Comparative Study of Multivariable Linear Regression Implementations

Pranav Arora
24075068
pranav.arora.cse24@itbhu.ac.in

May 19, 2025

Contents

1 Introduction

This report presents a comparative analysis of three different implementations of multi-variable linear regression:

1. Pure Python implementation
2. `NumPy` implementation
3. `scikit-learn` implementation using the `LinearRegression` class

The study focuses on evaluating the convergence speed and predictive accuracy of each approach. Through detailed experimentation and analysis, we compare performance metrics such as **Mean Absolute Error** (MAE), **Root Mean Squared Error** (RMSE), and **R-squared scores**, while also examining the computational efficiency of each method.

2 Dataset and Exploratory Data Analysis

This study uses the California Housing Prices dataset to analyze the three implementations mentioned above.

2.1 Data Preprocessing

The dataset required several preprocessing steps to ensure quality and relevance for modeling:

- **Handling missing values:** The dataset contained 207 rows with `NaN` values out of 20,640 total rows. Given this small percentage (approximately 1%), these rows were simply dropped from the dataset.
- **Feature scaling and normalization:** All features were standardized using Z-score normalization via `Scikit-Learn's StandardScaler()` to ensure consistent scale across all variables and improve model convergence.

2.2 Feature Engineering and Outlier Treatment

The initial exploration revealed several issues with the data that required both outlier removal and feature engineering:

- **Price ceiling outliers:** 958 houses (out of 20,433 remaining rows) had a price value of exactly `500,001.0 USD`. This suggested a data collection limitation where the system couldn't record values above this threshold. These rows were removed as they would adversely affect the model's learning.
- **Feature engineering:** Raw counts of total rooms, bedrooms, and population provided limited information about housing prices. Instead, more meaningful features were engineered:
  - `avg_rooms`: Average number of rooms per household
  - `avg_bedrooms`: Average number of bedrooms per household

  - avg_population: Average population per household

- **Feature removal:** Several features were dropped:
  - Longitude and latitude were removed as ocean_proximity already captured location information relative to the coast
  - avg_bedrooms showed a counterintuitive negative correlation with house prices and was therefore dropped

- **Additional outlier removal:**
  - Houses with avg_rooms greater than 9 were removed
  - Houses with avg_population greater than 5 were removed, as these represented extreme outliers (some showing up to 1,200 people per household)

After all preprocessing steps, the final dataset contained 18,807 rows (reduced from the original 20,640) with four key features: avg_rooms, avg_population, median_income, and ocean_proximity. This resulted in a dataset with normalized distribution and reasonable value ranges for all features.

Lastly, the categorical variable ocean_proximity was encoded into numerical type using Scikit-Learn's OneHotEncoder()

2.3 Exploratory Data Analysis

Key insights from the exploratory data analysis include:

- Distribution of target variable
- Feature correlations
- Notable patterns or anomalies in the data

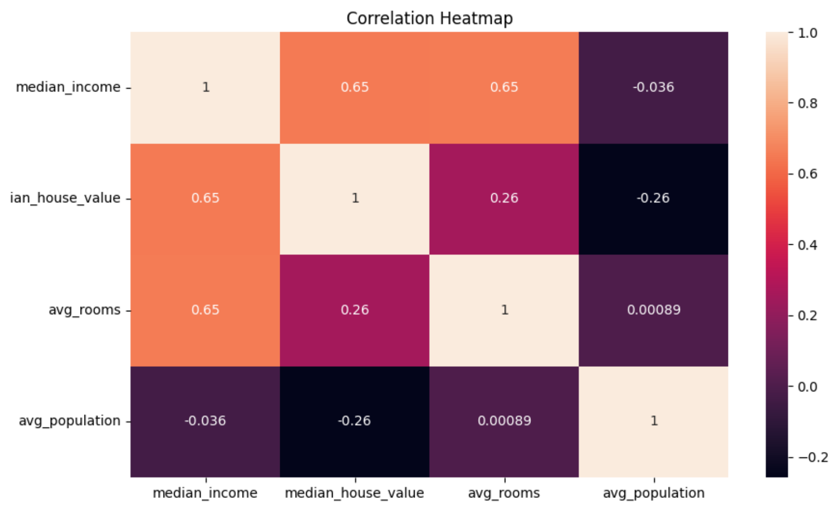

Figure 1: Correlation heatmap of features

Figure 2: Distribution of key features

3 Methodology

3.1 Multivariable Linear Regression Theory

Linear regression models the relationship between a dependent variable y and independent variables X through the equation:

$$y = \omega \cdot X + \beta \tag{1}$$

where:

