Comparative Study of Multiple Linear Regression Implementations

Hardik Prabhakar

24095038

Introduction

Linear regression is one of the simplest machine learning algorithms that has been studied extensively and implemented in various ways. In this report, we will study three different implementations of the algorithm and compare them.

The three implementations that we will use are:

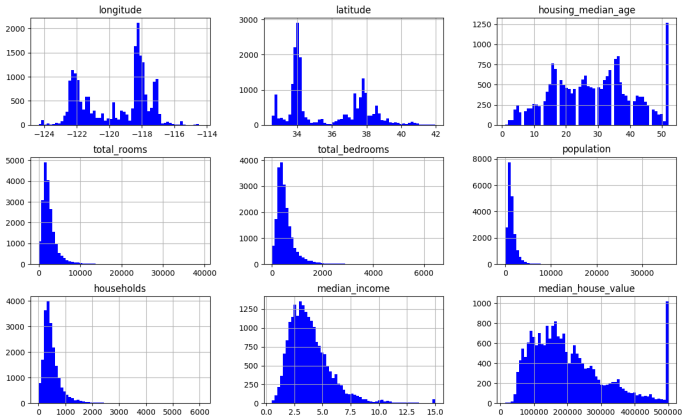
1. Pure Python Implementation
2. Optimized Numpy Implementation
3. Scikit-learn Implementation

The dataset we are using is the California Housing Prices Dataset, predicting the variable `median_house_value`.

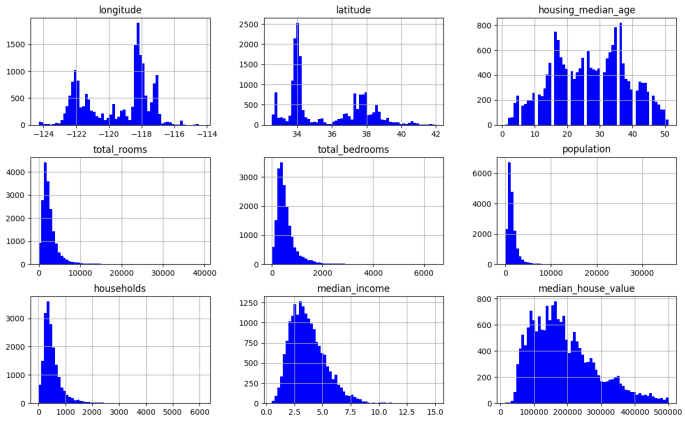
Data Pre-Processing

- We first load the dataset using pandas and check the total number of records and the number of null records. They come out to be 20640 and 207 respectively. As the number of null records is very small compared to the total, we will drop them.

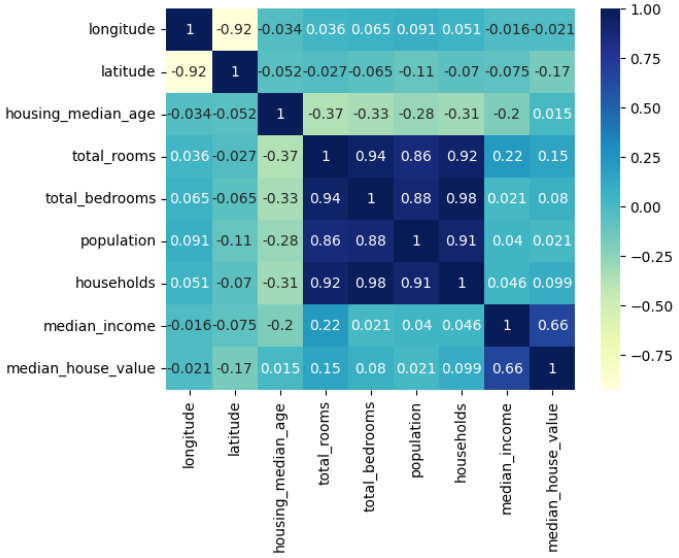
- We then visualize the dataset using an histogram.



- As we can clearly see from the graphs, there seems to be a outlier in each `housing_median_age` and `median_house_value` fields. These outlier values occur at the maximum value of the respective fields, which suggests that these values were likely out of range. As these values can introduce inaccuracy to our model, we will remove them.



