Multivariable Linear Regression on California Housing Dataset

Divyansh Soni
Roll No: 24124018

May 19, 2025

Abstract

This report documents the implementation and evaluation of multivariable linear regression using the California Housing Prices dataset. The project explores three approaches: a pure Python implementation using gradient descent, an optimized version using NumPy, and a baseline using Scikit-learn's LinearRegression. The study includes data preprocessing, feature selection, normalization, and a comparison of convergence time and regression performance metrics across all methods.

1 Introduction

The goal of this project is to predict median house values using multiple features from the California Housing dataset. We compare three regression approaches to understand performance, efficiency, and trade-offs.

2 Exploratory Data Analysis

3 Dataset Overview

The dataset includes features such as median income, average rooms, population, and a categorical feature: ocean proximity. We perform correlation-based feature selection, one-hot encoding, and normalization.

We observe that median income has the strongest correlation with house prices. Ocean proximity was converted into numerical form using one-hot encoding.

4 Implementation

4.1 Part 1: Pure Python Linear Regression

Implemented using core Python features with list operations and gradient descent.

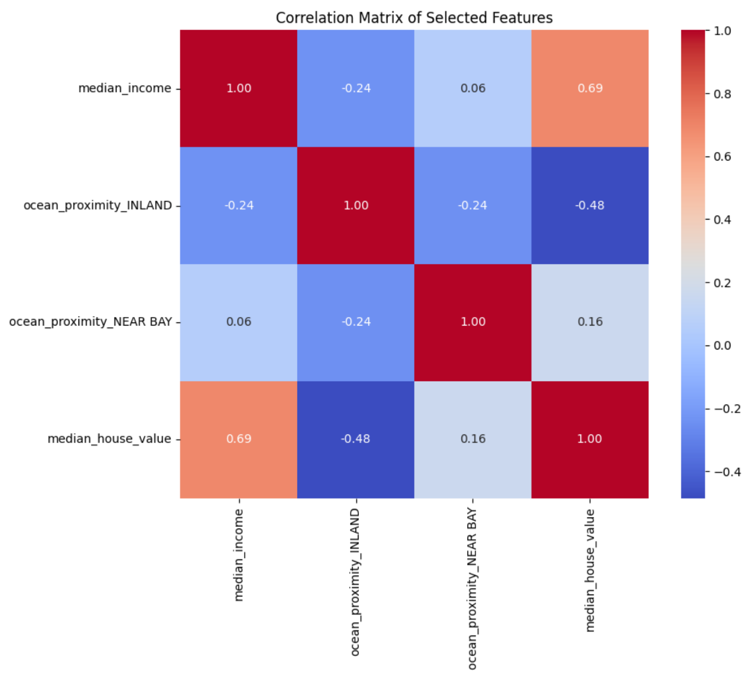

Figure 1: Correlation Matrix Heatmap

Gradient Descent Logic in Pure Python. In this implementation, we use Gradient Descent to optimize the weights for multivariable linear regression. The goal is to minimize the Mean Squared Error (MSE) cost function, defined as:

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

where $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$ is the predicted house price, θ are the model parameters, and m is the number of training samples.

In the pure Python version, we manually compute the gradient for each parameter using basic list operations and for-loops. The update rule for each parameter θ_j is:

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

This means for each iteration, we:

- Calculate the prediction error for every training example,
- Compute the partial derivative of the cost function with respect to each parameter,
- Update each parameter using the learning rate α.

While this approach is slower compared to vectorized implementations, it provides a clear, step-by-step understanding of how gradient descent works internally.

4.2 Part 2: NumPy Optimized Linear Regression

Same logic as above but rewritten using NumPy arrays and matrix operations for faster computation.

4.3 Part 3: Scikit-learn Linear Regression

Utilized `LinearRegression` from the `sklearn.linear_model`

5 Evaluation and Results

5.1 Convergence Time

- Pure Python: 9.69 seconds
- NumPy: 0.085 seconds
- Scikit-learn: 0.0016 seconds

5.2 Evaluation Metrics on Validation Set

Pure Python

- MAE: 54746.42144414328
- RMSE: 75248.76652421166
- R^2 Score: 0.5859376960345541

NumPy

- MAE: 54746.42144414328
- RMSE: 75248.76652421166
- R^2 Score: 0.5859376960345541

Scikit-learn

- MAE: 54747.71203219531
- RMSE: 75248.50194391274
- R^2 Score: 0.5859406077775801

