Analysis of Linear Regression Training Methods

-By Swarit Srivastava

Dataset: California Housing Prices
Target Variable: median_house_value
Features: 8 numerical + 5 one-hot encoded categorical

1. Executive Summary

We evaluated three approaches to train linear regression:

1. **Only Python**
2. **Utilizing Numpy**
3. **scikit-learn's LinearRegression**

Key Findings:

- **The Only Python approach took the largest time but also provided insights into the internal working and tuning of the model.**
- **NumPy Utilizing its features of using GPU and also better optimized for matrix operation was faster but most accurate**
- **scikit-learn provided the fastest and accuracy closer to that of the NumPy regression model.**

2. Methodology

2.1 Data Preprocessing

- Scaled numerical features: StandardScaler (mean=0, std=1)
- Scaled target: $y = (y - \mu)/\sigma$
- Train/Test Split: 80/20
- Ocean Proximity was one-hot encoded as it affects housing prices (intuition)

2.2 Implementation

- Method – I (Only Python)

  The weights are initially randomly assigned along with the bias. A function **sumofvariable** is defined in order to perform the dot product operation it returns the prediction.

  In the **training loop** for each observation in y_true the error is calculated and the weights and biases are updated accordingly in order to converge the cost function. The model is trained for 1000 epochs.

- Method – I I (NumPy)

  A similar thinking as Method-I was used but was highly optimized due to everything being a NumPy object. This caused the training time to decrease drastically. The major change was converting the training and y_true data into NumPy arrays and hence declaring weights and biases through np.random().

  The predictions were then calculated through np.dot() which was a major improvement.

- Method – I I I (Scikit-Learn)

  An even more optimized approach for linear regression with increased accuracy was by using scikit-learn. It also handled edge cases perfectly.

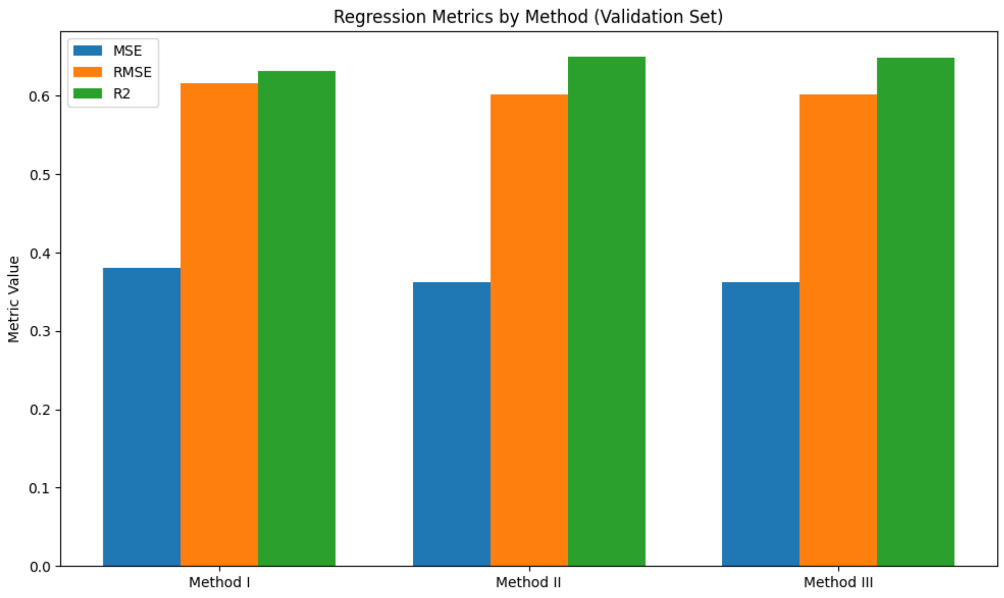
3. Results

- **Training - Set**

Methods	**Training Time (s)**	**MSE**	**RMSE**	**R^2 value**
Only Python	8980.19	0.371481	0.609493	0.6285
NumPy	0.3832	0.355396	0.596151	0.6446
Scikit-Learn	0.0028162	0.354351	0.595274	0.6456

- **Validation-Set**

Methods	**MSE**	**RMSE**	**R^2 value**
Only Python	0.38031247	0.6166947	0.6319
NumPy	0.36200726	0.6016704	0.6496
Scikit-Learn	0.36278746	0.6023184	0.6488



