Implement and compare three distinct approaches to multivariable linear regression, with emphasis on convergence speed and predictive accuracy.

Kushagra Yadav

May 2025

1 Introduction

This assignment details the implementation and comparative analysis of multivariable linear regression using three distinct computational approaches: pure Python, NumPy, and Scikit-learn. The primary objective is to evaluate these implementations based on their **convergence speed** and **predictive accuracy** when applied to the California Housing Price Dataset. By examining these diverse methodologies, this report aims to provide insights into the performance trade-offs inherent in building predictive models from fundamental principles to optimized library-based solutions..

2 Dataset and Data Preprocessing

The dataset was the **California Housing Price Dataset** from Kaggle, a standard benchmark for regression tasks. This dataset comprises nine features, including geographical and socioeconomic attributes, and one target variable: 'median house value'. An initial assessment of the features revealed no immediate necessity to discard any columns, as each appeared intuitively relevant to predicting housing prices.

2.1 Handling Categorical Features

The only feature that was unique was the 'Ocean Proximity', which had strings as inputs. I used the Ordinal Encoder to deal with it, which was fairly easy. An interesting fact I came to know about it was that it is best suited when the categories have an inherent order, which the category had with Near, Far, and inland as inputs.

"When the categories have a natural order, ordinal encoding is a simple yet effective method for turning categorical data into numerical representation"-**GeeksForGeeks**

2.2 Missing Value Imputation

Obviously, the next step was drop off **null values** from the dataset. I preferred to replace the data with mean values of that particular feature as it seemed appropriate.

2.3 Feature Scaling

The final preprocessing step involved **normalizing** the numerical features using **StandardScaler**. This technique transforms features to have a mean of zero and a standard deviation of one. Normalization is critical for gradient-descent-based algorithms, as it ensures that all features contribute equally to the distance calculations and prevents features with larger numerical ranges from disproportionately influencing the model's weights and the convergence speed.

3 Implementation

I used the from train_test_split from model selection in scikit to create a training and validation data for myself. I used the same preprocessing dataset in each of the three implementations to mantain consistency.

3.1 Pure Python Implementation

For the Pure Python, it was first necessary to convert the outputs we gained from the train test split as they are in the form of numpy arrays. I first created a class named LinearRegression1. [language=Python, basicstyle=]

```
class LinearRegression1:
    def __init__(self, learning_rate=0.01, iterations=1000):
        self.alpha = learning_rate
        self.iterations = iterations
        self.w = None
        self.b = 0
        self.costFunction = []
```

The ‘costFunction‘ variable was included to track the mean squared error (MSE) across each training iteration, facilitating the visualization of the model's convergence.

Initial experimentation with various learning rates (‘alpha‘) revealed significant differences in convergence behavior:

- alpha = 1: Led to overflow, indicating divergence due to an excessively large step size, causing the model's parameters to grow uncontrollably.
- alpha = 0.1 and alpha = 0.01: Both exhibited rapid convergence. However, alpha = 0.1 consistently achieved a lower final cost without signs of instability (e.g., oscillations), suggesting more efficient movement towards the optimal solution.

```
    def predict(self, X):
        new_y = []
        for i in range(len(X)):
            pred_y = self.b
            for j in range(len(X[0])):
                pred_y += self.w[j] * X[i][j]
            new_y.append(pred_y)
        return new_y
```

The 'predict' method calculates the predicted target values ($\hat{y}$) for a given set of input features (X) using the learned weights (w) and bias (b). This involves iterating through each sample and feature, summing the weighted feature values plus the bias.

The cost function that we use here is the MSE cost function: Weights (w) are randomly initialized while bias (b) is initialized to zero. The MSE cost function $J(w, b)$ is defined as the average of the squared differences between the actual and predicted values across all training examples:

$$J(w,b) = \frac{1}{2n}\sum_{i=1}^{n}(h_w(x^{(i)}) - y^{(i)})^2$$
$$= \frac{1}{2n}\sum_{i=1}^{n}((w^T x^{(i)} + b) - y^{(i)})^2$$

The source code implementation was done using basic list operations while iterating training examples.

```
    def cost_function(self, X, y):
        m = len(y)
        predictions = self.predict(X)
        cost = sum((predictions[i] - y[i]) ** 2 for i in range(m)) / (2 * m)
        return cost
```

The main Gradient Descent Algorithm takes place inside the fit function. The general update rule for any parameter θ is: ---

$$\theta_{\text{new}} = \theta_{\text{old}} - \alpha\frac{\partial J(w,b)}{\partial \theta_{\text{old}}}$$

