Comparative Study of Multivariable Linear Regression Implementations

Tanishq Ahuja (24074033)

May 19, 2025

Abstract

This report presents a comparative study of **multivariable linear regression** implementations using **gradient descent** to predict housing prices. A real-world dataset is used and values are predicted based on multiple features. We explore **three approaches**: a pure Python implementation, a vectorized Numpy based version, and a Scikit-learn model. The latter uses OLS and not gradient descent. Data preprocessing, including feature selection, scaling, and error evaluation is applied to increase model validity, The three methods are compared through visual plots of regression metrics (MAE, RMSE, R^2 Score) and model fit time which highlights each model's uniqueness.

Contents

1 Implementation Overview

1.1 Goal

The objective of this assignment is to implement multivariable linear regression using gradient descent from scratch with emphasis on convergence speed and predictive accuracy,

What are we predicting? The model is simple, we use several features provided in the housing dataset to predict the label `median_house_value`

1.2 Dataset

The dataset used for this study is the California Housing Prices dataset. It contains about 20,000 observations, 9 input features and 1 output label `median_house_value`. The input features include

- Geographical features
  - `longitude`
  - `latitude`
  - `ocean_proximity`
- Demographic Features
  - `population`
  - `households`
  - `median_income`
- Housing Characteristics
  - `housing_median_age`
  - `total_rooms`
  - `total_bedrooms`

2 Exploratory Data Analysis

2.1 Geographical Visualization

California housing prices based on geography .
I created a scatterplot of all districts to visualize the data (Figure 1). Then, I color coded it taking `median_house_value` as key. The density and sparsity of some areas is due to the population there.

```python
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))
plt.scatter(dataset['longitude'], dataset['latitude'],
            c=dataset['median_house_value'], cmap="coolwarm",
            ↪  alpha=0.5)
plt.colorbar(label="median_house_value")
plt.xlabel("longitude")
plt.ylabel("latitude")
plt.title("Geographical Plot")
plt.show()
```

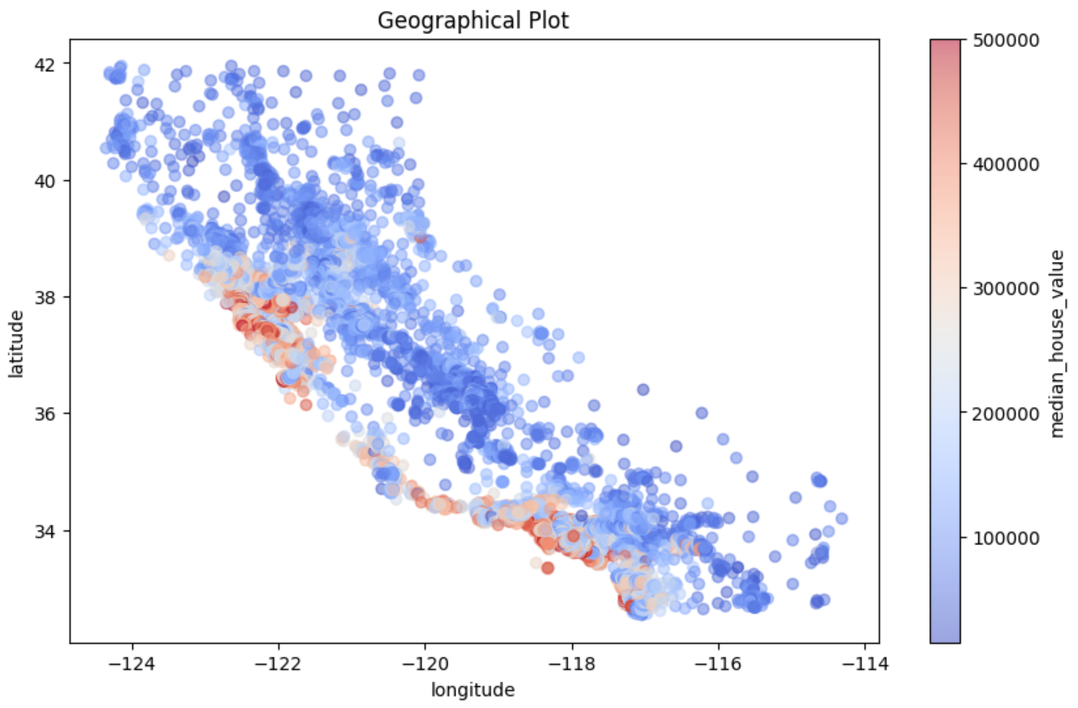

Figure 1: Geographical Scatterplot

Role of `ocean_proximity` .

