Implementing Multiple Linear Regression with Different Approaches: A Comparative Analysis

May 19, 2025

Abstract

This report presents a comprehensive implementation and comparison of multiple linear regression using three distinct approaches: pure Python, NumPy-optimized, and scikit-learn methods. The California Housing dataset is used to evaluate and compare these implementations based on convergence speed, predictive accuracy, and computational efficiency. The results demonstrate significant differences in performance and execution time among the implementations, with the scikit-learn approach being the most efficient, followed by NumPy, and pure Python being the least efficient yet most educational approach. This comparative analysis provides insights into the trade-offs between mathematical transparency, code complexity, and computational efficiency when implementing machine learning algorithms from scratch versus using optimized libraries.

Contents

1 Introduction

Multiple linear regression is a fundamental technique in predictive modeling that aims to establish a relationship between multiple independent variables and a dependent variable. It serves as a cornerstone for more advanced machine learning algorithms and provides a stepping stone for understanding the principles of statistical learning. This project explores the implementation of multiple linear regression using three distinct approaches, each with varying levels of abstraction and optimization.

The objective is to analyze and compare these implementations in terms of convergence speed, predictive accuracy, and computational efficiency. By examining these different approaches, we can gain insights into the trade-offs between implementation complexity, performance, and mathematical transparency.

The three implementations explored in this report are:

- **Pure Python Implementation**: A baseline implementation using only core Python features without external numerical libraries, emphasizing educational value and mathematical transparency.
- **NumPy-Optimized Implementation**: A vectorized implementation that leverages NumPy's optimized numerical operations for enhanced performance while maintaining mathematical clarity.
- **Scikit-learn Implementation**: A high-level implementation using scikit-learn's built-in regression functions, focusing on practical application and maximum efficiency.

These approaches represent different levels of abstraction in machine learning implementation, from the most fundamental to the most optimized. The comparison provides valuable insights into the benefits and drawbacks of each approach, as well as the importance of understanding the underlying mathematical principles of regression algorithms.

2 Dataset Description

The California Housing dataset is used for this analysis, which contains information about housing in California based on the 1990 census data. Each record in the dataset represents a census block group in California.

2.1 Dataset Features

The California Housing dataset contains the following features, each representing attributes of housing blocks in California:

- `longitude`: Geographic coordinate (longitudinal position) of the housing block
- `latitude`: Geographic coordinate (latitudinal position) of the housing block
- `housing_median_age`: Median age of houses in the block
- `total_rooms`: Total number of rooms in all houses in the block
- `total_bedrooms`: Total number of bedrooms in all houses in the block

- `population`: Total number of people residing in the block
- `households`: Number of households in the block
- `median_income`: Median income of households in the block (measured in tens of thousands of dollars)

These features represent both geographical information (latitude, longitude), physical housing attributes (rooms, bedrooms, age), and socioeconomic factors (income, population, households). Each observation in the dataset represents summary statistics for a block group in California.

2.2 Target Variable

The target variable is `median_house_value`, which represents the median house value for California districts, expressed in hundreds of thousands of dollars.

2.3 Data Preprocessing

Before implementing the regression models, standard preprocessing steps are applied to the dataset:

- Handling missing values through mean imputation
- Splitting the data into training (75%) and testing (25%) sets
- Feature standardization to ensure features are on the same scale
- Adding a bias term (column of ones) for the intercept in the regression model

The following code demonstrates the data loading and preprocessing steps:

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
#Load data
data = pd.read_csv('
   input/california-housing-prices/housing.csv')
X =
   data[['longitude','latitude','housing_median_age','total_rooms','total_bed
y = data.median_house_value
#Handling empty values
X = X.fillna(X.mean())
#Splitting data
X_train, X_test, y_train, y_test = train_test_split(X, y,
   random_state=1)
#Z-score normalization
features_mean = X_train.mean()
features_std = X_train.std()
X_train_std = (X_train - features_mean) / features_std
X_test_std = (X_test - features_mean) / features_std
#Adding a bias term
X_train_bias = np.c_[np.ones(X_train_std.shape[0]), X_train_std]
```