The partial derivative of the cost function with respect to a specific weight w_j (for the j-th feature) is: ---

$$\frac{\partial J(w,b)}{\partial w_j} = \frac{1}{n}\sum_{i=1}^{n}(h_w(x^{(i)}) - y^{(i)})x_j^{(i)}$$

--- Where $x_j^{(i)}$ is the value of the j-th feature for the i-th training example. The partial derivative of the cost function with respect to the bias b is: ---

$$\frac{\partial J(w,b)}{\partial b} = \frac{1}{n}\sum_{i=1}^{n}(h_w(x^{(i)}) - y^{(i)})$$

Both are implemented by iterating over the list.

```
    def cost_derivative(self,y_predict,y,X,j):
        m=len(y)
        sum=0
        for i in range(m):
            sum+=(y_predict[i]-y[i])*X[i][j]
        return sum/m

    def b_derivative(self,y_predict,y):
        m=len(y)
        sum=0
        for i in range(m):
            sum+=(y_predict[i]-y[i])
        return sum/m

    def fit(self,X,y):
        m = len(y)
        n = len(X[0])
        self.w = [random.uniform(-0.01, 0.01) for _ in range(n)]
        self.b = 0
        for iteration in range(self.iterations):
            y_predict=self.predict(X)
            for i in range(n):
                self.w[i]-=self.alpha*self.cost_derivative(y_predict,y,X,i)
            self.b -= self.alpha * self.b_derivative(y_predict, y)
            self.costFunction.append(self.cost_function(X, y))
```

The pure python implementation works fine but more time to run.

3.2 Numpy Implementation

Since, the outputs from train test split are in the form of numpy arrays, the could directly be used. There are no changes in the logic that goes behind the Numpy Implementation. The big difference that occurs is the optimization that occurs due to the numpy operations. NumPy, performs operations on entire arrays (vectors and matrices) at once. These vectorized operations are implemented in highly optimized.

```
import numpy as np
class LinearRegression2:
    def __init__(self, learning_rate=0.1, iterations=1000):
        self.alpha = learning_rate
        self.iterations = iterations
        self.w = None
        self.b = 0
        self.costFunction=[]

    def predict(self, X):
        return np.dot(X, self.w) + self.b

    def cost_function(self, X, y):
        m = len(y)
```

```
        predictions = self.predict(X)
        cost = sum((predictions - y) ** 2) / (2 * m)
        return cost

    def cost_derivative(self, y_predict, y, X):
        m = len(y)
        return np.dot(X.T, (y_predict - y)) / m

    def b_derivative(self, y_predict, y):
        m = len(y)
        return np.sum(y_predict - y) / m

    def fit(self, X, y):
        X = np.array(X)
        y = np.array(y)

        m, n = X.shape
        self.w = np.zeros(n)
        self.b = 0

        for iteration in range(self.iterations):
            y_predict = self.predict(X)
            self.w -= self.alpha * self.cost_derivative(y_predict, y, X)
            self.b -= self.alpha * self.b_derivative(y_predict, y)
            self.costFunction.append(self.cost_function(X, y))
```

Parallel operations take place on the whole array at once, leading to lesser time.

3.3 Scikit-learn Implementation

The scikit learn implementation uses the Linear Regression class from the Scikit-learn library. The big difference here is that the LinearRegression uses an analytical solution known as Ordinary Least Squares (OLS). This method directly calculates the optimal weights and bias using linear algebra leading to lower convergence times than the pure python or numpy implementation:

```
from sklearn.linear_model import LinearRegression
model3 = LinearRegression()

start_time = time.time()
model3.fit(X_train_scaled, y_train)
end_time = time.time()

sk_time=end_time-start_time
predicted_y3 = model3.predict(X_val_scaled)
```

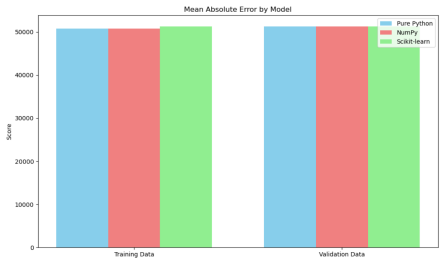

Figure 1: Mean Absolute Error by Model

4 Analysis

4.1 Time

The time taken be each implementation is listed below: Pure Python Implementation: 29.84807276725769 Numpy Implementation: 1.095057487487793 Scikit Implementation: 0.008042335510253906
Pure Python Implementation: It is consistently the slowest. This is due to the use of explicit Python loops for every mathematical operation. Each operation is handled individually by the python interpretor.
Numpy Implementation: Numpy is much more faster as it performs operations on entire arrays at once. These are highly optimized vectorized operations. Thus, the primary reason for the vast speed difference is vectorization.
Scikit Implementation: It took the least time to implement as it doesn't use Gradient Descent Algorithm, rather it uses the Ordinary Least Squares. It uses linear algebra directly, so there is no iteration as such.

