Comparative Study of Multivariable Linear Regression Implementations

Samarpan Verma

24124041

1 Introduction

This study implements and compares three distinct approaches to multivariable linear regression using the California Housing Price Dataset. The implementations include a pure Python approach, an optimized NumPy implementation, and a scikit-learn solution. All implementations are evaluated based on convergence speed and predictive accuracy.

2 Exploratory Data Analysis

2.1 Dataset Overview

The California Housing Price Dataset contains information collected from the 1990 California census. Initial exploration revealed a comprehensive dataset with features describing housing districts across California, including geographic location, housing age, room counts, population, and median house values.

2.2 Data Cleaning and Preprocessing

2.2.1 Handling Missing Values

Initial analysis revealed 207 null values in the `total_bedrooms` column, representing approximately 1% of the dataset. Since the proportion was small, median imputation was selected over dropping records.

2.2.2 Distribution Analysis

Histogram analysis of all features revealed several highly skewed distributions, particularly in `total_rooms`, `total_bedrooms`, `population`, and `households`:

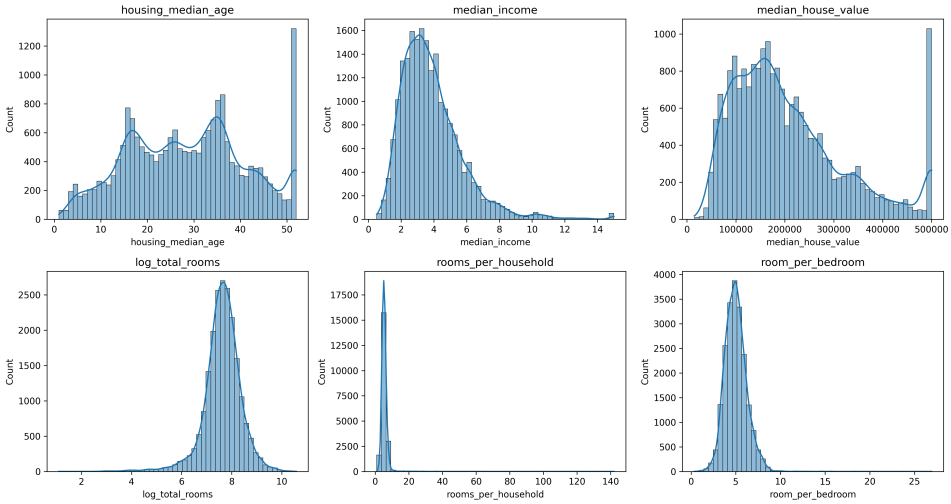

Figure 1: Distribution of Original Features

2.3 Feature Engineering

2.3.1 Log Transformation

To address skewness, log transformation (specifically `log1p`) was applied to `total_rooms`, `total_bedrooms`, `population`, and `households`. This transformation improved correlation with the target variable `median_house_value`:

Table 1: Feature Correlations with Median House Value After Log Transformation

Feature	Correlation
median_house_value	1.000000
median_income	0.688075
log_total_rooms	0.159422
total_rooms	0.134153
housing_median_age	0.105623
log_households	0.073612
households	0.065843
log_total_bedrooms	0.053059
total_bedrooms	0.049457
log_population	-0.021205
population	-0.024650
longitude	-0.045967
latitude	-0.144160

2.3.2 Custom Feature Creation

Several ratio-based features were derived to capture interactions between original features:

- `rooms_per_household`: Ratio of total rooms to households
- `room_per_bedroom`: Ratio of total rooms to bedrooms

- `population_per_household`: Ratio of population to households

These derived features showed improved correlation with the target variable:

Table 2: Feature Correlations with Median House Value After Custom Feature Creation

Feature	**Correlation**
median_house_value	1.000000
median_income	0.688075
room_per_bedroom	0.367217
log_total_rooms	0.159422
rooms_per_household	0.151948
total_rooms	0.134153
housing_median_age	0.105623
log_households	0.073612
households	0.065843
log_total_bedrooms	0.053059
total_bedrooms	0.049457
log_population	-0.021205
population_per_household	-0.023737
population	-0.024650
longitude	-0.045967
latitude	-0.144160

