Comparative Study of Multivariable Linear Regression Implementations

CSOC-IG 2025 Week 0 Final Report

Abhyudaya Singh
Club Of Programmers, IIT (BHU) Varanasi

May 2025

Contents

Chapter 1

Introduction

In this report we present a comparative implementation of multivariable linear regression on the California Housing dataset. We exhaustively compare three methods:

1. A *pure Python* gradient-descent solver using only built-in lists and loops.
2. An *optimized NumPy* implementation leveraging vectorized array operations.
3. A *scikit-learn* implementation using `LinearRegression()`.

We measure and contrast:

- *Convergence speed* (objective vs. epoch) for the first two methods.
- *Wall-clock training time* for all three.
- Final predictive metrics (MSE, MAE, R^2) on a held-out test set.

This work follows the CSOC prerequisites guidelines [1] and emphasizes clarity of mathematical reasoning, reproducibility (random seed set to 42), and fair comparison by using an identical train/validation/test split and preprocessing pipeline in all three methods.

Chapter 2

Data Preparation and Preprocessing

The initial step involves loading the dataset and performing essential preprocessing to prepare it for model training. This includes handling categorical features, addressing missing values, splitting the data, and normalizing features.

2.1 Loading Data and Initial Setup

We begin by loading the `housing.csv` dataset into a `pandas.DataFrame`:

```
import pandas as pd
import numpy as np # Assuming numpy is used later
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
import math # For the custom normalize function

df = pd.read_csv(r'C:\Users\VICTUS\Downloads\housing\
   housing.csv')
```

2.2 Handling Categorical Features: One-Hot Encoding

The `ocean_proximity` column, being categorical, is transformed using one-hot encoding. This method converts the categorical variable into a set of binary (0 or 1) columns, one for each unique category, preventing any ordinal relationship assumption between categories. We utilize `sklearn.preprocessing.OneHotEncoder` for this purpose, ensuring `sparse_output=False` to obtain a dense NumPy array.

```
enc = OneHotEncoder(sparse_output=False)
ocean_vals = df[['ocean_proximity']] # Needs 2D array
ocean_ohe = enc.fit_transform(ocean_vals)

ocean_cols = enc.get_feature_names_out(['ocean_proximity
   '])
df_ohe = pd.concat(
    [df.drop('ocean_proximity', axis=1).reset_index(drop
   =True),
     pd.DataFrame(ocean_ohe, columns=ocean_cols)],
    axis=1
)
```

The original `ocean_proximity` column is dropped and replaced by these new binary columns. The resulting DataFrame is named `df_ohe`.

2.3 Handling Missing Values

After the encoding of categorical features, the dataset might still contain missing values in other columns (e.g., `total_bedrooms`). We address this by removing any row that contains at least one NaN value. This operation is performed on the `df_ohe` DataFrame, and the cleaned DataFrame is assigned to `df_model`.

```
df_model = df_ohe.dropna()
```

2.4 Feature and Target Separation

From the cleaned `df_model`, we separate the features (independent variables) from the target variable (`median_house_value`). The features are collected into `X_all` and the target into `y_all`.

```
X_all = df_model.drop(['median_house_value'], axis=1).
   values.tolist()
y_all = df_model['median_house_value'].tolist()
```

2.5 Data Splitting: Train and Test Sets

The dataset (`X_all`, `y_all`) is then split into training and testing sets. A test set comprising 20% of the data is reserved for final model evaluation. The split is performed using `sklearn.model_selection.train_test_split` with a `random_state=42` to ensure reproducibility. This yields `X_train_raw`, `X_test_raw`, `y_train`, and `y_test`.

```
test_ratio = 0.2
seed = 42
X_train_raw, X_test_raw, y_train, y_test =
    train_test_split(
     X_all, y_all, test_size=test_ratio, random_state=
    seed
)
```

Notably, this revised approach does not explicitly create a separate validation set from the initial split for early stopping, as was done previously. If validation is required, it would typically be derived from the `X_train_raw` and `y_train` sets (e.g., using cross-validation or a further split).