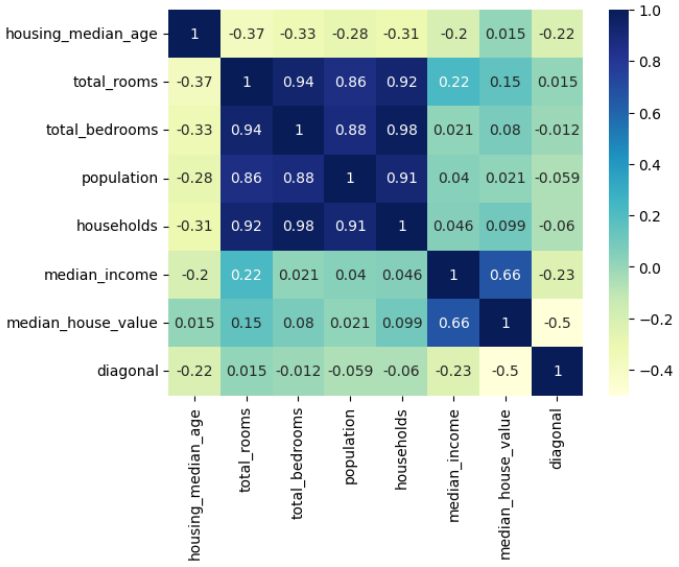
- We now plot a correlation heatmap from the dataset. It visualizes how each value changes with change in some other value.



- From the heatmap, we can see that `median_income` has the highest correlation coefficient with respect to `median_house_value`, whereas `population` and `longitude` have the lowest coefficients.

- To increase the significance of `longitude` field, we will combine it with the `latitude` field, forming a new field that we shall call `diagonal`. This field will represent the diagonals on which the houses lie, which would signify their proximity to the sea.

- The new heatmap after modifying the fields will be as follows :



- Now, we can clearly see that the `diagonal` field is highly correlated with our target variable. Due to this field being sufficient to represent the location of the houses, we won't need to use the `ocean_proximity` field.

- We split our dataset into target and feature variables, with `median_house_value` being the target and the remaining fields being the features.
  We further split these datasets into training and testing sets, using a 70/30 split, to ensure that overfitting does not occur in our model.

- We scale our features and target using *Z-Score Normalization*, to ensure that gradient descent proceeds uniformly and the cost function doesn't grow out of bounds. The scaling is done according to the formula:

$$\frac{x - \mu}{\sigma}$$

  where μ = mean and σ = standard deviation

- Finally, we initialize the weight and bias matrices populated with values uniformly distributed between -1 and 1.

Implementation

Now that we have preprocessed our dataset, the next step is to develop a linear model that can predict house prices with relatively high precision.

The linear model is given as :

$$f_{\mathbf{w},b}(\mathbf{x}) = w_0x_0 + w_1x_1 + ... + w_{n-1}x_{n-1} + b$$

or in vector notation:

$$f_{\mathbf{w},b}(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b$$

where **w** represents the weights associated with the model, **b** the bias and **x** the features. The model makes prediction of house prices when the features are fed into it.

To evaluate the accuracy of the predictions, we use a mathematical function called the **cost function** which calculates the sum of the squares of differences between the predicted and expected output divided by twice the number of training examples. It is given as :

$$J(\mathbf{w}, b) = \frac{1}{2m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})^2$$

Our goal is to minimize the cost function by varying the weights and bias of the model. To do this, we use **gradient descent**, an optimization algorithm that is commonly used to train machine learning models, much like this one.

The algorithm works by calculating the cost function gradient with respect to weights and bias and then taking small steps in its opposite direction. This process repeats until the cost function converges to a minimum. Mathematically, this is represented as :

$$w_j = w_j - \alpha \frac{\partial J(\mathbf{w}, b)}{\partial w_j} \quad \text{for j = 0..n-1}$$

$$b = b - \alpha \frac{\partial J(\mathbf{w}, b)}{\partial b}$$

where, n is the number of features and α is the learning rate

The gradients are given as :

$$\frac{\partial J(\mathbf{w}, b)}{\partial w_j} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)}$$

$$\frac{\partial J(\mathbf{w}, b)}{\partial b} = \frac{1}{m} \sum_{i=0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})$$

The learning rate α is chosen such that cost function doesn't diverge or doesn't converge too slowly. This was determined by trial and error in our implementation.