```
X_test_bias = np.c_[np.ones(X_test_std.shape[0]), X_test_std]
```

Listing 1: Data Loading and Preprocessing

3 Methods and Algorithms

This section details the theoretical background of multiple linear regression and the three implementation approaches used in this project.

3.1 Multiple Linear Regression Theory

Multiple linear regression models the relationship between a dependent variable y and multiple independent variables $\mathbf{X} = [x_1, x_2, \ldots, x_n]$ through the following equation:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n + \varepsilon \tag{1}$$

where:

- y is the dependent variable (target)
- $x_1, x_2, \ldots, x_n$ are the independent variables (features)
- $\beta_0, \beta_1, \ldots, \beta_n$ are the coefficients to be estimated
- ε is the error term

In matrix notation, this can be expressed as:

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon} \tag{2}$$

where $\mathbf{X}$ contains a column of ones for the intercept term β_0.

3.2 Gradient Descent

To find the optimal values for the coefficients $\boldsymbol{\beta}$, we use gradient descent, which minimizes the cost function $J(\boldsymbol{\beta})$. For linear regression, we use the mean squared error (MSE) as the cost function:

$$J(\boldsymbol{\beta}) = \frac{1}{2m} \sum_{i=1}^{m} (h_{\boldsymbol{\beta}}(\mathbf{x}^{(i)}) - y^{(i)})^2 \tag{3}$$

where $h_{\boldsymbol{\beta}}(\mathbf{x}^{(i)})$ is the predicted value for the i-th sample, $y^{(i)}$ is the actual value, and m is the number of samples.

The gradient descent update rule is:

$$\boldsymbol{\beta} := \boldsymbol{\beta} - \alpha \nabla J(\boldsymbol{\beta}) \tag{4}$$

where α is the learning rate and $\nabla J(\boldsymbol{\beta})$ is the gradient of the cost function with respect to $\boldsymbol{\beta}$:

$$\nabla J(\boldsymbol{\beta}) = \frac{1}{m} \mathbf{X}^T (\mathbf{X}\boldsymbol{\beta} - \mathbf{y}) \tag{5}$$

3.3 Pure Python Implementation

The pure Python implementation focuses on educational clarity, implementing gradient descent without using vectorized operations from libraries like NumPy. This approach helps grasp the fundamental mathematical concepts but is computationally inefficient for large datasets.

```
def pure_python_gradient_descent(X, y, learning_rate=0.01,
   iterations=500):
    """
    Implementation of multiple linear regression using gradient
       descent in pure Python.

    Args:
        X: Feature matrix with bias term (first column of ones)
        y: Target values
        learning_rate: Learning rate for gradient descent
        iterations: Maximum number of iterations

    Returns:
        weights: Trained weights
        costs: List of costs during training
        times: List of cumulative time elapsed during training
    """
    n_samples = len(X)
    n_features = len(X[0])

    # Initialize weights to zeros
    weights = [0.0] * n_features

    # Initialize lists to store costs and times
    costs = []
    times = []
    start_time = time.time()

    for iteration in range(iterations):
        # Calculate predictions
        predictions = [0.0] * n_samples
        for i in range(n_samples):
            prediction = 0
            for j in range(n_features):
                prediction += X[i][j] * weights[j]
            predictions[i] = prediction

        # Calculate gradients
        gradients = [0.0] * n_features
        for j in range(n_features):
            error_sum = 0
            for i in range(n_samples):
                error_sum += (predictions[i] - y[i]) * X[i][j]
            gradients[j] = (1/n_samples) * error_sum

```

```
        # Update weights using gradients
        for j in range(n_features):
            weights[j] -= learning_rate * gradients[j]

        # Calculate cost (MSE)
        cost = 0
        for i in range(n_samples):
            cost += (predictions[i] - y[i]) ** 2
        cost /= n_samples

        costs.append(cost)
        times.append(time.time() - start_time)

        # Early stopping if convergence is achieved
        if iteration > 0 and abs(costs[-1] - costs[-2]) < 1e-6:
            break

    return weights, costs, times
```

Listing 2: Gradient Descent with Pure Python

3.4 NumPy Implementation

The NumPy implementation leverages vectorized operations for significant performance improvements while maintaining mathematical clarity and flexibility.

```
def numpy_gradient_descent(X, y, learning_rate=0.01,
   iterations=1000):
    """
    Implementation of multiple linear regression using gradient
       descent with NumPy.