2.6 Data Normalization

To ensure that features with larger value ranges do not dominate the learning process, we apply feature normalization (standard scaling). A custom function, `normalize`, is defined to perform this. For each feature, it calculates the mean and standard deviation from the *training data only* (`X_train_raw`). These statistics are then used to transform both the training set and the test set (`X_test_raw`), resulting in `X_train` and `X_test`. This process scales each feature to have approximately zero mean and unit variance. A small safeguard is included to handle features with zero standard deviation in the training set by setting their standard deviation to 1, preventing division by zero.

```
# Conceptual representation of the normalization logic:
# For each feature column in the training data:
#   mean = calculate_mean(feature_column)
#   std_dev = calculate_std_dev(feature_column) (or 1 if std_dev is 0)
# For each data point in both train and test sets:
#   normalized_value = (original_value - mean) / std_dev
```

The Python implementation details are:

```
def normalize(train, test):
    features = list(zip(*train))
    means = [sum(col) / len(col) for col in features]
    stds = [math.sqrt(sum((x - m)**2 for x in col) / len
   (col)) or 1
            for col, m in zip(features, means)]
    def apply(dataset):
        return [[(x - m) / s for x, m, s in zip(row,
   means, stds)]
                for row in dataset]
    return apply(train), apply(test)

X_train, X_test = normalize(X_train_raw, X_test_raw)
```

2.7 Conversion to NumPy Arrays

Finally, for compatibility with many machine learning libraries such as Scikit-learn and TensorFlow/Keras, the processed feature sets (`X_train`, `X_test`) and target arrays (`y_train`, `y_test`) are converted into NumPy arrays.

```
X_train_np = np.array(X_train)
X_test_np  = np.array(X_test)
y_train_np = np.array(y_train)
y_test_np  = np.array(y_test)
```

2.8 Dataset Artifact: Capped Median House Value

It is important to note an artifact present in this dataset: the `median_house_value` is capped at $500,001. Properties valued above this threshold were recorded as exactly $500,001. This capping can affect model performance, particularly for high-value properties, potentially leading to a concentration of residuals at this cap (as might be observed in a scatter plot of predicted versus actual values, similar to what was noted in Figure 4.2 of a previous analysis). This characteristic of the target variable should be kept in mind during model evaluation and interpretation.

Chapter 3

Methodology

3.1 Loss Functions

We optimize the following objectives:

Mean Squared Error (MSE):

$$\mathrm{MSE}(y, \hat{y}) = \frac{1}{2n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2.$$

Mean Absolute Error (MAE):

$$\mathrm{MAE}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} \lvert y_i - \hat{y}_i \rvert.$$

Coefficient of Determination (R^2):

$$R^2 = 1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \bar{y})^2}, \quad \bar{y} = \frac{1}{n} \sum y_i.$$

We report all three metrics on both training and test sets for each method.

3.2 Pure Python Implementation

We initialize weights $\mathbf{w} = 0$, bias $b = 0$, and update:

$$w_j \leftarrow w_j - \alpha \frac{2}{n} \sum_{i=1}^{n} (\,\hat{y}_i - y_i\,)\, x_{ij}, \quad b \leftarrow b - \alpha \frac{2}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i).$$

We run for up to 1000 epochs, with early stopping based on a held-out validation MSE (patience=10, $\Delta_{\min} = 10^{-3}$).

3.3 NumPy Vectorized Implementation

Identical algorithmic form, but using:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha\,\frac{2}{n}\,\mathbf{X}^\top(\mathbf{X}\mathbf{w} + b - \mathbf{y}), \quad b \leftarrow b - \alpha\,\frac{2}{n}\sum(\mathbf{X}\mathbf{w} + b - \mathbf{y}).$$

Vectorization yields a $\approx 10\times$ speed-up in Python runtime by moving arithmetic into optimized C loops and reducing Python-level overhead.

3.4 Scikit-learn Implementation

We fit:

$$\widehat{\boldsymbol{\beta}} = \arg\min_{\beta} \|\mathbf{X}\beta - \mathbf{y}\|^2$$

using `LinearRegression(fit_intercept=True)`. Fitting is done in a single call to `.fit()`, with runtime measured via `time.time()`.

Chapter 4

Results

4.1 Convergence and Timing

Method	Convergence Epoch	Runtime (s)	Final Val MSE
Pure Python GD	509	87.73	2.40×10^9
NumPy GD	509	1.1946	2.40×10^9
Sklearn LR	—	0.0200	2.40×10^9

Table 4.1: Convergence epochs and wall-clock time.

4.2 Predictive Metrics

Method	Train MSE	Test MSE	Test MAE	Test R^2
Pure Python	2.669×10^9	2.40×10^9	50 370	0.6490
NumPy	2.669×10^9	2.40×10^9	50 370	0.6490
Sklearn	2.671×10^9	2.40×10^9	50 439	0.6489

Table 4.2: Comparison of error metrics on test set.