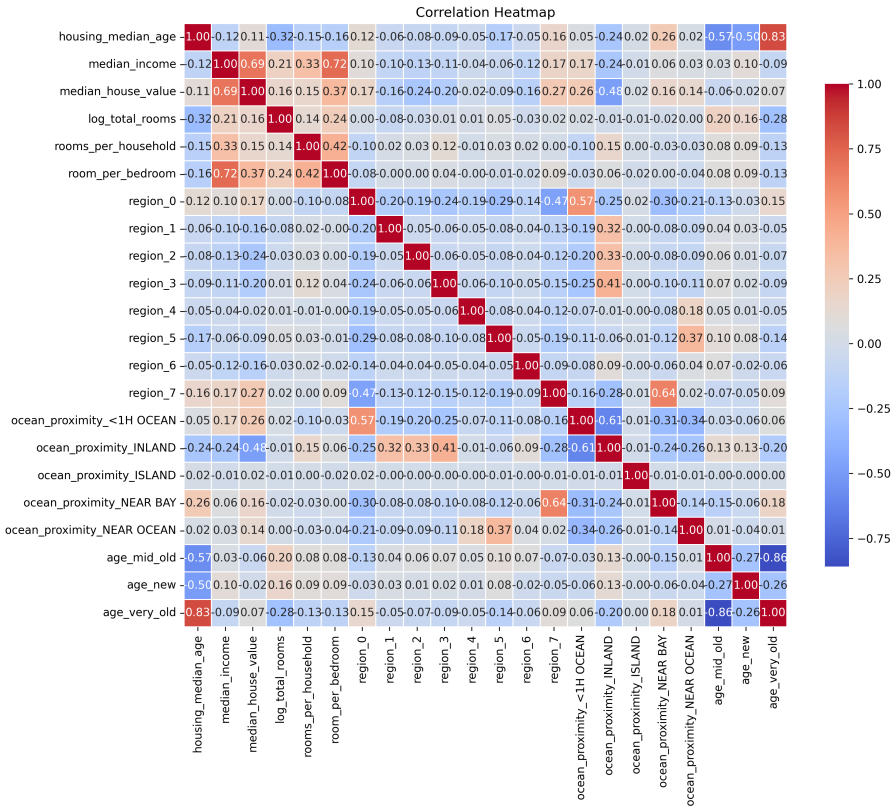

Figure 2: Correlation Heatmap of Features

2.3.3 Geographical Feature Engineering

Analysis of heatmaps revealed that raw latitude and longitude values had limited predictive power. Instead, a K-means clustering approach was used to divide California into distinct regions:

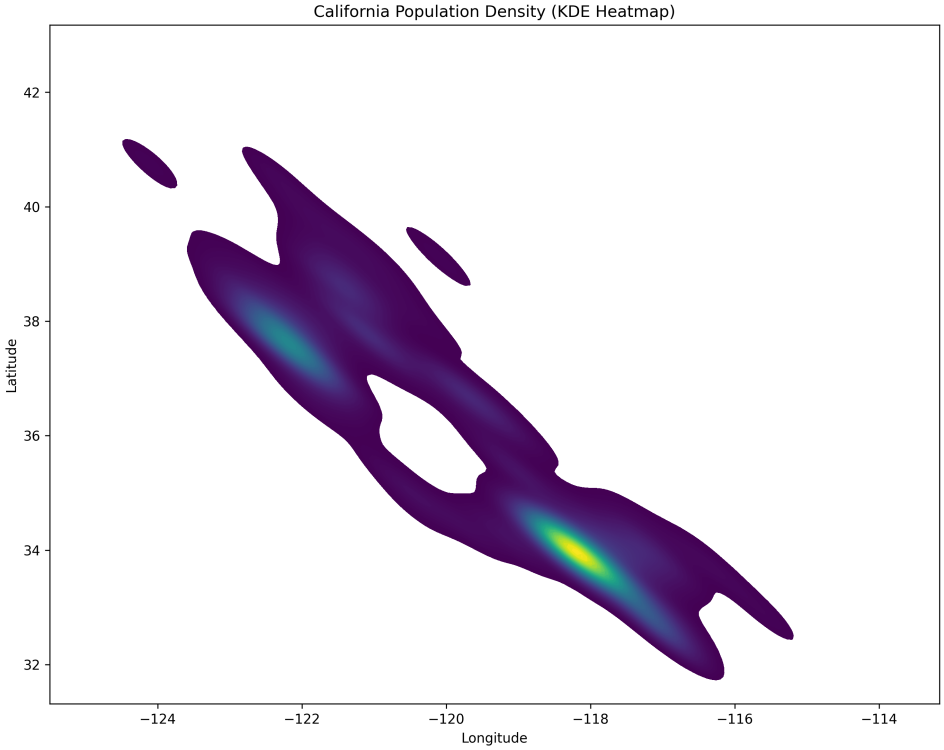

Figure 3: Population Density Across California

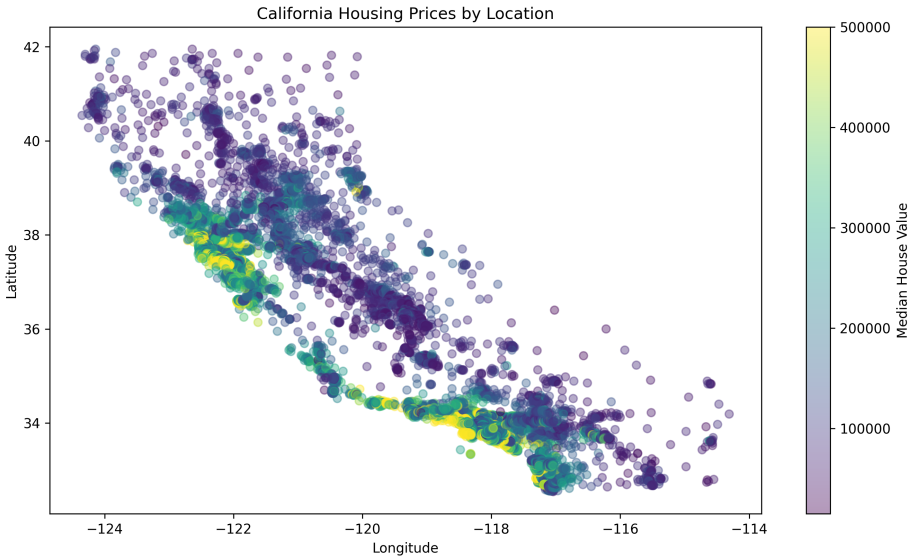

Figure 4: Housing Prices by Location

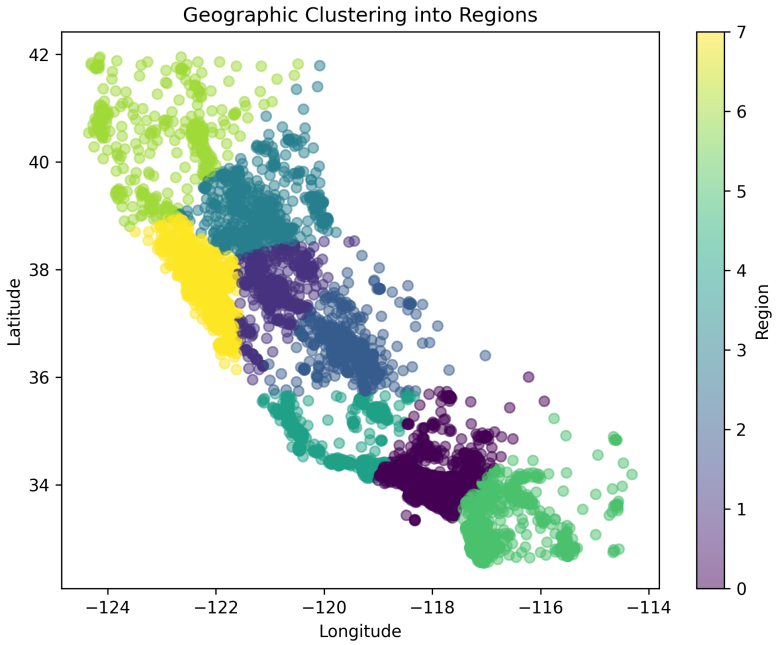

Figure 5: Geographical Clustering into Regions

2.3.4 Categorical Feature Encoding

One-hot encoding was applied to:

- Geographical regions (from K-means clustering)
- `ocean_proximity` feature
- Housing age categories (`new`, `mid_old`, `very_old`) derived from `housing_median_age`

2.4 Feature Selection

Based on correlation analysis and feature importance assessment, the following features were dropped from the final dataset:

```
cols_to_drop = [
    'total_rooms',
    'log_households',
    'households',
    'log_total_bedrooms',
    'total_bedrooms',
    'log_population',
    'population_per_household',
    'population',
    'longitude',
    'latitude',
]
```

The final prepared dataset was exported as `california_prepared.csv` for use in the regression implementations.

3 Implementation Approaches

3.1 Pure Python Implementation

The first implementation relies solely on core Python features without external libraries. The gradient descent optimization is implemented using nested loops and standard mathematical operations.

3.1.1 Mathematical Foundation

The hypothesis function for multivariate linear regression is given by:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n = \theta^T x \tag{1}$$

The cost function is defined as:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 \tag{2}$$

The gradient descent update rule with learning rate $\alpha = 0.5$ is:

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \tag{3}$$

3.1.2 Early Convergence Criteria

To optimize computation time, an early convergence criterion was implemented with a threshold of 1×10^{-6} between consecutive cost function values:

$$|J(\theta)_t - J(\theta)_{t-1}| < 1 \times 10^{-6} \tag{4}$$

Where $J(\theta)_t$ represents the cost at iteration t, and $J(\theta)_{t-1}$ represents the cost at the previous iteration.

3.2 Optimized NumPy Implementation

The second implementation leverages NumPy's vectorized operations to enhance computational efficiency while maintaining the same mathematical foundation as the pure Python implementation.

3.2.1 Vectorization Approach

Using NumPy's vectorization, the gradient descent update can be rewritten as:

$$\theta := \theta - \alpha \frac{1}{m} X^T (X\theta - y) \tag{5}$$

Where X is the matrix of input features, y is the vector of target values, and θ is the parameter vector.

3.2.2 Early Convergence Check

The same early convergence criterion of 1×10^{-6} between consecutive cost values was implemented in a vectorized manner:

3.2.3 Implementation Details

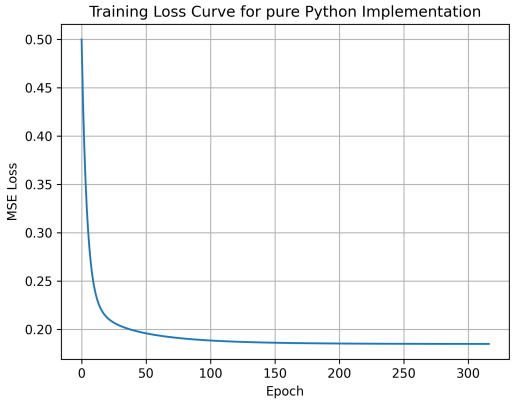

Figure 6: Pure Python Implementation

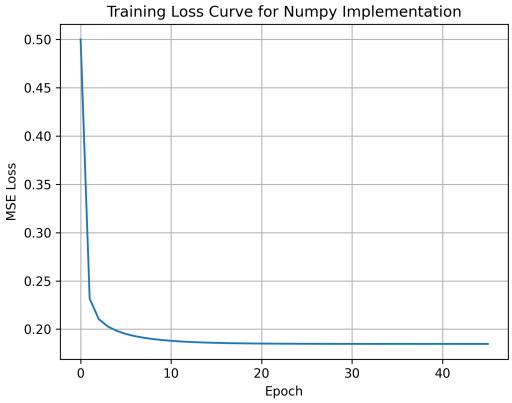

Figure 7: NumPy Implementation

Figure 7: Convergence Speed Comparison Between Pure Python and NumPy Implementations

3.3 Scikit-learn Implementation

The third implementation utilizes scikit-learn's LinearRegression class, which employs Ordinary Least Squares (OLS) for parameter estimation.

