Comparative Study of Multi-variable Linear Regression Implementations

`Saikat Ghorai, Dept. Of Computer Science and Engineering`
`+91-8264760801, saikat.ghorai.cd.cse24@itbhu.ac.in`

1 Introduction

This document serves as my report for the submission to COPS IG CSoC '25 's Prerequisite assignment. Through this document, I aim to summarize and outline my methodology in analyzing the California Housing Prices dataset using a **Multi-variate Linear Regressional model**. As instructed in the CSoC Prequisites guide, I have ensured that my assignment is divided into 3 parts namely:

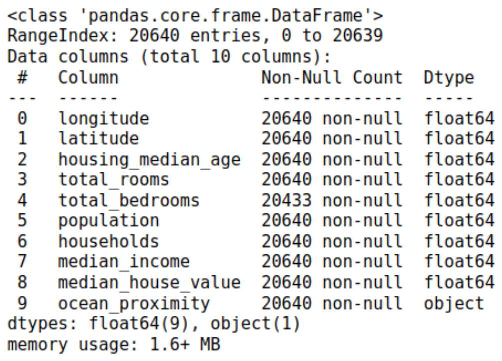
1. Pure Python implementation
2. `NumPy` implementation
3. `scikit-learn` implementation using the `LinearRegression` class

The Pure Python and NumPy implementation use the Batch Gradient descent to optimize the parameters present in the model. The `LinearRegression` class, however, does not use such methods and instead opts to directly solve for the best value of the parameters using Ordinary Least Squares.

2 Dataset and Preprocessing

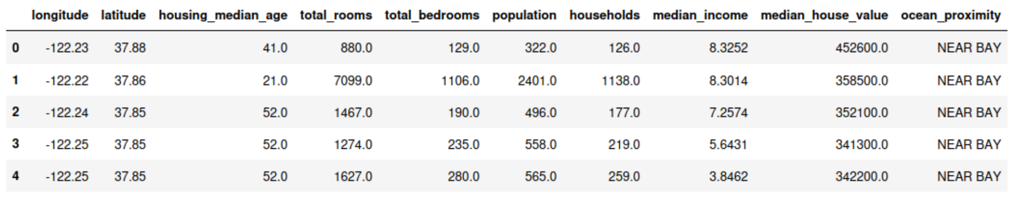
2.1 Dataset Loading

The aforementioned California Housing Prices dataset gives us various information about the 1990 California census.
The dataset contains 9 numerical features and 1 categorical feature.
A snapshot of the data is provided after suitable loading into `Pandas`.



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   longitude           20640 non-null  float64
 1   latitude            20640 non-null  float64
 2   housing_median_age  20640 non-null  float64
 3   total_rooms         20640 non-null  float64
 4   total_bedrooms      20433 non-null  float64
 5   population          20640 non-null  float64
 6   households          20640 non-null  float64
 7   median_income       20640 non-null  float64
 8   median_house_value  20640 non-null  float64
 9   ocean_proximity     20640 non-null  object 
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

Figure 1: Information generated using `df.info()`



	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.0	8.3252	452600.0	NEAR BAY
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	8.3014	358500.0	NEAR BAY
2	-122.24	37.85	52.0	1467.0	190.0	496.0	177.0	7.2574	352100.0	NEAR BAY
3	-122.25	37.85	52.0	1274.0	235.0	558.0	219.0	5.6431	341300.0	NEAR BAY
4	-122.25	37.85	52.0	1627.0	280.0	565.0	259.0	3.8462	342200.0	NEAR BAY

Figure 2: The first five rows of the dataset

2.2 Preprocessing

From Figure 1, it is clearly evident that there are some NULL values present in the `total_bedrooms` column. This was addressed by filling in those empty gaps with the mean of all the values present in that specific column.

Further, the categorical variable `ocean_proximity` was encoded into numerical type using the following sequence.

Category	Encoded into
NEAR BAY	0
1H OCEAN	1
INLAND	2
NEAR OCEAN	3
ISLAND	4

Table 1: Encoded form of the categorical variable

Furthermore, all the data except `ocean_proximity` was Z-scale normalized.

3 Exploratory Data Analysis

To get a bird's eye view of how pieces of the data might be related to each other, I generated pairplots and a heatmap of the correlation matrix using `seaborn`.

