Comparison of Linear Regression Implementations: Pure Python, NumPy, and Scikit-learn

Shreshth Vishwakarma
Roll number :24095105
CSOC IG

May 19, 2025

Chapter 1

Introduction

Linear regression is a foundational technique in machine learning for modeling the relationship between a set of features and a continuous target variable. This report compares three implementations:

- Pure Python (manual loops and calculations)
- NumPy (vectorized operations)
- Scikit-learn (library)

The objective is to analyze their differences in terms of approach, computational efficiency, and predictive performance.

Chapter 2

Methodology

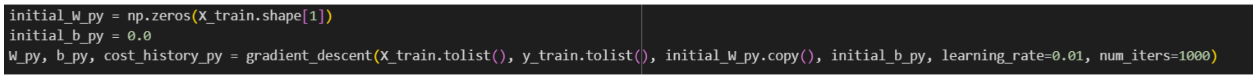
2.1 Data Preparation

- The California housing dataset was loaded from a CSV file stored in the mounted google drive as the code was written on google colab, so using g drive was the most feasible way. The loading dataset part of the code was written using ChatGPT.
- Features were normalized using the Z-score normalization (commonly used as it prevents the cost function to be elongated, rather it makes a nearly uniform cost function which helps to run the gradient descent much faster) to zero mean and standard variance to ensure fair convergence during gradient descent.
- The dataset was split into 70% training and 30% cross-validation sets for giving the model appropriate number of examples to cross validate with new data. Also used "random_state" in the train_test_split() function to divide the same examples in train and CV set, because train_test_split() function randomly divides the dataset, which is not feasible for fair comparison.

2.2 Implementations

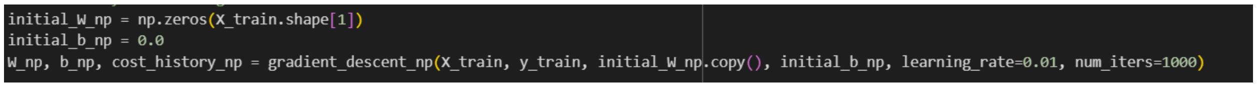
2.2.1 Pure Python

- All computations, including predictions, cost, and gradients, were implemented using explicit loops. Gradient descent was performed manually, without the use NumPy anywhere !!
- The parameters(weights and biases) are initialized to zero, as there is only one set of weights and no layers (as in neural networks), and it's a convex optimization problem. The gradient directs toward the global minimum, regardless of initial values.
- The use of initial_W_np.copy() [see figure 2.1] in the gradient descent call header ensures that the original initial_W_np array remains unchanged, otherwise it will be changed as NumPy arrays are mutable. While it is of no use in this particular code(if not debugging or rerunning) but is a good practice if doing so.



```
initial_W_py = np.zeros(X_train.shape[1])
initial_b_py = 0.0
W_py, b_py, cost_history_py = gradient_descent(X_train.tolist(), y_train.tolist(), initial_W_py.copy(), initial_b_py, learning_rate=0.01, num_iters=1000)
```

Figure 2.1:



```
initial_W_np = np.zeros(X_train.shape[1])
initial_b_np = 0.0
W_np, b_np, cost_history_np = gradient_descent_np(X_train, y_train, initial_W_np.copy(), initial_b_np, learning_rate=0.01, num_iters=1000)
```

Figure 2.2:

- Also using tolist() function with X_train and Y_train in the gradient descent call header of pure python implementation, because it uses lists everywhere and not NumPy arrays.

2.2.2 NumPy

- The same algorithm as above was implemented using NumPy using vectorized operations, significantly reducing computational time.
- Similar to the previous implementation, the parameters(weights and biases) are initialized to zero, as there is only one set of weights and no layers (as in neural networks), and it is a convex optimization problem. The gradient directs toward the global minimum, regardless of initial values.
- The use of initial_W_np.copy() [see figure 2.1] in the gradient descent call header ensures that the original initial_W_np array remains unchanged, otherwise it will be changed as NumPy arrays are mutable. While it is also of no use in this particular code(if not debugging or rerunning) but is a good practice if doing so.

2.2.3 Scikit-learn

- The `LinearRegression` model from Scikit learn was used, leverging highly optimised fitting and prediction.
- It does not use gradient descent. There is no iterative process requiring initial values.
- The error was calculated directly with the use of Scikit learn;s metrics module.
- Time was calculated using the time module in all the three cases.

Chapter 3

Results and Discussion

3.1 Cost Function Convergence

- **Pure Python:**
- The cost trajectory was an exponentially decaying curve.
- Cost decreased from 0.495 to 0.194 over 1000 iterations.
- Computation time: 175.871535 seconds .
- **NumPy:**
- The cost trajectory was exactly identical to the above cost trajectory because of using the same formulations for cost function.
- But computation time reduced to 0.279125 seconds, which was obvious because the vectorized implementation is much much faster than using explicit for loops.
- **Scikit-learn:**
- Achieved similar final cost (training cost: 0.183), but less than the previous implementations.
- Computation time of 0.063424 seconds, even smaller than the NumPy implementation, which shows that Scikit Learn is a fastest implementation out of three.