We implemented the above algorithm in two ways. First, by using only core Python features, without utilizing any external libraries. Then we implemented it using an external library named *NumPy*, which would speed up the necessary calculations

Finally, we implemented linear regression through the `LinearRegression` class from the `scikit-learn` library. This model uses **Ordinary Least Squares Regression** which calculates the optimum weights and biases with the formula:

$$\hat{\boldsymbol{\beta}} = \left(\mathbf{X}^{\mathrm{T}}\mathbf{X}\right)^{-1} \mathbf{X}^{\mathrm{T}}\mathbf{y}$$

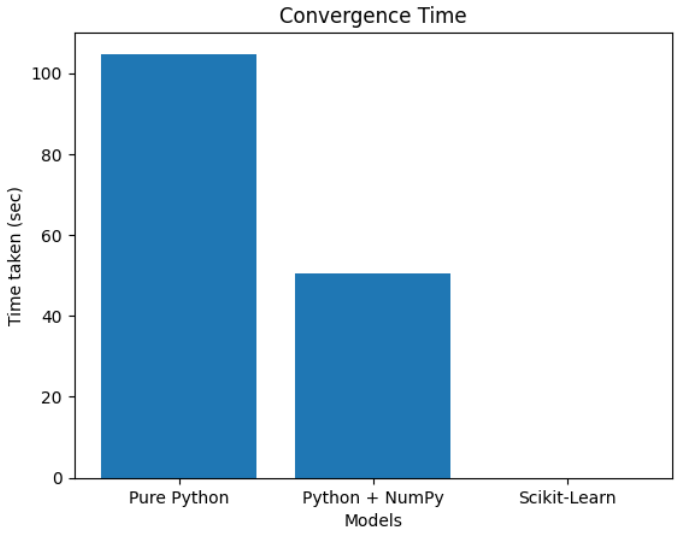
where $\mathbf{X}$ is the feature matrix and $\mathbf{y}$ is the target matrix

Evaluation

After implementing multiple linear regression in three different ways, we evaluate the effectiveness of our linear model by various criteria.

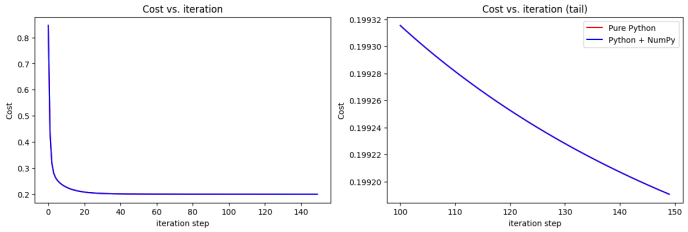
Convergence Time

We measured the time taken by each of the three implementations to fit the training set and return the optimal parameters. Plotting them in the form of a bar chart, we get :



From the figure, we can clearly see that `scikit-learn`'s `LinearRegression` class is vastly superior to the first two parts in terms of fitting duration. Between the two parts, the NumPy implementation has a significantly shorter fitting duration.

The **Cost vs Iteration** graph for the 1st and 2nd implementations exactly overlap each other. This signifies that the only difference between the two implementations is the speed with which the calculations are being done.