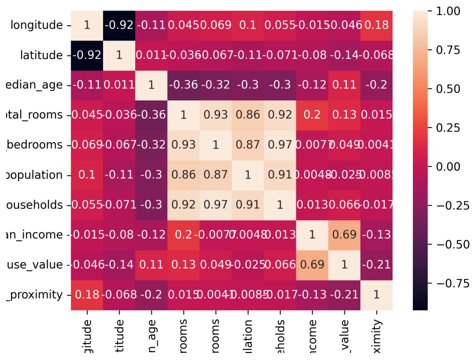


Figure 3: Correlation HeatMap

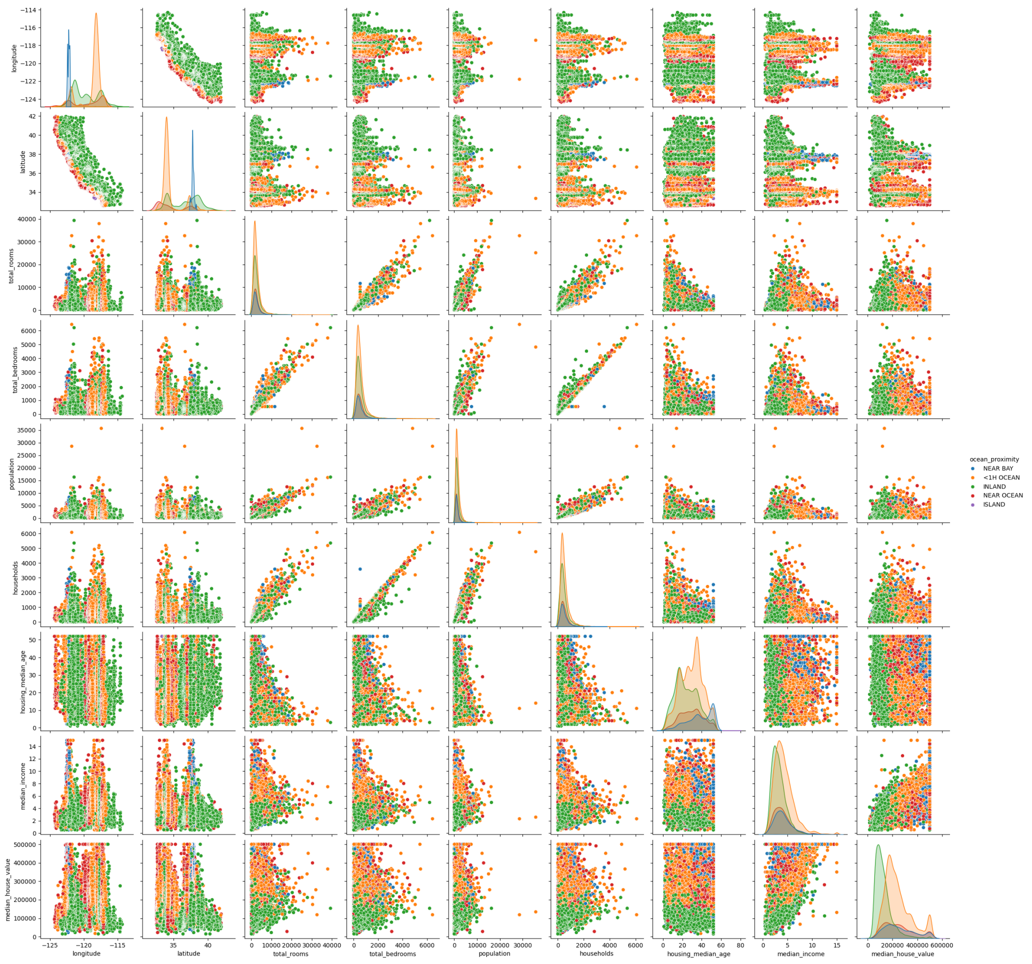


Figure 4: Pairplot of the data

Interesting conclusions can be drawn from such preliminary inspection itself. Many features of the data are not at all correlated with the `median_house_value` label. Only the `median_income` feature shows a promising linear correlation that is to be mapped by the model.

4 Method

I consider the problem of learning a multi-variate linear regression model over the processed dataset to accurately predict `median_house_value` or median housing value. To load the dataset and perform preprocessing on it, `Pandas` library has been used throughout the three different approaches. I have set the value of the learning rate to be a constant 0.01 and the initial value of θ to be a null vector.