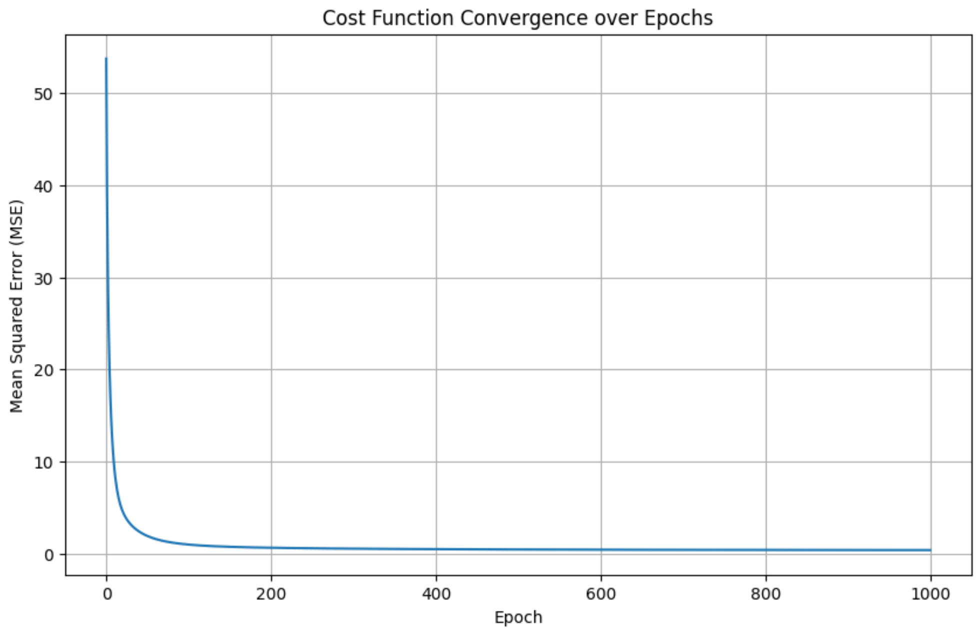
Plot Showing the Various Metrics of Performance of the methods

4. Discussions

4.1 Time analysis

- **Method-I** is the slowest taking **2 Hours, 29 Minutes and 40 Seconds** to perform **1000** iterations.

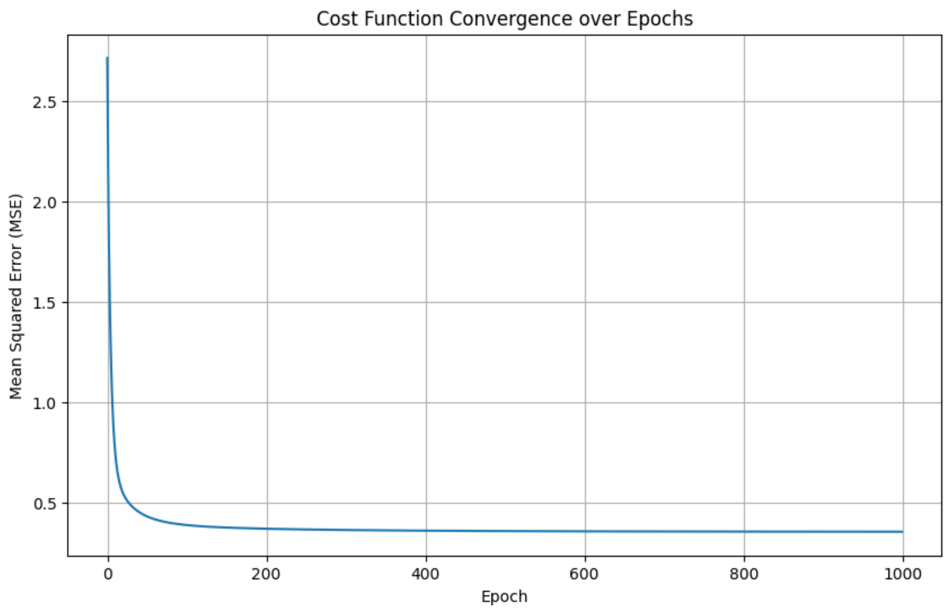
The convergence graph is as follows –



Convergence graph for Method-I (Only Python)

- **Method-II** is the faster taking **0.3832 seconds** to perform the **1000** iterations.