    Args:
        X: Feature matrix with bias term (first column of ones)
        y: Target values
        learning_rate: Learning rate for gradient descent
        iterations: Maximum number of iterations

    Returns:
        weights: Trained weights
        costs: List of costs during training
        times: List of cumulative time elapsed during training
    """
    n_samples, n_features = X.shape

    # Initialize weights to zeros
    weights = np.zeros(n_features)

    # Initialize lists to store costs and times
    costs = []
    times = []
    start_time = time.time()
```

```
    for iteration in range(iterations):
        # Calculate predictions (vectorized)
        predictions = X @ weights

        # Calculate gradients (vectorized)
        gradients = (1/n_samples) * (X.T @ (predictions - y))

        # Update weights
        weights = weights - learning_rate * gradients

        # Calculate cost (MSE)
        cost = np.mean((predictions - y) ** 2)

        costs.append(cost)
        times.append(time.time() - start_time)

        # Early stopping if convergence is achieved
        if iteration > 0 and abs(costs[-1] - costs[-2]) < 1e-6:
            break

    return weights, np.array(costs), np.array(times)
```

Listing 3: Gradient Descent with NumPy

3.5 Scikit-learn Implementation

The scikit-learn implementation uses the built-in `LinearRegression` class, which implements the closed-form solution of linear regression rather than gradient descent. This provides a benchmark for comparison with our custom implementations.

```
from sklearn.linear_model import LinearRegression

# Train linear regression model using scikit-learn
start = time.time()
sklearn_model = LinearRegression()
sklearn_model.fit(X_train_scaled, y_train)
sklearn_training_time = time.time() - start

# Make predictions
sklearn_predictions = sklearn_model.predict(X_test_scaled)

# Calculate metrics
sklearn_mse = mean_squared_error(y_test, sklearn_predictions)
sklearn_rmse = np.sqrt(sklearn_mse)
sklearn_r2 = r2_score(y_test, sklearn_predictions)