Implementation	**Computation time**
Pure Python	175.871535 seconds
NumPy	0.279125 seconds
Scikit-learn	0.063424 seconds

Table 3.1: Computation time

Implementation	**MAE (Train)**	**RMSE (Train)**	**R^2 (Train)**
Pure Python	0.45554290158339433	0.6229983018860678	0.6117787091814388
NumPy	0.45554290158339433	0.6229983018860678	0.6117787091814388
Scikit-learn	0.4413944064280527	0.6053910432709143	0.6334125389213838

Table 3.2: Training Performance Metrics

Implementation	**MAE (CV)**	**RMSE (CV)**	**R^2 (CV)**
Pure Python	0.45616452681872216	0.6184778446996979	0.6176628088423946
NumPy	0.45616452681872216	0.6184778446996979	0.6176628088423946
Scikit-learn	0.43925904162368795	0.5963657838492047	0.6445130291082352

Table 3.3: Cross Validation Performance Metrics

3.2 Performance Metrics

Note: Pure Python and NumPy match because they are implementated by using identical functions and identical initial values for the parameters.[see Table 3.2 and 3.3]

- MAE and RMSE: Lower on the training set, 20% higher on the cross-validation set.
- $\mathbf{R^2}$ Score: Higher on the training set, 5.88% decrease on the cross-validation set.
- This shows that validations errors are higher (MAE and RMSE) that means the model somewhat not faces difficulty for predicting for the new data.

3.3 Comparison

- **Efficiency:** NumPy and Scikit-learn are orders of magnitude faster than Pure Python due to vectorization and optimized libraries.
- **Accuracy:** All three methods converge to similar cost values and performance metrics, indicating correct implementation.

3.4 Visualization

- ChatGPT was used in the code for plotting the cost function convergence graph, as I was not confident with plotting graphs in python. But learned it now :)
- (Fig. 3.2) The Pure pyhton and NumPy have the identical convergence speeds as they need identical number of iterations to reach to their convergences. While the Scikit-learn does not use iterative optimization for linear regression, rather it computes the solution in one step. Therefore, it does not have a "convergence curve" just a final cost value.

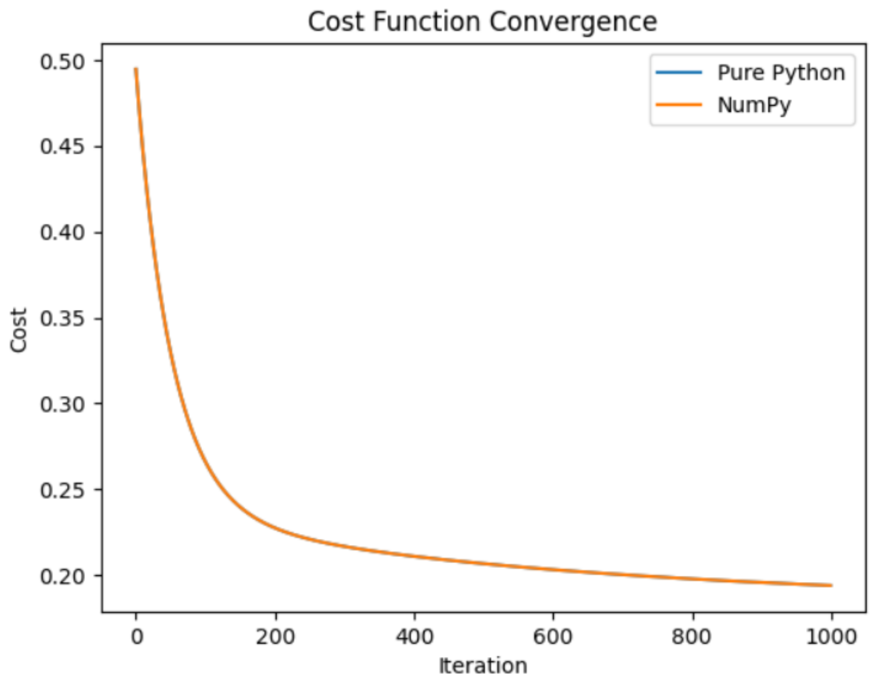


Figure 3.1: Cost Function Convergence for All Implementations

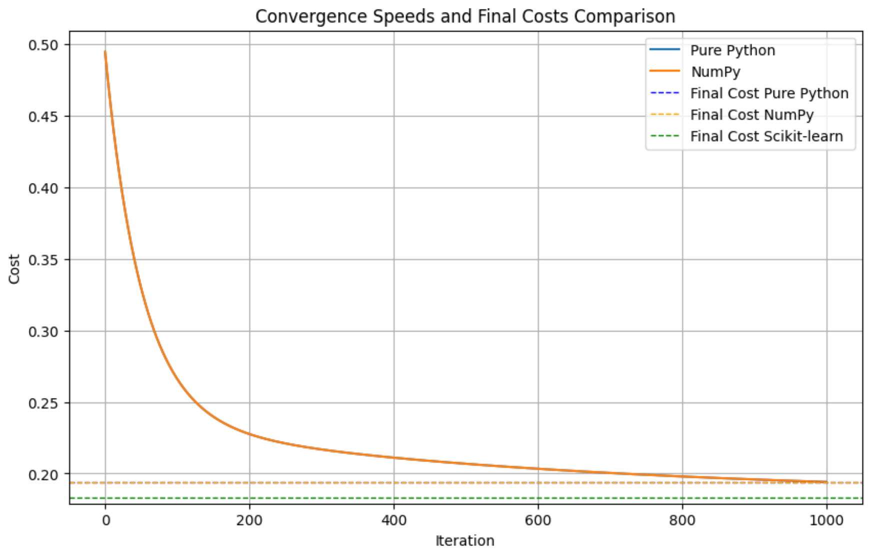


Figure 3.2: Convergence speed and final cost comparison

Chapter 4

Conclusion

The comparative analysis demonstrates that while Pure Python provides transparency for learning, NumPy and Scikit-learn are far superior in computational efficiency. The model was trained on 70% data which was the training set, and analysed on the remaining data as cross validation set to check models performance. All three approaches yield similar predictive performance, validating the correctness of the implementations. For practical applications, Scikit-learn is efficient due to its speed and ease of use.