4.1 Pure Python Implementation

The Pure Python implementation utilizes manual for & while-loops that are clubbed within functions to produce the hypothesis function $h(\theta, \mathbf{X})$, $cost(\theta, \mathbf{X}, \mathbf{Y})$, $update\_theta\_j(\theta, \mathbf{X}, \mathbf{Y}, j)$ and the $update\_theta(\theta, \mathbf{X}, \mathbf{Y})$ functions.
These functions calculate each necessary and individual component of the linear regression model. The pseudocode for the various functions are given:

```
\* The hypothesis function *\
h(theta, X):
    set partial to 0.0
    insert 1 at the beginning of X to produce biased feature vector
    for i := 1 to NUMBER OF FEATURES:
        increase partial by theta[i] * X[i]
    return partial

\* Uses Squared Error as the cost function *\
cost(theta, X, Y):
    set result to 0.0
    set m to Y.size
    for each x belongs to X:
        increase res by (h(theta, x) - Y[corresponding]) ** 2
    return (1 / (2 * m)) * result

\* Update each individual parameter indexed by j *\
update_theta_j(theta, X, Y, j):
    set m to Y.size
    set res to 0.0
    for i := 1 to m:
        set xi as biased X
        increase result by xi[j] * (h(THETA, X[i]) - Y[i])
    res := res * (LEARNING_RATE / m)
    return res

\* Update the entire theta vector *\
update_theta(theta, X, Y):
    set theta_old as theta.copy()
    set n to theta.length
    for i := 1 to n:
        decrease theta[i] by update_theta_j(theta_old, X, Y, i)
    return theta
```

4.2 `NumPy` implementation

Much of the code structure used in the Python implementation has been preserved. However, I have replaced the for & while loops with vectorized NumPy functions. The length of each function, as a result, has decreased considerably.

4.3 Using the `LinearRegression` class from scikit-learn

Much of the heavylifting has been handled by the `LinearRegression` class itself. The dataset was trained using

```
LR.fit(X, Y)
```

4.4 Training the model using Pure Python and `NumPy` implementations

I used Batch Gradient Descent to train the models. A fairly abridged version of the training algorithm is presented here:

```
Repeat until convergence {
    THETA := update_theta(THETA, X, Y)
}
```

5 Experiments

Various running parameters of the model were duly stored and processed to obtain useful information about the model's working. The time taken for each epoch of the model was noted and how the MAE, RMSE and R^2 values evolve over time was also saved.

5.1 Time taken and Value of Cost Function

It was seen that for both the Pure Python and `NumPy` implementations, the time taken for a certain number of epochs was linear to the number of epochs as expected. For the cost function, it is clearly evident that the `NumPy` implementation converged faster as compared to the Pure Python implementation.

	Time taken (s)	**Value of `cost()`**	R^2	**RMSE**	**MAE**
Python	139.55	0.274	0.453	0.741	1.087
NumPy	100.18	0.274	0.453	0.741	1.087

Table 2: Values after 100 iterations

	Time taken (s)	**Value of `cost()`**	R^2	**RMSE**	**MAE**
Python	630.37	0.226	0.549	0.672	0.965
NumPy	501.71	0.226	0.549	0.672	0.965

Table 3: Values after 500 iterations

	Time taken (s)	**Value of `cost()`**	R^2	**RMSE**	**MAE**
Python	936.15	0.222	0.557	0.667	0.957
NumPy	751.47	0.222	0.557	0.667	0.957

Table 4: Values after 750 iterations

	Time taken (s)	**Value of `cost()`**	R^2	**RMSE**	**MAE**
Python	1241.06	0.220	0.562	0.663	0.951
NumPy	1004.5	0.220	0.562	0.663	0.951

Table 5: Values after 1000 iterations

5.2 Information Gathered from scikit-learn's `LinearRegression` object

scikit-learn's `LinearRegression` fit the data quite quickly in about 0.0027 seconds due to the fact that it does not perform gradient descent at all. Instead, it uses Ordinary Least Squares method and fast matrix multiplication to obtain the optimal value in one go.
The various values obtained are tabulated.

Time taken (s)	R^2	**RMSE**	**MAE**
0.0027	0.629	0.364	0.438b

Table 6: Data obtained from scikit-learn's `LinearRegression` object

6 Conclusions

It is seen that the implementations based in `NumPy` and Pure Python practically do not have any difference other than the time taken to complete epochs. This is due to the fact that availing features of `NumPy` such as fast vectorized loops and broadcasting vectors only speed up each iteration and not the values obtained.
Further, the `LinearRegression` object from scikit-learn gave much better results on the same dataset as compared to the other approaches. This can be explained due to the fact that it does not use gradient-based approach. Rather, it opts for a OLS solver.
Personally, I have learnt a lot from this assignment mainly in the field of Exploratory Data Analysis, MatPlotLib, Seaborn and scikit-learn. I hope to learn even more in the upcoming weeks. The source code for this assignment is uploaded on GitHub at this repo!.