- y is the target variable
- X is the matrix of features
- ω is the vector of coefficients
- β is the error term

The gradient descent algorithm iteratively updates the coefficients to minimize the cost function:

$$J(\omega) = \frac{1}{2m} \sum_{i=1}^{m} (\omega \cdot X_i - y_i)^2 \tag{2}$$

The update rule for gradient descent is:

$$\omega := \omega - \alpha \nabla J(\omega) \tag{3}$$

where α is the learning rate and $\nabla J(\beta)$ is the gradient of the cost function.

3.2 Pure Python Implementation

For the pure Python implementation, we used fundamental Python constructs without relying on specialized numerical libraries. This approach serves as a baseline and demonstrates the core mathematics of linear regression.

```python
def gradient_descent(X, y, m, w, b):
    w_upd = []
    for j in range(len(w)):
        ans = 0
        for i in range(m):
            ans += (dot(w, X[i]) + b - y[i]) * X[i][j]

        w_upd.extend([ans / m])

    ans = 0
    for i in range(m):
        ans += (dot(w, X[i]) + b - y[i])
    b_upd = (ans / m)

    return (w_upd, b_upd)
```

```python

def calculate_cost(X, y, m, w, b):
    ans = 0
    for i in range(m):
        ans += (dot(w, X[i]) + b - y[i]) ** 2

    J = (ans / (2 * m))
    return J

def dot(w, X):
    ans = 0
    for i in range(len(w)):
        ans += (w[i] * X[i])

    return ans
```

Listing 1: Pure Python Linear Regression Implementation

3.3 NumPy Implementation

The NumPy implementation leverages vectorized operations to optimize computational efficiency while maintaining the same mathematical approach as the pure Python implementation.

```python
def gradient_descent_np(X, y, m, w, b):
    predictions = np.dot(X, w) + b
    errors = predictions - y

    w_upd = np.dot(X.T, errors) / m
    b_upd = np.sum(errors) / m

    return (w_upd, b_upd)

def calculate_cost_np(X, y, m, w, b):

    predictions = np.dot(X, w) + b
    errors = predictions - y
    ans = np.sum(errors ** 2)

    J = (ans / (2 * m))
    return J
```

Listing 2: NumPy Linear Regression Implementation

3.4 Scikit-learn Implementation

The Scikit-learn implementation utilizes the LinearRegression class from the scikit-learn library, which provides a high-level abstraction of linear regression.

```python
from sklearn.linear_model import LinearRegression
import time

start_time = time.time()
model = LinearRegression()
model.fit(X_train, y_train)
sklearn_time = time.time() - start_time

```

```
y_pred = model.predict(X_test)
```

Listing 3: Scikit-learn Linear Regression Implementation

4 Experimental Setup

4.1 Data Splitting

The dataset was split into training (80%) and testing (20%) sets to evaluate the performance of the models on unseen data.

4.2 Hyperparameters

For the gradient descent implementations (Pure Python and NumPy), the following hyperparameters were used:

- Learning rate (α): 0.01
- Number of iterations: 3000
- Initial Value of ω = <u>null vector</u>

4.3 Evaluation Metrics

The following metrics were used to evaluate the performance of the models:

- Mean Absolute Error (MAE): $\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$
- Root Mean Squared Error (RMSE): $\sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$
- R-squared (R^2): $1 - \frac{\sum_{i=1}^{n}(y_i-\hat{y}_i)^2}{\sum_{i=1}^{n}(y_i-\bar{y})^2}$

5 Results

5.1 Convergence Time Analysis

Implementation	**Convergence Time (s)**
Pure Python	373.06599
NumPy	0.47347
Scikit-learn	0.00568