Obviously, the geographical proximity of a housing from ocean would play a lot of role in its price (housings near the ocean show higher prices). To visualize this, I plotted a bar chart of the mean housing prices grouped by `ocean_proximity`. This allows us to compare the average median house value across different proximity categories to the ocean, helping to understand how location relative to the ocean affects housing prices.

```python
import seaborn as sns
sns.barplot(
    dataset.groupby("ocean_proximity")
    .mean()["median_house_value"].sort_values())
```

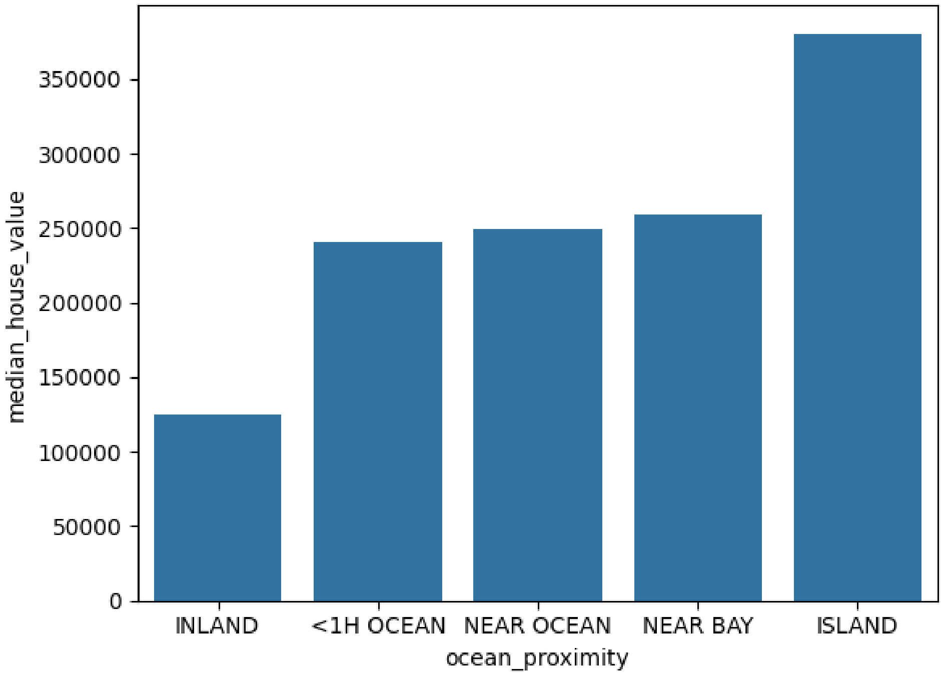

Figure 2: Housing Prices based on `ocean_proximity`

As we can see this feature shows high correlation. The price increases as we reach closer to the ocean. So, we can assign the categorical data a numerical value for example

- INLAND - 0
- <1H OCEAN - 1

and so on...

```python
dataset['ocean_proximity'] = dataset['ocean_proximity'].map({
    'INLAND': 0, '<1H OCEAN': 1, 'NEAR OCEAN': 2, 'NEAR BAY': 3,
    ↪  'ISLAND': 4
    })
```

2.2 Feature Relationships

We can plot a simple heatmap of feature correlations to find what all features are highly correlated with our label. This will help us in identifying important features later during feature selection and remove noise from our model.

```python
import seaborn as sns

corr = dataset.corr(numeric_only=True)
plt.figure(figsize=(12, 10))
sns.heatmap(corr, annot=False, cmap='coolwarm')
plt.title('Feature Correlation Heatmap')
plt.show()
```

