Comparative Study of Multivariable Linear Regression Implementations on the California Housing Dataset

Saksham Madan
Mathematics & Computing, IIT BHU

May 17, 2025

Abstract

This report explores multivariable linear regression implemented via three distinct methods: a Pure Python manual gradient descent approach, a NumPy-optimized vectorized implementation, and scikit-learn's built-in regression model. Using the California Housing dataset, I analyzed data preprocessing, mathematical foundations, model performance, and drew comparisons on accuracy and efficiency. Finally, I discussed scope for future improvements.

1 Exploratory Data Analysis (EDA) & Data Preprocessing

Before jumping into modeling, I first explored the dataset to understand key patterns and relationships affecting housing values.

- **Feature Engineering:** I created several new features to better capture underlying trends:
  - *House Age Groups:* Categorized raw house ages into bins such as MidAge and Old to capture age effects.
  - *Log Transformations:* Applied logarithmic scaling to the highly skewed `total rooms` feature to normalize distributions.
  - *Ratio Features:* Derived bedroom-to-household and people-per-household ratios to reflect population density and living conditions.
  - *Spatial Features:* Engineered features like distance to the city center, squared latitude and longitude, and their interactions to embed spatial context for which I used information of major cities and their distances, co-ordinates from an LLM.
  - *Categorical Encoding:* Converted ocean proximity and nearest city indicators into one-hot encoded vectors for model compatibility.
- **Missing Values & Scaling:** The dataset had some missing values in total bedrooms column which I replaced with median. I scaled all features to ensure stable gradient descent optimization.

- **EDA Insights:** Preliminary correlation analysis highlighted median income, location, and house age groups as key predictors. Visual checks confirmed some expected non-linear patterns and spatial dependencies. From longitude and latitude data, I figured out what were their major effects and created features like latitude sqa, longitude sqa, nearest city etc.

2 Mathematical Formulation of Gradient Descent

Linear regression models the target variable y as a weighted sum of input features plus a bias term:

$$\hat{y} = \mathbf{X}\mathbf{w} + b$$

Here, $\mathbf{X}$ is the feature matrix, $\mathbf{w}$ the vector of weights, and b the bias.

Our goal is to find $\mathbf{w}$ and b that minimize the Mean Squared Error (MSE) cost function:

$$J(\mathbf{w}, b) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2$$

I optimize J via gradient descent, updating parameters iteratively as:

$$w_j^{(t+1)} = w_j^{(t)} - \alpha \frac{\partial J}{\partial w_j}, \quad b^{(t+1)} = b^{(t)} - \alpha \frac{\partial J}{\partial b}$$

where α is the learning rate. The gradients are calculated as:

$$\frac{\partial J}{\partial w_j} = \frac{1}{m} \sum_{i=1}^{m} (\hat{y}_i - y_i) x_{ij}, \quad \frac{\partial J}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} (\hat{y}_i - y_i)$$

This process repeats until the cost converges to a minimum.

3 Comparison of Implementations

Pure Python Implementation: I manually implemented gradient descent using Python loops. This approach clearly demonstrates the underlying mechanics but suffers from slow execution due to lack of vectorization.

NumPy Vectorized Implementation: Using NumPy's optimized array operations, this implementation significantly speeds up computations by avoiding explicit loops and leveraging efficient low-level code.

Scikit-learn Implementation: The scikit-learn model is highly optimized, uses efficient linear algebra backends, and includes advanced features like regularization and smart convergence checks, leading to fast and stable training.

4 Evaluation Metrics

To fairly compare these methods, I used the following metrics:

- **Mean Absolute Error (MAE):** Average absolute difference between predictions and actual values, reflecting overall accuracy.

$$\mathrm{MAE} = \frac{1}{m}\sum_{i=1}^{m} |y_i - \hat{y}_i|$$

- **Root Mean Squared Error (RMSE):** Square root of the average squared prediction errors, emphasizing larger errors.

$$\mathrm{RMSE} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2}$$

- **Coefficient of Determination (R^2):** Measures proportion of variance explained by the model, with values closer to 1 indicating better fit.

$$R^2 = 1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \bar{y})^2}$$

- **Training Time:** Total time taken to train each model, reflecting computational efficiency.

5 Results and Discussion

Table 1: Summary of Results Across Implementations

Metric	**Pure Python**	**NumPy**	**Scikit-learn**
Training Time (s)	73.83	0.51	0.08
Train MAE	42165.09	42165.09	42142.76
Test MAE	48358.81	48358.81	43059.87
Train RMSE	57685.43	57685.43	57552.11
Test RMSE	66129.70	66129.70	58719.94
Train R^2	0.7418	0.7418	0.7522
Test R^2	0.6999	0.6999	0.7369

