CSOC IG Prerequisites: Final Report

Anant Saxena
Roll Number: 24054002

May 19, 2025

Objective

Brief overview of the task: implementing and comparing three approaches to multivariable linear regression — Pure Python, NumPy, and scikit-learn.

Dataset Description

The Dataset of California Housing Prices from kaggle was given to us. It can be found at the following link: `https://www.kaggle.com/datasets/camnugent/california-housing-prices`
We were required to train a model with linear regression using different features of houses to predict the prices of houses. The useful features in the Dataset are

['housing_median_age', 'total_rooms', 'total_bedrooms',
'median_income', 'households', 'ocean_ proximity']

The target chosen for training is 'median_house_value'.

Implementation Details

Part 1: Pure Python Implementation

One of the core algorithms of Machine Learning which is used very widely is **Gradient Descent**. Suppose the price of the houses is just dependent on a single feature, e.g. Area. Then we can define a single relation between the feature - area (x) and the target - price (y) such that,

$$y = w_{ideal}x + b_{ideal}$$

where w is called weight of the feature - area,
and b is the bias.

Generally, we can predict the price $\hat{y}$ for the feature x using a similar relation,

$$\hat{y} = wx + b$$

The prediction can be done by trying to approximate the values of w and b very close to w_{ideal} and b_{ideal} .

The Gradient descent algorithm is an optimization technique which tends to do the same, by trying to minimize the error between the target value and predicted value.

Before we try to understand how Gradient descent alogrithm operates, we first need to define what is called a Cost Function (J). For a dataset with m no. of entries, cost function can be defined as

$$J = \frac{1}{2m}\sum_{i=1}^{m}(\hat{y}^i - y^i)^2$$

We can observe that J is a analogous to the Mean Square Error of the model. This is used in the gradient descent algorithm to change the values of weight and bias repeatedly in terms with following equation

$$w^{new} = w^{old} - \alpha\frac{\partial J}{\partial x}$$
$$b^{new} = b^{old} - \alpha\frac{\partial J}{\partial b}$$

Upon finding partial derivative, the following equations can be written as

$$w^{new} = w^{old} - \frac{\alpha}{m}\sum_{i=1}^{m}(\hat{y}^i - y^i)x^i$$

$$b^{new} = b^{old} - \frac{\alpha}{m}\sum_{i=1}^{m}(\hat{y}^i - y^i)$$

where α is known as learning rate and is a small constant of our choice.

The equation is iterated thousands of times over large chunk of data, and the values of J gets smaller with each iteration, thus reducing the error between predicted values $\hat{y}$ and target values y.

To train a model to predict $\hat{y}$ for database with many features, i.e. $x_1, x_2, ..., x_n$ we can use the following relation

$$\hat{y} = w_1x_1 + w_2x_2 + ... + w_nx_n + b$$

The Gradient descent algorithm will remain same for each weight w corresponding to each feature x. Thus for n number of features, any weight w^j of an intermediate feature x^j can be modified as

$$w_j^{new} = w_j^{old} - \frac{\alpha}{m}\sum_{i=1}^{m}(\hat{y}^i - y^i)x_j^i$$

The equation for change in bias (b) will remain same.

The algorithm does not run infinitely and can be stopped by either defining a minimum threshold value for cost function (J), or by visually plotting value of (J) and feeding sufficient lengths of datasets to reach near the convergence.

The above mathematical operations can be implemented in python using simpler loops and functions without the use of any libraries, but they will lead to poor runtime and memory complexity.

Part 2: NumPy Implementation

With access to Numpy library of python, we get a huge benefit due to concept of vectorization.
The multiple features $x_1, x_2, ..., x_n$ can be represented by a single variable $\vec{x}$ such that

$$\vec{x} = x_1\hat{i_1} + x_2\hat{i_2}.... + x_n\hat{i_n}$$

where $\hat{i_1}, \hat{i_2}, ..., \hat{i_n}$ corresponds to orthogonal directions in n-dimensions.

