Analysis of Multivariable Linear Regression Approaches

Overview

This analysis compares three different implementations of multivariable linear regression applied to the California Housing dataset: a pure Python implementation using explicit loops, a vectorized version using NumPy, and the scikit-learn `LinearRegression` class. The main focus is on convergence speed, predictive accuracy, and scalability.

Model Performance and Accuracy

All three methods aim to model the relationship between housing features and median house value. The pure Python implementation yields moderate predictive accuracy, but it is the slowest to converge due to its reliance on loops and non-optimized calculations. The NumPy-based method improves both accuracy and convergence speed by leveraging efficient matrix operations. The scikit-learn implementation outperforms both in terms of accuracy and speed, since it uses advanced optimization algorithms and closed-form solutions.

Convergence Behavior

The pure Python model's cost function converges slowly and somewhat noisily because it updates parameters through explicit loops over all samples and features. The NumPy implementation shows much smoother and quicker convergence, benefiting from vectorized operations that compute gradients in bulk. The scikit-learn model does not rely on iterative optimization for this problem; instead, it directly computes the optimal weights using linear algebra techniques, so it converges essentially instantly without an iterative cost curve.

Reasons for Differences

The main factors driving differences in performance are:

- **Vectorization:** NumPy's matrix operations significantly reduce the overhead of Python loops by performing batch computations in compiled code.
- **Optimization Strategies:** scikit-learn uses analytical solutions and advanced numerical solvers such as singular value decomposition (SVD), which converge faster and more reliably than gradient descent.
- **Implementation Overhead:** The pure Python code has more overhead due to the interpreted nature of Python loops and manual summations.

Scalability and Efficiency

As datasets grow larger, the pure Python implementation quickly becomes impractical due to its slow execution. The NumPy version scales well and remains efficient because of vectorized computations. The scikit-learn implementation is production-grade and optimized for both small and large datasets, making it the most efficient and scalable option.

Effect of Initial Parameters and Learning Rate

All gradient descent implementations started with weights initialized to zero. While this can work well in many cases, poor initialization could slow down convergence or cause the algorithm to get stuck in suboptimal solutions. The learning rate, set to 0.01 in this work, plays a critical role in balancing convergence speed and stability. A learning rate too small results in slow progress, while too large may cause oscillations or divergence. scikit-learn's analytical solver does not require a learning rate.

Conclusion

Overall, the scikit-learn implementation is clearly the best choice for practical purposes due to its superior speed and accuracy. The NumPy implementation offers a good balance between learning and performance, while the pure Python version is useful primarily as an educational tool to understand the mechanics of gradient descent.