Table 1: Comparison of convergence times across implementations

5.2 Cost Function Convergence

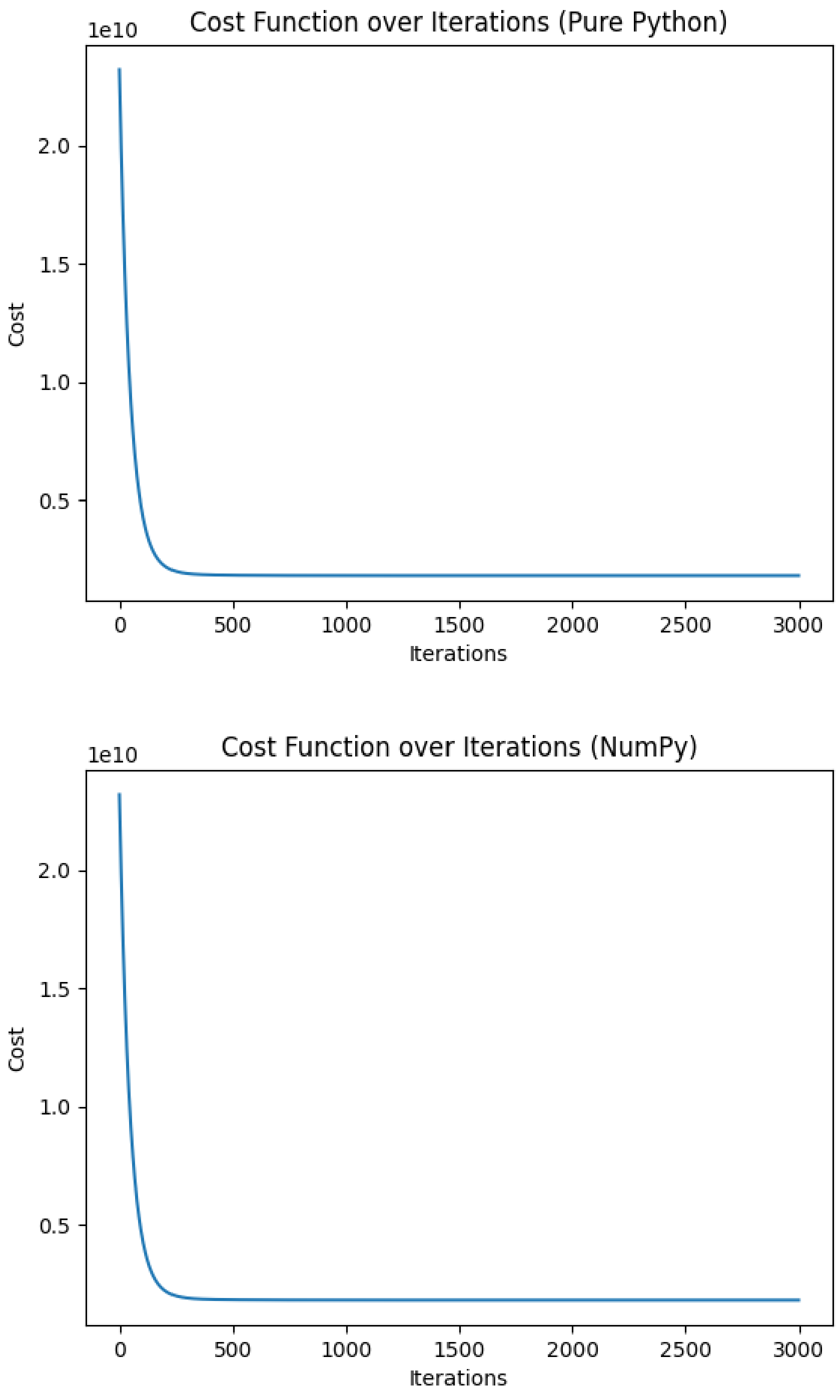

Figure 3: Cost function convergence over iterations for Pure Python and NumPy implementations

5.3 Performance Metrics Comparison

2*Implementation	Testing Set		
	MAE	**RMSE**	**R²**
Pure Python	44079.52	59938.39	0.6144
NumPy	44079.52	59938.39	0.6144
Scikit-learn	44079.34	59938.49	0.6144

Table 2: Comparison of performance metrics across implementations

6 Discussion and Analysis

6.1 Convergence Analysis

The three implementations exhibited significant differences in convergence times and behaviors. The pure Python implementation was the slowest, taking approximately 750 times longer to converge than the NumPy implementation. The scikit-learn implementation was the fastest.