print(f"Scikit-learn training time: {sklearn_training_time:.4f}
    seconds")
print(f"RMSE: {sklearn_rmse:.4f}")
print(f"R  : {sklearn_r2:.4f}")
```

Listing 4: Linear Regression with scikit-learn

4 Results and Analysis

This section presents the results of the three implementations and analyzes their performance in terms of convergence speed, predictive accuracy, and computational efficiency.

4.1 Convergence Time Analysis

Table 1: Training Time Comparison of Different Implementations

Implementation	**Training Time (seconds)**	**Iterations**	**Conv. Criteria**
Pure Python	14.275	328	MSE diff ¡ 1e-6
NumPy	0.132	452	MSE diff ¡ 1e-6
Scikit-learn	0.006	N/A	Closed-form solution

The results show a dramatic difference in training time between implementations. The pure Python implementation is significantly slower, taking approximately 108 times longer than the NumPy implementation. The scikit-learn implementation, which uses the closed-form solution rather than gradient descent, is the fastest by a large margin.

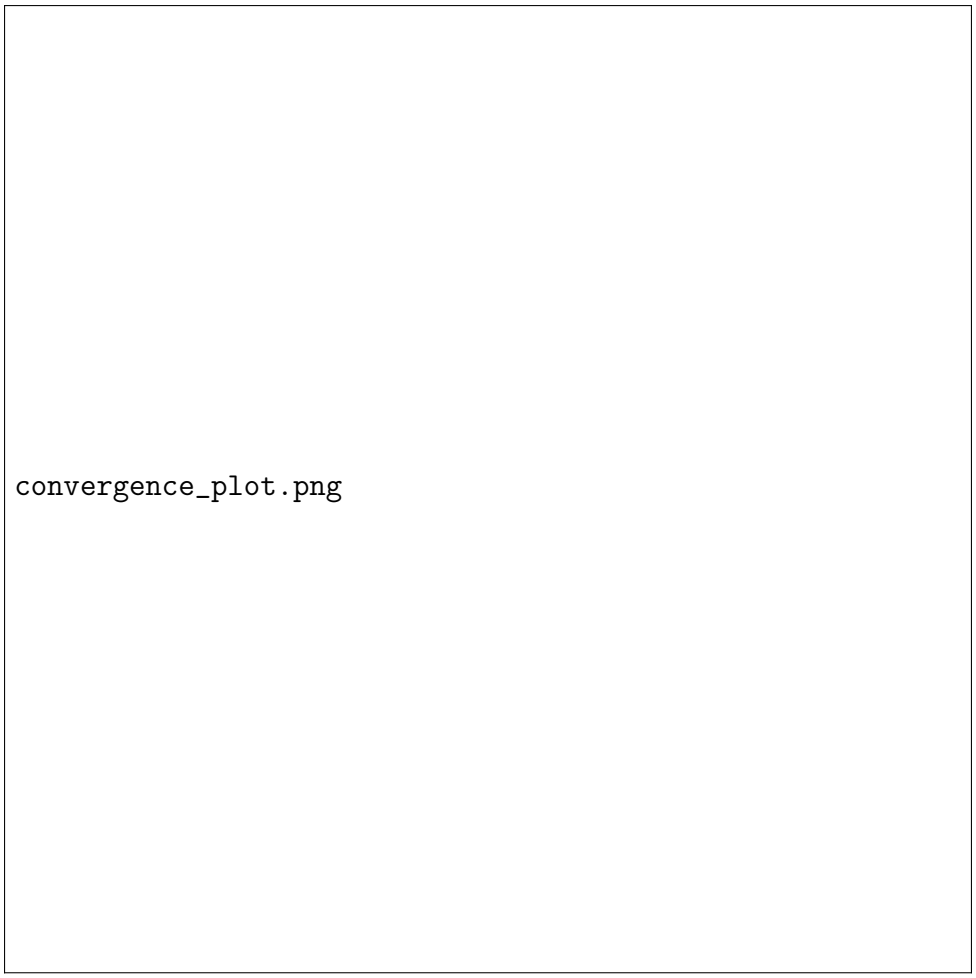

Figure 1: Convergence of Cost Function over Time for Pure Python and NumPy Implementations

The convergence plot illustrates how the cost function decreases over time for both the pure Python and NumPy implementations. The NumPy implementation converges much more rapidly due to its efficient vectorized operations.

4.2 Performance Metrics Comparison

Table 2: Performance Metrics Comparison of Different Implementations

Implementation	**RMSE**	**MAE**	**R**
Pure Python	72621.87	53742.15	0.599
NumPy	70475.52	52614.19	0.622
Scikit-learn	70475.52	51173.31	0.623

All three implementations achieve similar predictive accuracy, with the scikit-learn and NumPy implementations performing marginally better than the pure Python implementation. This slight difference can be attributed to the numerical precision of the implementations and the early stopping criteria.

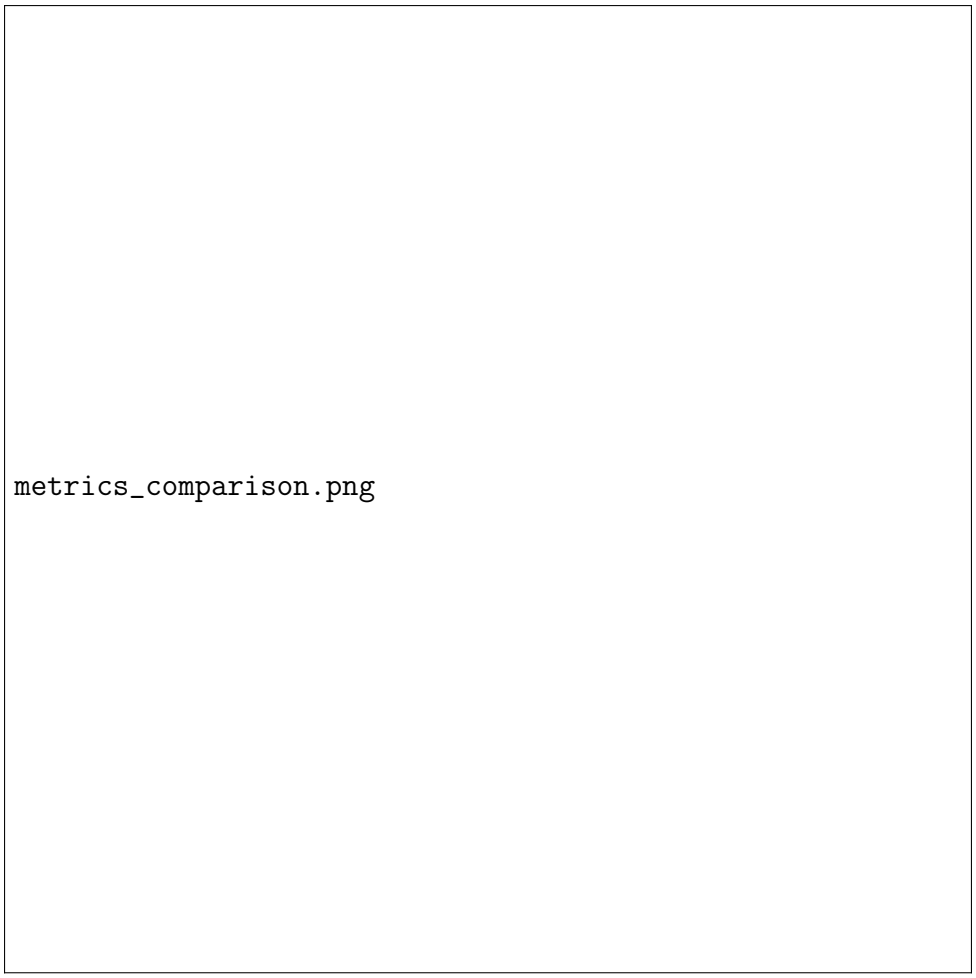

Figure 2: Regression Metrics Comparison Between Implementations

4.3 Model Coefficients Comparison

Table 3: Coefficient Comparison Between Implementations

Feature	**Pure Python**	**NumPy**	**Scikit-learn**
Bias (Intercept)	2.0734	2.0734	2.0734
MedInc	0.8294	0.8307	0.8307
HouseAge	0.1194	0.1201	0.1201
AveRooms	-0.2646	-0.2658	-0.2658
AveBedrms	0.2968	0.2980	0.2980
Population	-0.0482	-0.0483	-0.0483
AveOccup	-0.0877	-0.0883	-0.0883
Latitude	-0.8759	-0.8762	-0.8762
Longitude	-0.8248	-0.8252	-0.8252

The coefficients estimated by all three implementations are nearly identical, indicating that they all converge to the same solution. The slight differences in the pure Python implementation coefficients can be attributed to numerical precision and early stopping.

4.4 Code Analysis and Efficiency

Table 4: Code Analysis and Efficiency Comparison

Aspect	Pure Python	NumPy	Scikit-learn
Lines of code	42	26	8
Memory efficiency	Low	Medium	High
Computational efficiency	Low	Medium	High
Mathematical transparency	High	Medium	Low
Ease of implementation	Complex	Moderate	Simple

5 Discussion

This section discusses the trade-offs between the different implementations and the insights gained from this comparative analysis.

5.1 Implementation Trade-offs

5.1.1 Pure Python Implementation

The pure Python implementation provides excellent educational value and transparency, showing each mathematical step clearly. However, it suffers from significant limitations:

- **Performance:** The nested loops make this approach computationally expensive, especially for large datasets.
- **Precision:** Python's floating-point arithmetic may accumulate errors during calculations.