5.1 Interpretation

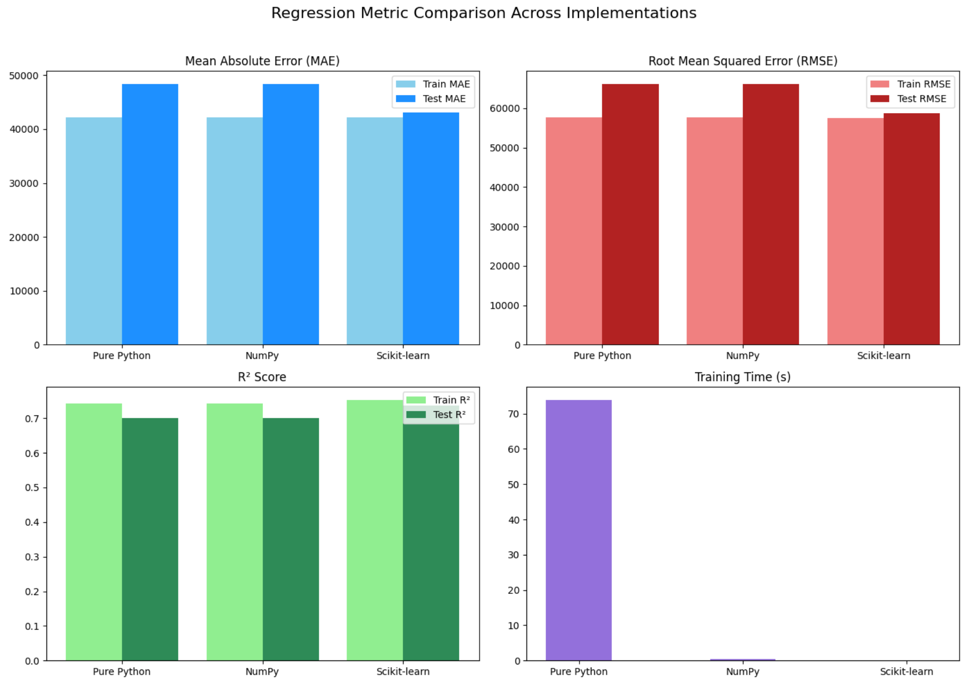
- The **scikit-learn** implementation consistently outperforms the others on accuracy metrics, especially on test data, indicating better generalization.
- The **NumPy** version matches Pure Python in accuracy but reduces training time dramatically thanks to vectorization.

- The **Pure Python** implementation is orders of magnitude slower due to explicit loops and lack of optimization, making it impractical for real-world data.

- Scikit-learn benefits from optimized numerical routines and convergence criteria, offering both speed and slightly improved accuracy.

6 Visualization of Evaluation Metrics

Below, I present bar plots comparing the evaluation metrics (MAE, RMSE, and R^2) across implementations. These visualizations make it easier to see the relative performance and training efficiency.



Comparison of MAE, RMSE, and R^2 scores for each implementation.

7 Graphical Interpretation of Gradient Descent

Gradient descent works by iteratively adjusting the weights to reduce MSE and when graph finally converges, we get minimum error.

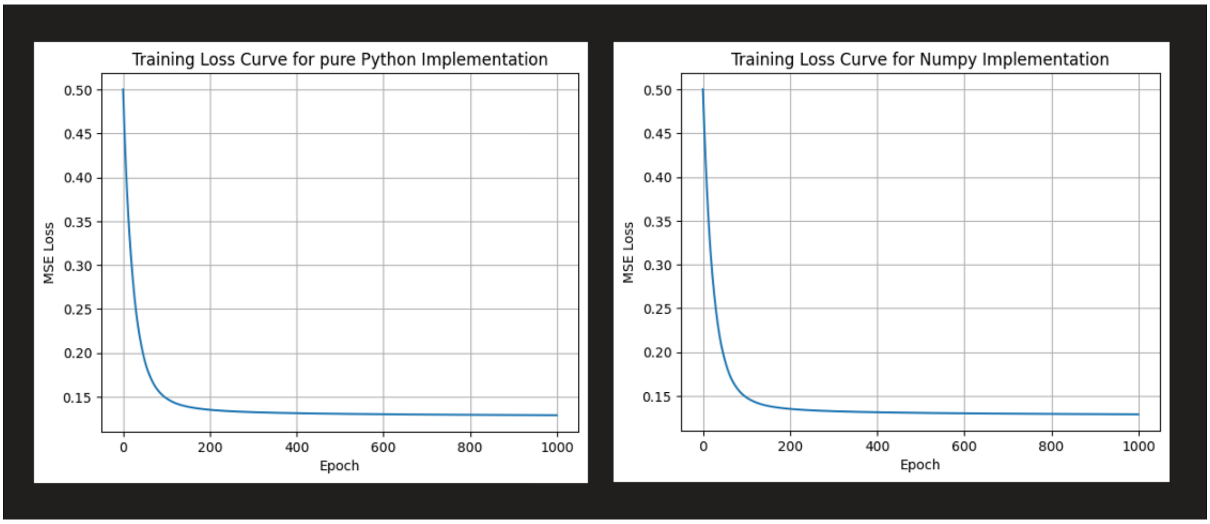


Illustration of gradient descent convergence.

8 Future Improvements

- **Feature Engineering:** Introducing polynomial terms, interaction features, or domain-specific knowledge could further enhance model performance.
- **Regularization:** Adding Ridge or Lasso penalties can prevent overfitting and improve generalization.
- **Non-linear Models:** Exploring decision trees, random forests, or gradient boosting can capture complex patterns missed by linear regression.
- **Hyperparameter Tuning:** Systematic tuning of learning rates, batch sizes, and iteration counts can optimize model training.

9 Conclusion

This study highlights the importance of efficient implementation in machine learning workflows. While the Pure Python version serves as a great educational tool for understanding gradient descent, it is far from practical for real datasets. NumPy's vectorized approach makes the process feasible, and scikit-learn delivers a production-ready solution with excellent performance and speed.