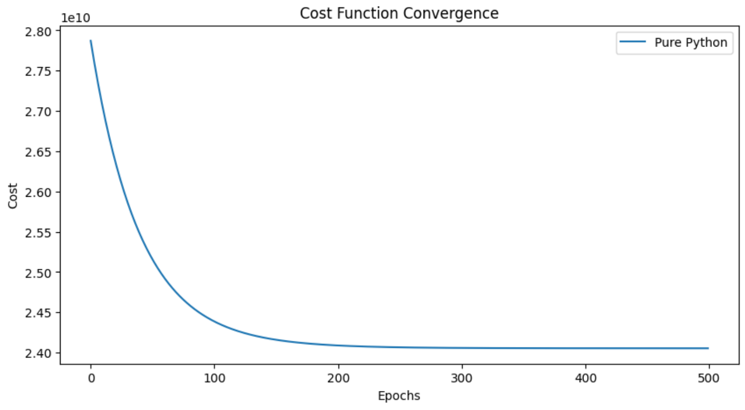

Figure 2: Cost Function Convergence - Pure Python

5.3 Convergence Plots

6 Analysis and Discussion

The NumPy version outperforms the pure Python version in terms of speed, because of vectorization. Scikit-learn is the fastest due to the strong backend although it uses the OLS method which, in large data sets, can be costly to calculate (owing to the complexity of time of O (n3). Numpy and Pure Python models reached similar accuracy because both used the same Gradient Descent optimization algorithm, which is more computation-friendly and a little less accurate than the OLS method used by the Sklearn Library.

Initial weights and learning rates had a significant impact on convergence time. Too high a learning rate led to divergence in some runs, especially in the pure Python version.

7 Conclusion

This project demonstrates how implementation choices (pure Python vs. NumPy vs. Scikit-learn) impact model training efficiency. NumPy strikes a balance between speed and control, while Scikit-learn offers unmatched convenience.

References

- Scikit-learn Documentation: `https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html`
- California Housing Dataset: `https://www.kaggle.com/datasets/camnugent/california-housing`

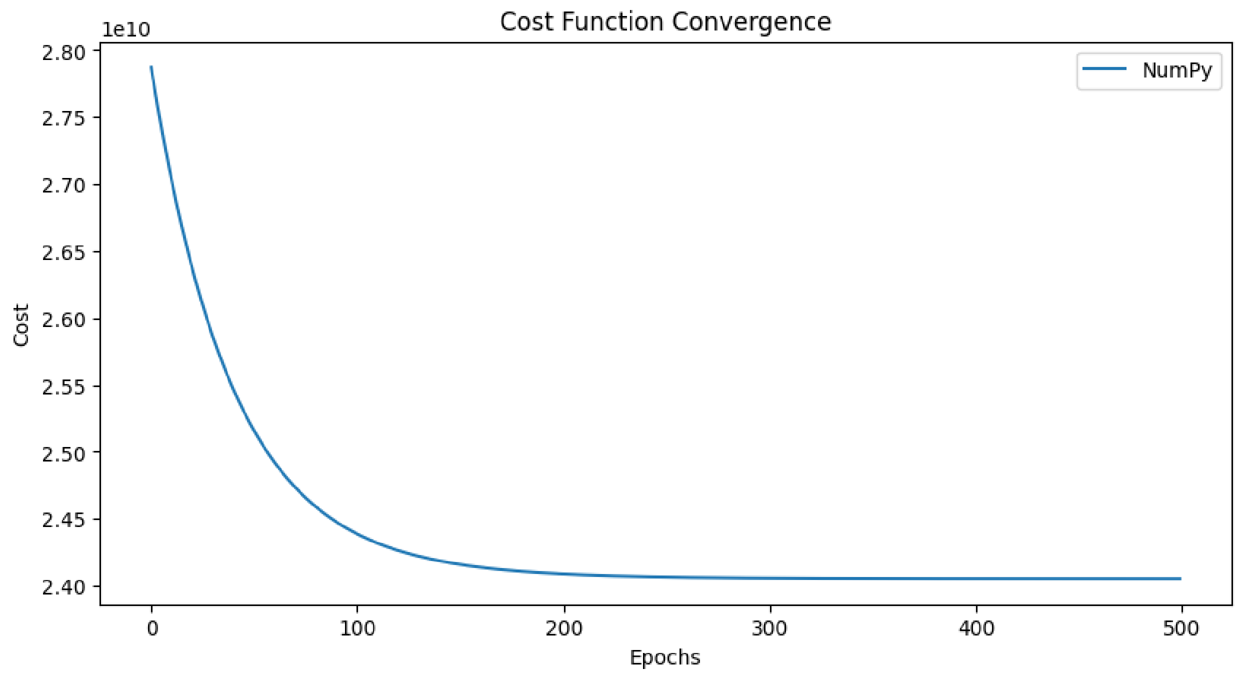

Figure 3: Enter Caption