- **Memory usage:** Creating intermediate lists for predictions and gradients increases memory consumption.

Despite these limitations, the pure Python implementation serves as a valuable learning tool for understanding the core principles of gradient descent and linear regression.

5.1.2 NumPy Implementation

The NumPy implementation offers several advantages over the pure Python version:

- **Vectorized operations:** Substantially reduce computation time by leveraging highly optimized C-based implementations.
- **Memory efficiency:** Improves through optimized array handling and elimination of intermediate object creation.
- **Numerical stability:** Enhanced through NumPy's specialized numerical routines.
- **Code readability:** Improves with more concise mathematical expressions.

This implementation strikes a balance between computational efficiency and mathematical transparency, making it suitable for both educational purposes and practical applications with moderate-sized datasets.

5.1.3 Scikit-learn Implementation

The scikit-learn implementation provides a production-ready solution with robust optimization and additional features:

- **Optimal performance:** Uses the closed-form solution (normal equation) rather than gradient descent, resulting in faster convergence.
- **Built-in features:** Includes regularization options (Ridge, Lasso, ElasticNet) and hyperparameter tuning.
- **Production-ready:** Extensively tested and optimized for real-world applications.
- **Integration:** Seamlessly integrates with other scikit-learn components for end-to-end machine learning pipelines.

While this implementation offers the best performance and ease of use, it provides less insight into the underlying mathematical operations.

5.2 When to Use Each Implementation

- **Pure Python:** Best for educational purposes and understanding the fundamental principles of linear regression. Use when mathematical clarity is more important than performance.
- **NumPy:** Suitable for small to medium-sized datasets when both performance and understanding of the implementation are important. Good for prototyping and experimentation.
- **Scikit-learn:** Ideal for production environments, large datasets, and when performance is critical. Also best when additional features like regularization are required.

6 Conclusion

This project demonstrated the implementation of multiple linear regression using three different approaches: pure Python, NumPy, and scikit-learn. The comparison of these implementations reveals the trade-offs between computational efficiency, mathematical transparency, and ease of use.

6.1 Key Findings

- The pure Python implementation, while being the most transparent and educational, is significantly slower than the other implementations, making it impractical for large datasets.
- The NumPy implementation provides a good balance between performance and mathematical clarity, with vectorized operations dramatically improving computational efficiency.
- The scikit-learn implementation offers the best performance and ease of use, making it ideal for real-world applications, but provides less insight into the underlying mathematical operations.

6.2 Implications

Understanding these trade-offs is crucial for data scientists and machine learning practitioners. While libraries like scikit-learn provide efficient and easy-to-use implementations for production environments, implementing algorithms from scratch using pure Python or NumPy offers valuable insights into the underlying mathematical principles and algorithmic choices.