Multivariable Linear Regression: Implementation Report

Navneet Kashyap

May 19, 2025

Analysis and Discussion

Differences and Similarities in Convergence Times and Accuracies

Table 1 summarizes the key convergence metrics obtained from the three implementations:

Method	Iterations	Time
Pure Python (RMSE metric)	919	21.83 s
Using Numpy (RMSE metric)	920	0.13 s
scikit-learn (Normal Eq.)	na	2.35 ms

Table 1: Convergence metrics comparison

All three approaches converge to similar predictive accuracies (with RMSE around 73 811.24 and MAE around 52 786.5 when validated on the test split), yet exhibit very different convergence behaviors. The pure python implementation requires an order of magnitude more time than the numpy implementation, while the normal-equation solver in scikit-learn produces exact coefficients almost instantaneously. It is also worth noting that both pure python and numpy took almost same number of iterations to converge.

Causes of Observed Differences

The primary factors behind these differences are:

- **Optimization Strategy:** The pure-Python and Numpy implementations uses batch gradient descent, updating all the weights at once until convergence criteria are met. In contrast, scikit-learn solves the normal equations analytically via a single matrix inversion. Batch gradient descent reaches the optimum solution in fewer iteration than stochastic but is slower. The scikit-learn's analytic solver on the contrast is very fast, provided the number of features is small.

- **Vectorization and Low-Level Optimizations:** Pure python mostly uses python for loops which are very slow. Numpy arrays are densely packed arrays of homogeneous type. Python lists, by contrast, are arrays of pointers to objects, even when all of them are of the same type. scikit-learn's implementation is written in C/Cython and heavily optimized, minimizing Python-level overhead.

Scalability and Efficiency Trade-offs

Iterative gradient-descent methods scale linearly with the number of training samples but require multiple passes (epochs) according to desired precision, making them slow on large datasets. They, however, are faster in scenarios where storing and inverting large matrices is infeasible, that is when the number of features is large.

The normal-equation approach has a computational complexity of $O(p^3)$ for inverting a $p \times p$ matrix (where p is the feature count), and memory complexity of $O(p^2)$. For small to moderate p, it is efficient, but becomes slow in large number of features case..

Influence of Initial Parameter Values and Learning Rates on Convergence

The choice of initial parameters and learning rate α affects gradient-descent convergence:

- **Initialization:** Starting all coefficients at zero led to symmetric behavior across all the methods and features.
- **Learning Rate:** We selected $\alpha = 0.01$. Increasing α beyond 0.05 caused oscillations and divergence in the MSE-based solver, while reducing α below 0.001 slowed the code excessively (requiring > 5000 iterations for convergence).