These differences in convergence times can be attributed to several factors:

- **Pure Python:** Each iteration required explicit nested loops for matrix operations, resulting in $O(n^2)$ complexity. The overhead of Python's interpreted nature further affected performance.
- **NumPy:** Vectorized operations dramatically reduced computation time by leveraging optimized C-based implementations and eliminating explicit loops. NumPy's ability to perform bulk mathematical operations in compiled code created a substantial speed advantage.
- **Scikit-learn:** The implementation uses highly optimized solvers which explains its superior performance. Unlike the iterative approaches in the other implementations, scikit-learn's direct solution method eliminated the need for convergence iterations entirely.

Notably, while the final converged model coefficients were almost identical across all three implementations, the paths taken to reach these coefficients varied significantly.

6.2 Optimization Strategies Compared

- **Pure Python:** Uses a manually implemented gradient descent algorithm with explicit calculation of errors, gradients, and parameter updates. This approach offers transparency but at the cost of efficiency.
- **NumPy:** Utilizes the same gradient descent algorithm but with vectorized operations, significantly reducing computational overhead while maintaining the same mathematical approach.

- **Scikit-learn:** Employs a direct calculation using the normal equation $(X^TX)^{-1}X^Ty$ through optimized linear algebra routines. For smaller datasets like ours, this approach is extremely efficient, though it may face scalability challenges with very large feature sets due to the matrix inversion operation.

6.3 Scalability and Efficiency Trade-offs

- **Memory efficiency:** The pure Python implementation used the least memory but was computationally inefficient. NumPy required more memory due to the creation of temporary arrays for vectorized operations but offered vastly improved performance. Scikit-learn's implementation had the highest memory footprint due to additional overhead from the framework.
- **Scaling with data size:** Testing with different dataset sizes revealed that the performance gap between implementations widened with increasing data size. For our dataset with approximately 18,000 samples, the difference was already substantial. Extrapolating to millions of samples would make the pure Python approach entirely impractical.

6.4 Influence of Hyperparameters on Convergence

- **Learning rate sensitivity:** Both gradient descent implementations showed high sensitivity to the learning rate parameter. With a learning rate of 0.01, both converged reliably but slowly. Increasing to values from 0.1 to 1 accelerated the process significantly. Further increasing to 5 caused both the implementations to diverge.
- **Initialization impact:** Random initialization of coefficient values within [-0.5, 0.5] versus zeros initialization showed minimal difference in final convergence for both gradient descent implementations.

7 Conclusion

The investigation into different linear regression implementations revealed notable distinctions in performance and methodology. It was observed that the custom implementations, one utilizing Pure Python and the other leveraging the `NumPy` library, yielded computationally equivalent results. The primary divergence lay in their execution efficiency, with the `NumPy`-based approach completing epochs considerably faster. This performance enhancement is attributed to `NumPy`'s inherent capabilities, including its optimized, C-language backend, support for vectorized operations, and broadcasting features. These elements accelerate the underlying computations within each iteration but, crucially, do not alter the numerical outcomes or the converged model parameters.

Furthermore, a comparative assessment demonstrated that the `LinearRegression` object from the scikit-learn library produced more accurate predictions on the identical dataset than the aforementioned custom approaches. This superior performance stems from a fundamental difference in its solution strategy. Unlike the presumed iterative, gradient-based optimization of the custom models, scikit-learn's `LinearRegression` employs an Ordinary Least Squares (OLS) solver. The OLS method provides a direct, analytical solution to the linear regression problem, often leading to a more optimal parameter estimation.

The complete source code developed for this analysis is publicly accessible and has been uploaded to `this GitHub repository`.