In form of a $1D$ array, the features can be expressed as

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ x_n \end{bmatrix}$$

The corresponding weights can be expressed as

$$\vec{w} = \begin{bmatrix} w_1 \\ w_2 \\ . \\ . \\ w_n \end{bmatrix}$$

The predicted value $\hat{y}$ can be expressed in terms of dot$(\cdot)$ product of $\vec{x}$ and $\vec{w}$ as

$$\hat{y} = \vec{w} \cdot \vec{x} + b$$

$$\hat{y} = \begin{bmatrix} w_1 \\ w_2 \\ . \\ . \\ w_n \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ x_n \end{bmatrix} + b$$

The advantage in using vector dot product with Numpy over linear equation with building manual loops is in way the compiler performs operations and store values. Numpy dot product (np.dot) supports parallelized operations and thus, utilizes the GPU of the computer rather than just using the CPU. Due to parallelized dot multiplication of the matrices, the operations end very soon due to the performance of many calculations at the same time.
Also due to decrease in looping elements in the body of code, the number of extra variables decreases. This ensures efficient memory utilization.

Part 3: scikit-learn Implementation

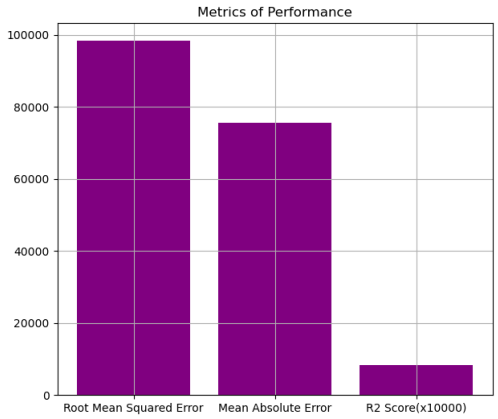
Scikit-learn library's linear regressor function operates on **Ordinary Least Square (OLS)** linear regression. It is different from gradient descent linear regression and does not involve using any weights(w) and bias(b).
It tries to minimize Mean Square Error using an equation called the **Normal Equation**. It performs operations on the complete database, treating it as a $2D$-matrix, in a single step and does not involve any looping elements. The operation is very fast and accurate.

Convergence and Timing

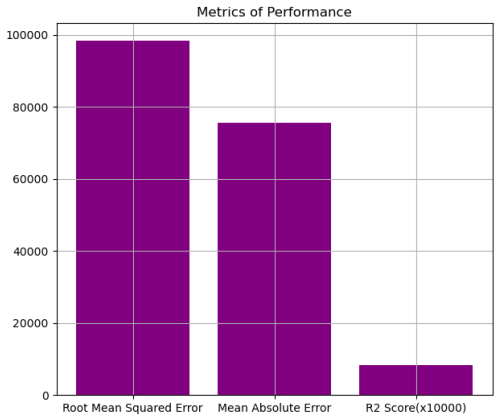
Initialization Parameters: $\vec{w}_{init} = \begin{bmatrix} 0.4 \\ 0.4 \\ 0.4 \\ 0.4 \\ 0.4 \\ 0.4 \end{bmatrix}$, $b = 0$

Part 1: Time taken to converge = 352.01 seconds
Part 2: Time taken to converge = 117.32 seconds
Part 3: Time taken to converge = 0.02 seconds

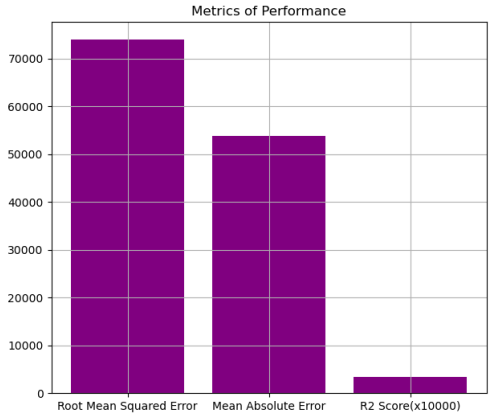
Performance Metrics



Part1
RMSE : 98375.98
MAE : 75559.75
R2 Score : 0.819



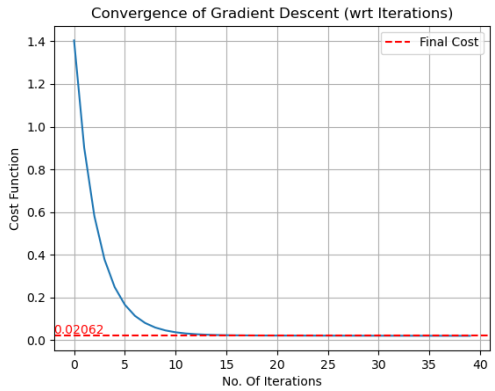
Part2
RMSE : 98375.98
MAE : 75559.75
R2 Score : 0.819