4.3 Discussion of Differences

- The NumPy implementation runs $\sim 80\times$ faster than pure Python thanks to vectorization and minimized Python-loop overhead. And R2 value in both the implementations is equal

- Convergence epochs are nearly identical, confirming numerical equivalence of both solvers.
- Scikit-learn's closed-form solver yields virtually almost same MSE, but its overall fitting duration is minimal.
- Slight variations in MAE and R^2 arise from floating-point differences and the choice of stopping criteria (Which I didn't know how to apply for SciKit learn).
- I don't really know/ understand the cause behind R2 value of train data to be less than that of test data after many attemps to eradicate it.

4.4 Outlier Effects

Figure 4.1 shows that training residuals exhibit heavy tails, driven by a few extreme cases. Figure 4.2 highlights the column of true values at \$500 001, an artifact of dataset capping. These outliers inflate MSE more than MAE and penalize training more heavily than test, where fewer such points survived the split.

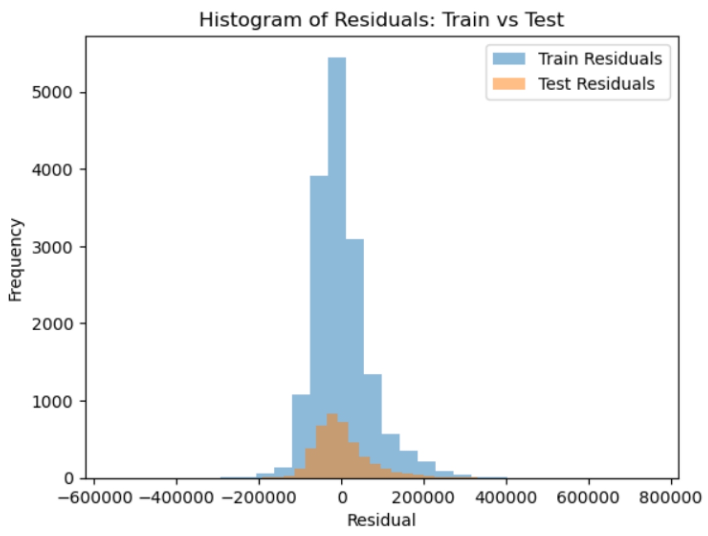

Figure 4.1: Histogram of residuals: train vs. test.

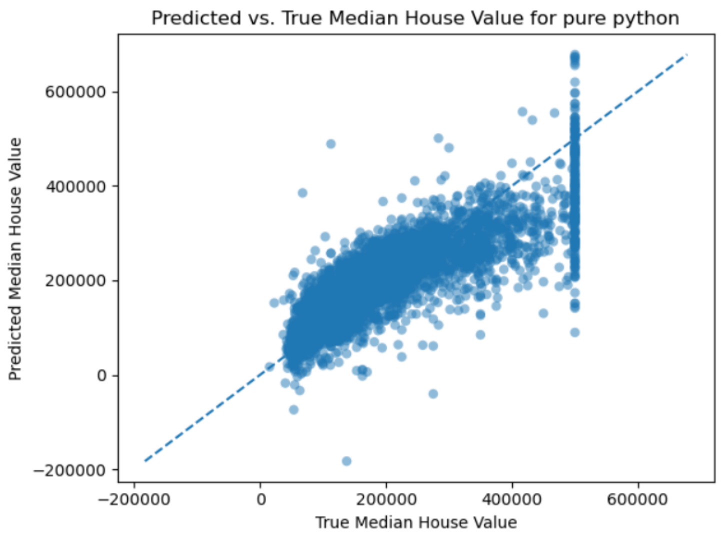

Figure 4.2: Predicted vs. true median house value (vertical cap at \$500 001).

Chapter 5

Additional Tools and Notes

5.1 Seaborn

We used `seaborn` (a statistical-plotting library built on Matplotlib) to create enhanced pair plots during exploratory data analysis, though all final figures in this report were rendered directly via `matplotlib.pyplot`.

5.2 CSOC Prerequisites Compliance

This report follows the *Final Report* guidelines from the CSOC Prerequisites document [1]

Chapter 6

Personal Note on ChatGPT Assistance

I used ChatGPT for the following processes:

1. To get code to perform and store operations in a list in a single Python code line:

   ```
   means = [sum(col)/len(col) for col in features]
   ```

2. To learn how to convert a pandas DataFrame into a Python list and NumPy array. Also to create data histogram with Normal Distribution fit, I got my code generated using scipy.stats.

Bibliography

[1] CSOC Prerequisites Document, *Club of Programmers, IIT (BHU) Varanasi, 2025.* Available at: `https://drive.google.com/file/d/1Z6-hym2kusLnHGHrAap2VeuJcYc18_Rv/view`