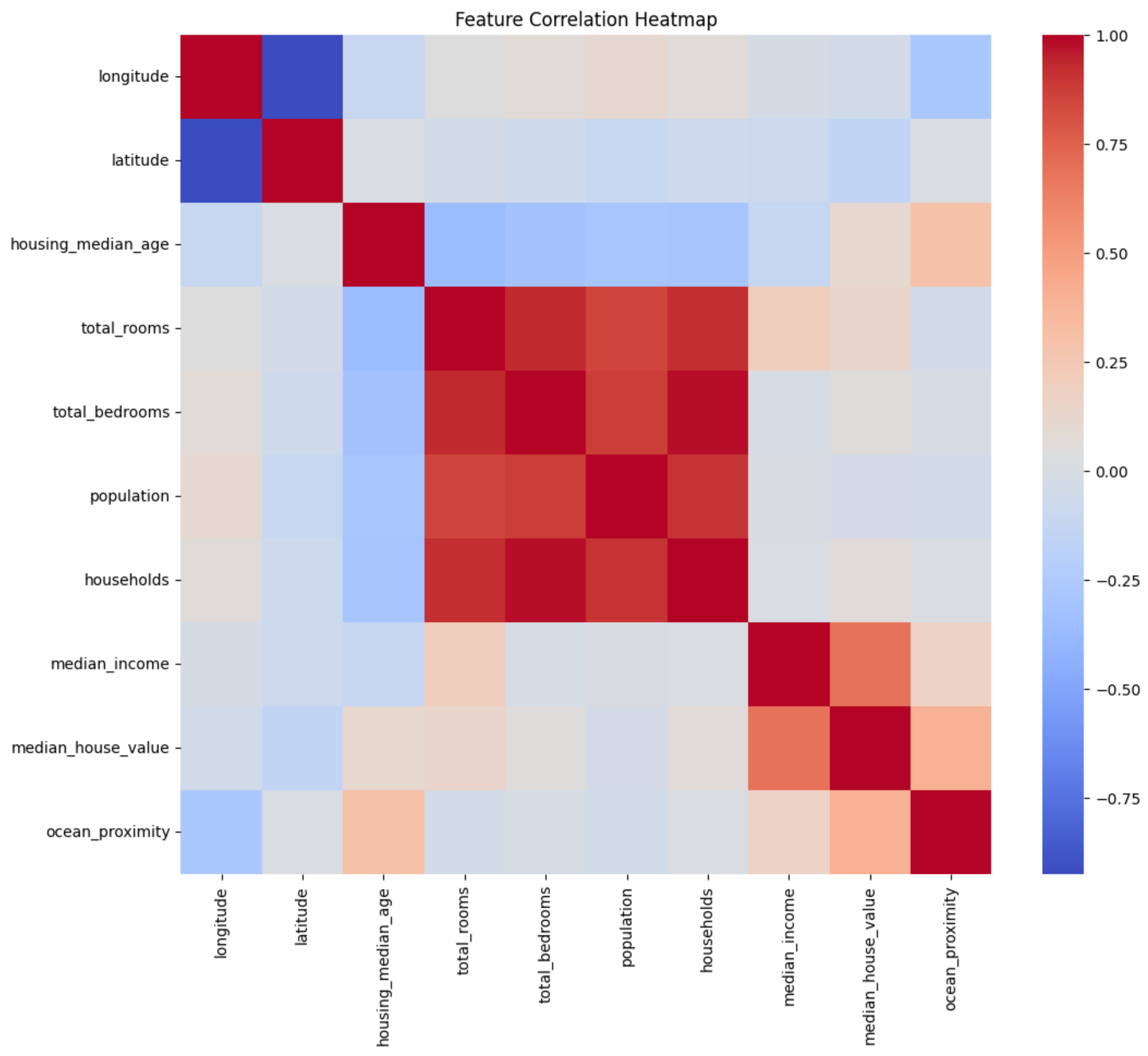

Figure 3: Heatmap of the correlation matrix of our data

3 Data Preprocessing

3.1 Data Imputation

The dataset provided to us contains some missing values in certain features. To address this, I perform data imputation using `SimpleImputer` frrm `sklearn.impute`, where the missing values were filled with **median** as the strategy. This step ensures a complete dataset, allowing for more reliable and accurate modeling.

```python
# to check what columns require imputation
dataset.isna().sum()

# performing imputation
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy='median')
dataset = pd.DataFrame(imputer.fit_transform(dataset),
 ↪  columns=dataset.columns)
```

3.2 Feature Creation

Features like `total_rooms` and `total_bedrooms` would not make much sense since they represent the total values in the whole district. Rather, features like `total_rooms_per_household` will make a more significant and direct impact in predicting house prices.

```python
dataset["rooms_per_household"] = dataset["total_rooms"] /
 ↪  dataset["households"]
dataset["bedrooms_per_household"] = dataset["total_bedrooms"] /
 ↪  dataset["total_rooms"]
dataset["population_per_household"] = dataset["population"] /
 ↪  dataset["households"]
```

3.3 Target Variable Correction

The target variable `median_house_value` is capped in this dataset (at 500,000), this can skew our model and may result in a model that undervalues expensive properties. So, for this I will drop all the datapoints that have capped values.

This will limit the range of data on which the model can be used but it will largely improve the accuracy.

```python
dataset = dataset[dataset['median_house_value'] <
 ↪  dataset['median_house_value'].max()]
```

3.4 Feature Selection

I plotted all numerical factors against `median_house_value` to see which features have strong linear dependence and which ones create noise.