4.2 Accuracy on Validation Data

Pure Python Analysis
MAE: 51268.8201890212
RMSE: 70944.41950907093
R2 Score 0.6229190802058171
Numpy Implementation Analysis
MAE: 51268.82018995449
RMSE: 70944.41950838693
R2 Score 0.6229190802130884
Scikit Implementation Analysis
MAE: 51276.3409486691
RMSE: 70942.84955503102
R2 Score 0.6229357691340576

4.2.1 Graphs

The metrics gives by the Pure Python and Numpy Implementation match it other quite perfectly. This is because both apply fundamentally the same Gradient Descent algorithm just uses different implementations.

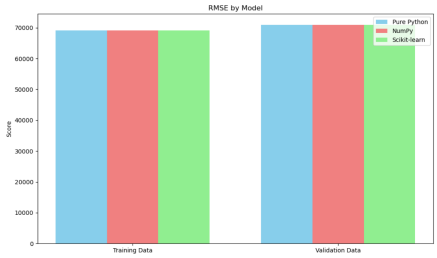

Figure 2: RMSE by Model

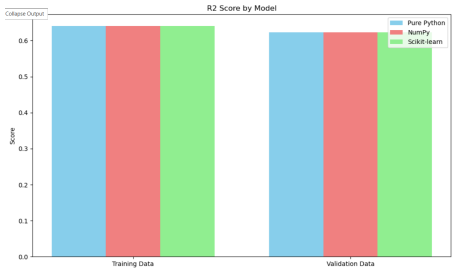

Figure 3: R2 score by Model

Predictions by Scikit Implementation also seem to be near as it uses a different algorithm.

4.3 Scalibility and Efficiency

Pure Python: It is not very scalable. The training time increases with increases in the number of samples (m) and features (n). It is highly inefficient for larger datasets because of large number of operations.
Numpy: It is more scalable than the Pure Python model because vectorized operations are much faster per step. It can handle larger datasets but since it uses the same iterative algorithm it will still be slower than direct methods.
Scikit-learn: Excellent for typical machine learning tasks. Its highly optimized and scale very well with the number of features. Directly computing the solution is the most computationally efficient approach.

4.4 Influence of Initial Parameter Values and Learning Rates on Convergence

Influence of Initial Parameter values: The Mean Squared Error cost function for linear regression is a convex function, thus it has only one global minimum. Thus, inital parameters does not affect the minimum that gradient descent converges to.
They primarily lead to different initial costs, though both of the paths leads to the same optimal solution.

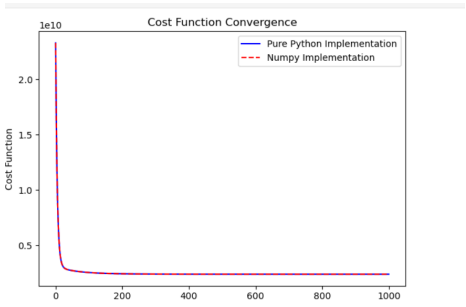

Figure 4: Cost Function: LR=0.1

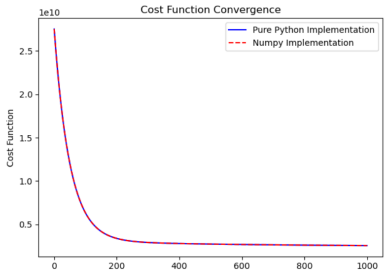

Figure 5: Cost Function: LR=0.01

Influence of the Learning rates: The learning rate controls the step size taken in direction of negative gradient in the gradient descent algorithm. It is very crucial for effective convergence. Observed Effects from the Plots: lr=0.1 (First Plot): This higher learning rate resulted in a lower final converged cost compared to lr=0.01. This suggests that lr=0.1 was more efficient at reaching a better minimum. It also showed rapid and stable convergence.. lr=0.01 (Second Plot): While also leading to stable convergence, it converged to a slightly higher final cost . This implies it either converged to a slightly less optimal minimum or required more iterations to reach the same minimum as lr=0.1.(Though this one seem unlikely)
For lr=0.1, the cost function overshooted indicating a very large learning rate.