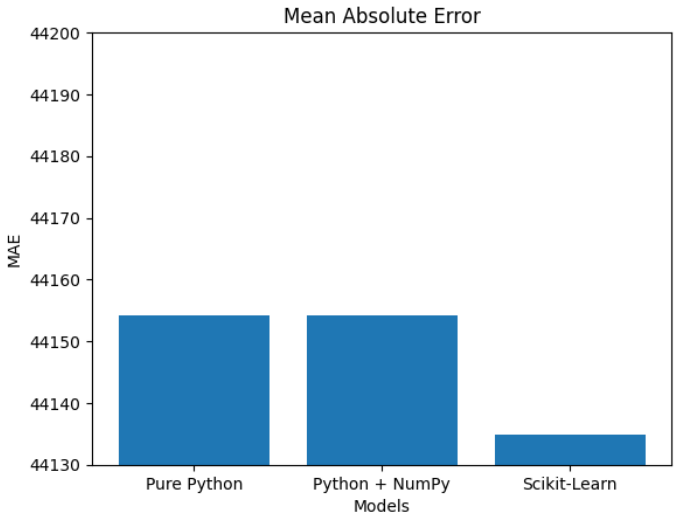


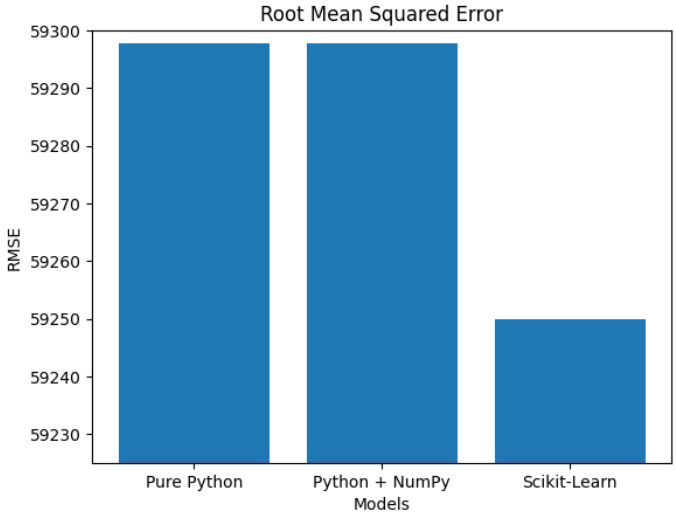
Performance Metrics

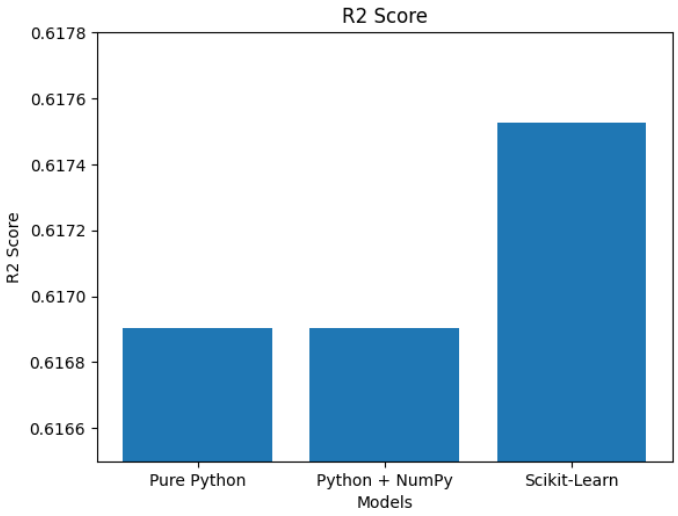
To evaluate our linear regression model, we will use three different regression metrics. They are :

1. Mean Absolute Error (MAE)
2. Root Mean Squared Error (RMSE)
3. R-squared (R^2 score)

Plotting their values for the three models on a bar chart, measured on the validation set, we get :







As we can see from the graphs, the `scikit-learn` model has comparatively better performance than the other two models. But the difference is so minuscule that there would be no discernible difference in the accuracy of their predictions.

Analysis & Discussion

- After analyzing the different evaluation criteria, we can conclude that `scikit-learn`'s `LinearRegression` model is the best one as it is orders of magnitude faster than the two other implementations and gives slightly more accurate predictions.
- The reason for the dominance of `scikit-learn`'s model is the optimization technique it uses, which is **Ordinary Least Squares Regression**. It provides a closed form solution for the optimal weights and bias, consisting of only matrix operations which are speed up due to parallelization through **SIMD**(Single Instruction, Multiple Data) processes. They utilize the computer's hardware to perform multiple similar types of calculations at the same time, thus making it more time-efficient.
- This parallelization is also present in the NumPy implementation. All the matrix operations like multiplication and addition are sped up by using SIMD processes, leading to a faster convergence time. This is shown by the difference in convergence time between the Pure Python and NumPy implementation, showing approximately 50% decrease, which are otherwise logically similar.
- Thus on larger datasets, the NumPy and `scikit-learn` implementation scale way better.
- Picking an appropriate learning rate is a crucial step. If it's too large, it would lead to the cost function diverging. If it's too small, convergence would take more time.
- The initial parameter values don't influence the convergence of the cost function. This is because our cost function is a convex function, which guarantees that we'll reach the global minima if we follow the gradient.