3.3.1 Ordinary Least Squares

Scikit-learn's LinearRegression uses a closed-form solution to find the optimal parameters:

$$\theta = (X^T X)^{-1} X^T y \tag{6}$$

This approach directly minimizes the sum of squared residuals without iterative optimization, providing an exact solution when $X^T X$ is invertible.

4 Performance Comparison

This section presents a comprehensive comparison of all three implementations based on convergence time, prediction accuracy, and computational efficiency.

4.1 Early Convergence Results

Table 3: Iterations to Convergence with 1×10^{-6} Threshold

Implementation	Iterations	Final Cost
Pure Python	316	0.1848189
NumPy	45	0.1847672

4.2 Performance Metrics

Table 4: Performance Metrics Across Implementations

Metric	Pure Python	NumPy	Scikit-learn
Training Time (s)	59.6650	0.4704	0.0175
MAE (Training)	51054.74	51028.82	50826.23
MAE (Testing)	51862.51	51816.81	52351.19
RMSE (Training)	70060.94	70051.17	69766.31
RMSE (Testing)	71510.45	71474.10	73292.02
R^2 (Training)	0.6304	0.6305	0.6359
R^2 (Testing)	0.6201	0.6205	0.5901

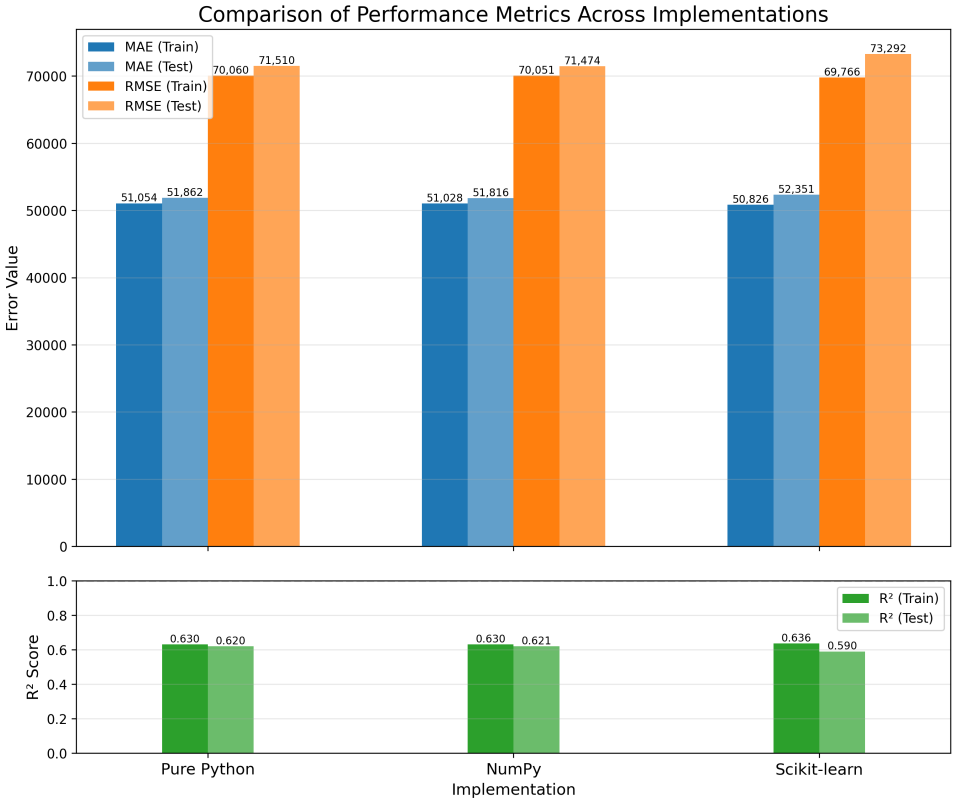

Figure 8: Comparison of Performance Metrics Across Implementations

4.3 Execution Time Comparison

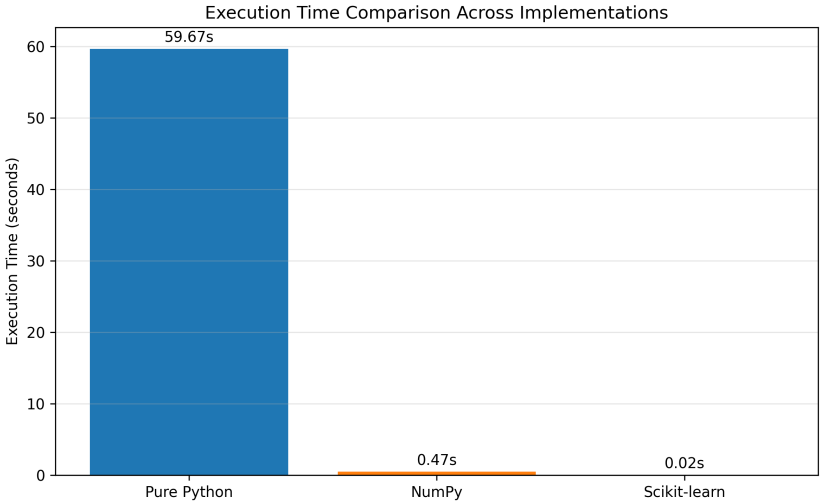

Figure 9: Execution Time Comparison

5 Discussion

5.1 Impact of Feature Engineering

The data preprocessing and feature engineering steps significantly improved the predictive power of all implementations. Particularly noteworthy was the creation of ratio-based features and the application of K-means clustering for geographical regions.

5.2 Impact of Vectorization

The NumPy implementation demonstrates significant performance improvements compared to the pure Python approach due to vectorized operations. This highlights the importance of efficient array operations for numerical computing. The execution time decreased from approximately 60 seconds in the pure Python implementation to just 0.47 seconds with NumPy-a 127× speedup-while achieving a similar final cost value (0.1848189 vs. 0.1847672).

5.3 Early Convergence Benefits

Implementing the early convergence criterion with a threshold of 1×10^{-6} resulted in substantial computational savings without sacrificing accuracy. The pure Python implementation required 316 iterations to converge, while the NumPy implementation reached convergence in just 45 iterations-a 7× reduction in the number of iterations needed.

