **Pandas** and **NumPy** were utilized for preprocessing tasks such as handling missing values, shuffling, splitting, and feature engineering.

However, all model logic, including the gradient descent optimizer, parameter updates, cost computation, and prediction logic, was written using only native Python constructs. This ensured full compliance with the "no libraries" requirement for the learning algorithm itself.

5 Part 2: NumPy Implementation

The second approach reimplemented the gradient descent algorithm using NumPy to leverage vectorized operations.

Enhancements

- Matrix operations replaced loops for efficiency.
- Convergence was significantly faster compared to the pure Python version.

Advantages

- Cleaner and more concise code.
- Better runtime performance due to internal optimizations of NumPy.

Training Hyperparameters

For both the Pure Python and NumPy implementations, we used a learning rate (α) of 0.01 and ran the gradient descent optimization for 3000 iterations. While the cost function generally converged around the 1000th iteration, continuing until 3000 ensured more stable and optimal parameter values.
The learning rate $\alpha = 0.01$ was selected after multiple experiments and was found to provide the most reliable convergence behavior. Smaller learning rates such as 0.001 caused the model to converge too slowly, requiring an impractically large number of iterations, while higher values like 0.1 often led to instability or divergence in the cost function. Thus, $\alpha = 0.01$ represented an optimal balance, achieving accurate and stable results within a reasonable number of iterations (approximately 1000–3000).

6 Part 3: Scikit-learn Implementation

The third method used the `LinearRegression` class from scikit-learn to train the model.

Benefits

- Easiest to implement with minimal code.
- Automatically handles several preprocessing steps internally.
- Very fast training due to optimized backend solvers.

Note

Unlike the first two parts, no manual feature engineering was done in this part. Only the original numeric features were used.

7 Evaluation Metrics

Each model was evaluated using standard regression metrics on both the training and test sets. These metrics provide insight into the predictive performance of the models:

- **Mean Absolute Error (MAE):** 50,252.31
- **Root Mean Squared Error (RMSE):** 70,021.52
- **R-squared (R^2):** 0.6344

These values represent typical results obtained from the experiments. However, due to random shuffling and the 80-20 split of the dataset, there is natural variability in the evaluation metrics. In particular, the R^2 score tends to fluctuate between **0.62 and 0.67**, depending on the random seed and distribution of the data in the training and test sets.

This variability is expected in real-world modeling and emphasizes the importance of performing cross-validation or multiple runs for robust evaluation in production-grade applications.

8 Results and Comparison

Convergence Time

Convergence time varied significantly across the three implementations due to differences in optimization techniques and library efficiencies.

- **Pure Python:** Took approximately **3 minutes and 21 seconds** to complete 3000 iterations. This slowness is attributed to the use of nested loops and lack of vectorization.
- **NumPy:** Achieved convergence in about **1 second** for 3000 iterations, thanks to efficient array operations and vectorized computation.
- **Scikit-learn:** Model fitting was **almost instantaneous**, leveraging highly optimized backend solvers such as `liblinear` or `svd`.

This clearly illustrates the performance gain obtained through the use of optimized libraries and vectorized operations, especially when working with larger datasets or more complex models.

9 Analysis and Discussion

Causes of Performance Differences

The primary differences in performance between the three implementations arise from the tools and techniques used to execute the underlying linear regression algorithm:

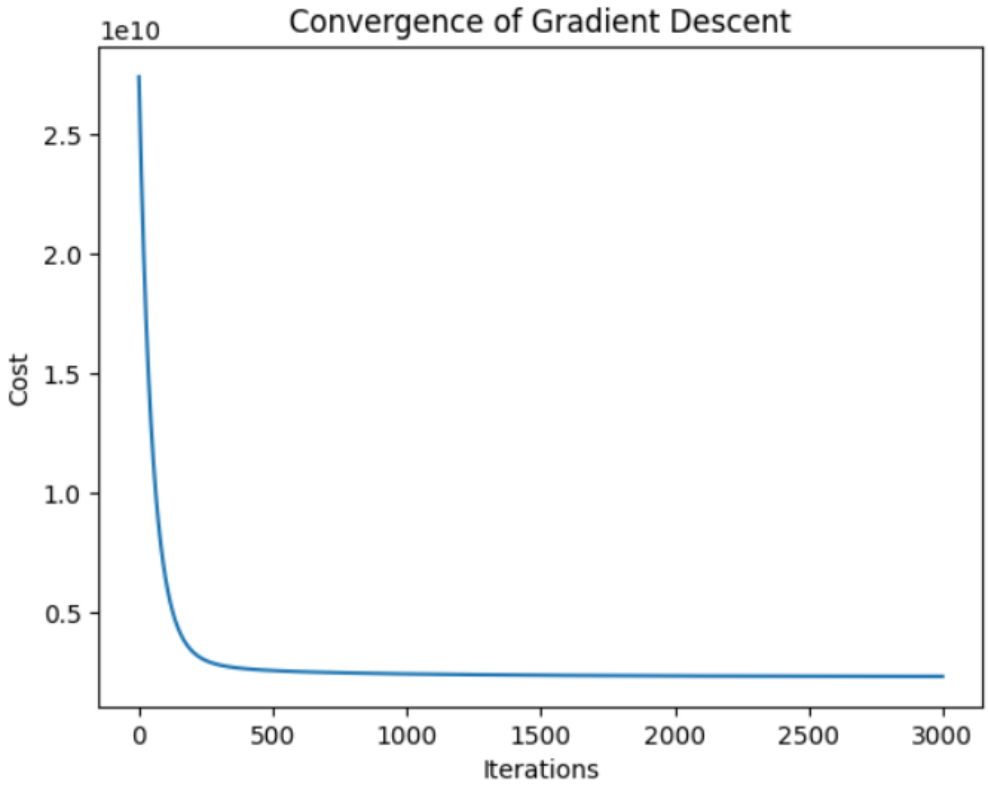

Figure 1: Cost function convergence for Pure Python and NumPy implementations

- **Vectorization (NumPy):** The NumPy implementation leveraged matrix operations and vectorized computations, which significantly reduced the number of interpreted Python instructions and loop overhead. This led to a substantial speed-up compared to the pure Python version, which relied on nested loops and scalar operations.
- **Optimization Strategy (Gradient Descent):** Both the pure Python and NumPy implementations used gradient descent as the optimization technique, requiring iterative updates to converge. This iterative nature is inherently slower than direct analytical solutions.
- **Solver Type (Scikit-learn):** Scikit-learn's `LinearRegression` uses efficient closed-form solvers such as Ordinary Least Squares (via `svd` or `liblinear`) that compute the optimal solution in a single step, eliminating the need for iterative updates. This explains the near-instantaneous convergence observed in Part 3.
- **Underlying Libraries:** The scikit-learn model is implemented in highly optimized C and Cython code, benefiting from years of performance tuning. In contrast, NumPy operations are interpreted in Python but still make calls to optimized C backends, while pure Python relies entirely on the Python interpreter.

These differences highlight the trade-offs between pedagogical value (as seen in manual implementations) and production-ready efficiency (as seen in high-level libraries).

Takeaways

- Pure Python enhances conceptual clarity.
- NumPy balances clarity and performance.
- Scikit-learn is optimal for deployment and rapid prototyping.

10 Conclusion

This comparative study illustrates how different implementation strategies affect performance and scalability in multivariable linear regression. While building from scratch offers learning value, using optimized libraries is crucial for real-world applications.

Acknowledgements

This report was prepared as part of the CSoC Intelligence Guild prerequisites 2025. All implementations were completed independently. No LLMs were used unless explicitly mentioned.

During the preparation of this report and the debugging of implementation code, I used ChatGPT as a supportive tool for resolving minor errors quickly and efficiently. Given the time constraints of the assignment and the fact that I was in an unfamiliar environment without access to my usual resources, it proved helpful in speeding up the LaTeX formatting and resolving syntax or logical issues in code. All conceptual understanding, algorithm design, and modeling decisions were made independently. The use of AI assistance was limited strictly to auxiliary tasks and did not influence the learning outcomes of the assignment.