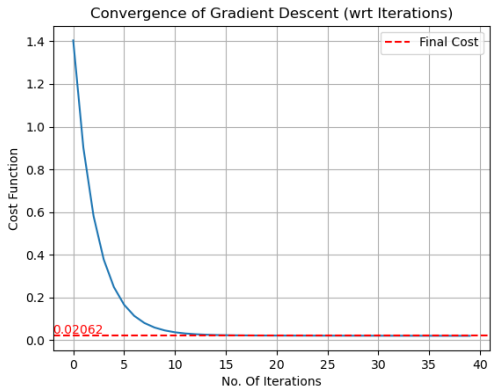
Part3
RMSE : 73980.34
MAE : 53754.69
R2 Score : 0.330

Figure 1: RMSE v/s MAE v/s R2

Visualization

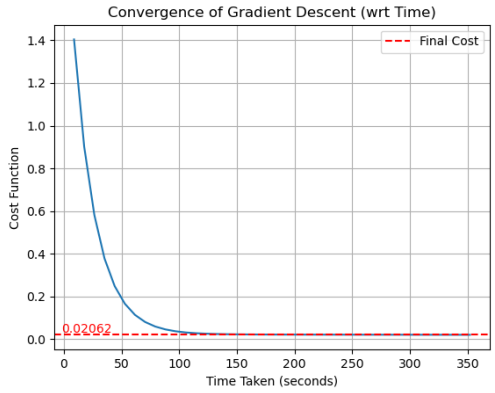


Part-1

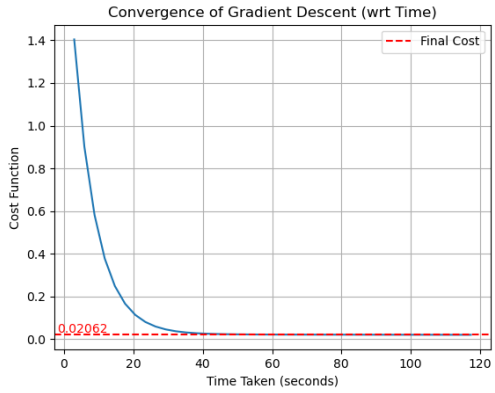


Part-2

Figure 2: Cost Function v/s Iterations



Part-1



Part-2

Figure 3: Cost Function v/s Time (seconds)

Analysis and Discussion

- Differences and Similarities in convergence times and accuracies
  - Part-1 , Part-2 and Part-3 had different convergence rates . Part-2 convergence rate was comparatively faster due to parallelized vector operations by Numpy. Part-3 had the fastest convergence rate due to application of OLS method rather than gradient descent.

- Scalability and Efficiency trade-offs
  - Part-1 code is the least Efficient in terms of memory and model training time, as it involves using gradient descent algorithm and linear equations which lead to involving multiple loops and variables in code ultimately leading to very large training durations. It is only suitable for very small scale model training.
  - Part-2 code is better than Part-1, as it involves using gradient descent algorithm with vectorized operations from Numpy library. Using arrays help in avoiding inefficient loops and extra variables. It is suitable for moderately small scale model training.
  - Part-3 code is the best out of all the codes in the report. It involves using OLS method rather than gradient descent. OLS method uses Normal Equation to train the model, which involves treating whole database as a single matrix and direct single line operations are applied to it. This leads to most efficient and scalable training algorithm.

- Data Preprocessing
  - For Gradient descent algorithm (Part-1 and Part-2), rescaling the features leads to quicker convergence rate. As the data is reduced by a large factor and is pushed to negative side of number line during rescaling, the calculations become easy to handle and also the numbers start cancelling each other preventing overflow error.
  - Part-3 code's Linear regression model uses OLS training algorithm, which is unaffected by preprocessing of data.

- Initial Parameters and learning Coefficient
  - Part-1 and Part-2 are affected by both initial parameters (w_{init}, b_{init}) and learning coefficient (α). This can be seen from the above observations that by having same initial parameters and learning coefficient, the Convergence point and Performance metrics of Part-1 and 2 were identical. Starting from different intial parameters would have resulted in different final weights and bias, and thus different predicted outcomes.
    Adjusting learning rate is very important. In case of large or wrong learning rate,

the algorithm will fail to converge, and the Cost function (J) will begin to increase rather than decrease. Very small learning rate will result in slower convergence rate. Thus an optimal value in middle of both is to be tested and opted.

– Part-3 code's Linear regression uses Normal Equation to train model and is independent of initial parameters (w_{init}, b_{init}). It does not involve use of Learning coefficient (α).

End of File