5.4 Convergence Stability with Fixed Learning Rate

Both custom implementations used a fixed learning rate of $\alpha = 0.5$. This choice affects:

- Convergence speed: larger values of α generally lead to faster convergence
- Stability: too large values may cause overshooting or divergence

5.5 Trade-offs Between Approaches

The scikit-learn implementation provides a computationally efficient closed-form solution but offers less transparency into the optimization process. With a training time of just 0.0175 seconds, it was 27× faster than the NumPy implementation and 3,410× faster than the pure Python implementation. While scikit-learn achieved the highest training R^2 score (0.6359), it demonstrated a larger generalization gap with a testing R^2 of 0.5901 compared to the custom implementations (0.6201 and 0.6205). This difference in train-test performance may warrant further investigation into the model's generalization characteristics, or might be just a result of random distribution.

6 Conclusion

This study demonstrates the trade-offs between different implementation approaches for multivariable linear regression. While the pure Python implementation offers educational clarity, the NumPy implementation provides substantial performance improvements through vectorization. The scikit-learn solution balances ease of use with computational efficiency through its matrix-based closed-form solution.

The early convergence criterion of 1×10^{-6} proved effective in reducing computation time while maintaining prediction accuracy. Additionally, the feature engineering process, particularly the creation of custom ratio features and geographical clustering, significantly enhanced the predictive power of all implementations.

In terms of practical application, the NumPy implementation offers the best balance between performance and generalization, with a 127× speedup over pure Python while maintaining comparable prediction accuracy. The scikit-learn implementation provides exceptional computational efficiency but shows a somewhat larger difference between training and testing performance compared to the custom implementations.