The convergenece graph is as follows –



Convergence Graph of Method-II(NumPy)

- **Method- III** is the fastest taking only **0.02** seconds to converge

4.3 Elucidation for the observation.

- **Vectorization Effects (Method I vs. Method II):**

   - Method I (Pure Python): This method uses standard Python loops to iterate through the data points and features to calculate gradients and update the model parameters (a and b). Python loops are relatively slow because they involve interpreting Python code for each iteration, leading to significant overhead.
   - Method II (NumPy): This method leverages NumPy's vectorized operations. Instead of iterating through individual elements, NumPy performs operations on entire arrays or matrices at once. These operations are implemented in optimized C or Fortran code, which is much faster than Python loops. For example, the calculation of np.dot(x_np, a_np) and the gradient calculations (np.dot(x_np.T, error_np)) are highly optimized in NumPy. This is the primary reason why Method II is significantly faster than Method I and converges much quicker.

- **Optimization Strategies and Solver Types (Method III):**

   - Method III (Scikit-learn LinearRegression): Scikit-learn's LinearRegression does not use gradient descent as implemented in Methods I and II by default. It typically uses more efficient analytical or numerical methods to find the optimal solution. Common approaches include:
   - Solving the Normal Equations: This is an analytical method that directly calculates the optimal weights by solving a system of linear equations. It's very fast and accurate for well-conditioned data.
   - Singular Value Decomposition (SVD): SVD is a robust numerical method that can handle multicollinearity and poorly conditioned data better than the normal equations.

4.4 Trade-offs and Scalability

1. **Method I (Pure Python/Manual Gradient Descent):**

   Scalability: Poor. This method scales very poorly with the number of features and data points. Each iteration of the gradient descent involves nested loops over the data and features, leading to a time complexity that is roughly proportional to O(iterations * n * d), where 'n' is the number of data points and 'd' is the number of features. As 'n' and 'd' increase, the execution time grows rapidly.

- **Efficiency Trade-offs:**
  - Pros: Provides a fundamental understanding of how gradient descent works. Can be useful for small datasets or for educational purposes.
  - Cons: Extremely inefficient for even moderately sized datasets due to the lack of vectorization and the overhead of Python loops. Not suitable for real-world machine learning tasks with large datasets.

2. **Method II (NumPy Gradient Descent):**

   Scalability: Better than Method I, but still has limitations. Vectorization significantly reduces the constant factor in the time complexity, but it's still an iterative method with a time complexity roughly proportional to O(iterations * d^2 + iterations * n * d) for matrix multiplications. While it's much faster than Method I, the cost of matrix operations can still become substantial as the number of features increases.

   - **Efficiency Trade-offs:**
     - Pros: Much more efficient than Method I due to vectorization. Allows for relatively faster training on moderately sized datasets. Provides a good balance between understanding the gradient descent process and achieving reasonable performance.
     - Cons: The number of iterations required for convergence can still be a limiting factor for very large datasets. The choice of learning rate can significantly impact convergence speed and the risk of not reaching the optimal solution.

3. **Method III (Scikit-learn LinearRegression):**

   Scalability: Good. Scikit-learn's LinearRegression is highly optimized and uses efficient algorithms (like solving the normal equations or SVD) that have better theoretical time complexities than iterative gradient descent. The time complexity for solving the normal equations is roughly O(d^3 + n * d^2), and SVD has a similar or slightly higher complexity depending on the implementation. While the d^3 term might seem large, the constant factor is much smaller due to optimized libraries. This makes it much more scalable than the gradient descent methods for most practical scenarios.

- **Efficiency Trade-offs:**
    - Pros: Extremely efficient and fast for finding the optimal solution. Requires no hyperparameter tuning for convergence (like the learning rate). Generally scales well to large datasets.
    - Cons: The primary limitation is memory usage. Solving the normal equations involves calculating the inverse of a d x d matrix, which can require significant memory for a very large number of features. However, for typical machine learning problems, this is usually not a major issue. For datasets with an extremely large number of features where memory is a constraint, iterative methods or specialized linear regression algorithms might be preferred.

4.5 Effect of Changing Parameters (ex. Learning Rate):

- For Method I and Method II, the learning rate is a critical hyperparameter that directly impacts convergence speed and stability. An inappropriate learning rate can lead to slow convergence, oscillation, or divergence.
- Initial parameter values can influence the convergence speed for Methods I and II but do not prevent them from finding the global minimum in this linear regression case. However, in more complex models, they are much more important.
- Method III is unaffected by initial parameter values or learning rates because it uses direct, non-iterative methods to find the optimal solution.

END OF FILE