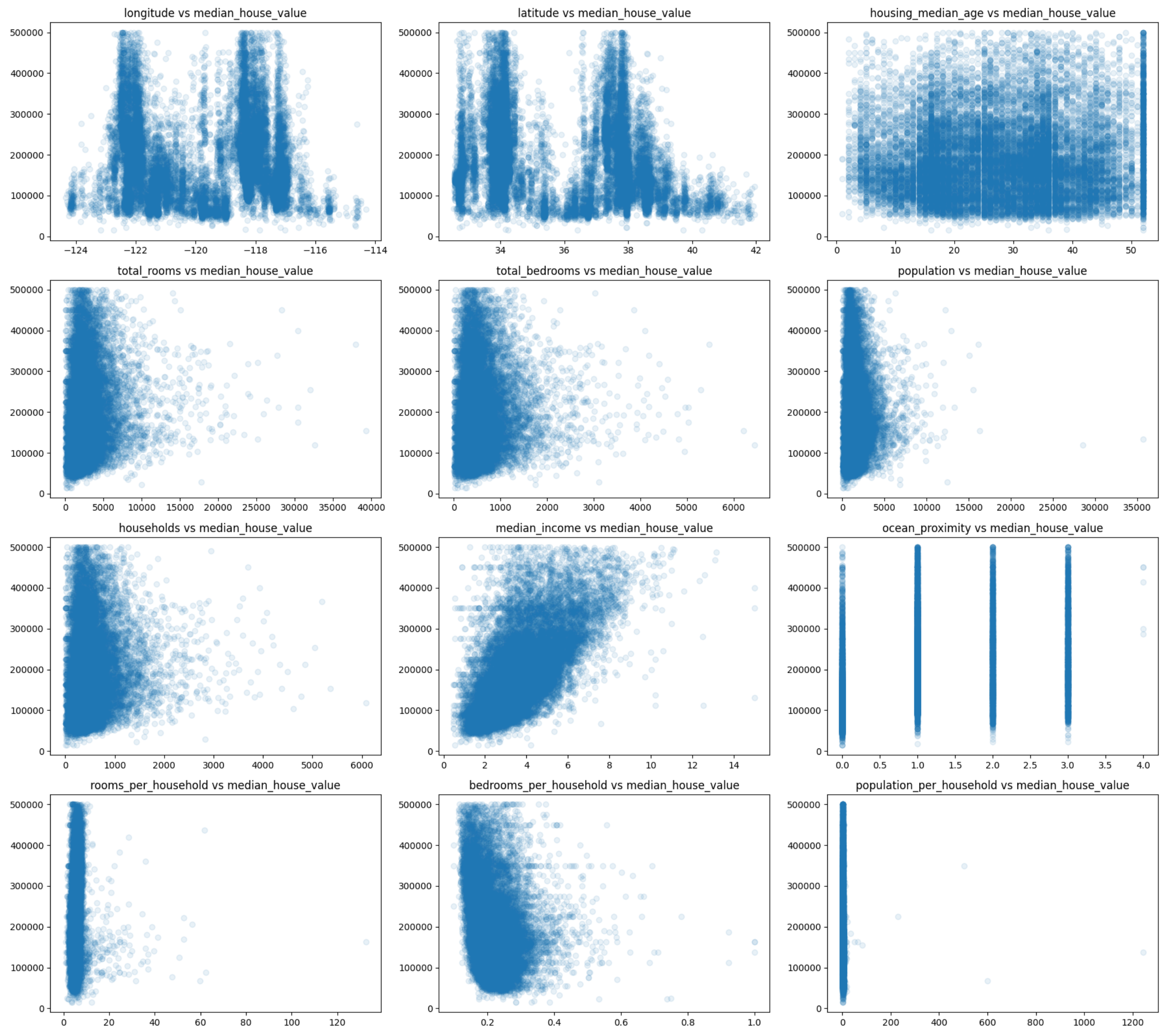

Figure 4: Plots to check linear dependence of features

Using the above plots and the previously generated correlation heatmap, I select the features that have very low significance in predicting the label and drop them from the dataset. This step helps reduce overfitting, improves model interpretability, and speeds up training by eliminating redundant or weakly correlated features.

```
low_corr_features = ["total_rooms", "total_bedrooms", "population",
↪  "households", "population_per_household", "longitude"]
dataset.drop(low_corr_features, axis=1, inplace=True)
```

3.5 Test Train Split

Now, I can split the data into testing and training data. I will use `sklearn` for that.

```python
from sklearn.model_selection import train_test_split
train, test = train_test_split(dataset, test_size=0.1)
X = train.drop("median_house_value", axis=1).to_numpy().tolist()
y = train["median_house_value"].to_numpy().tolist()
X_test = train.drop("median_house_value", axis=1).to_numpy()
y_test = train["median_house_value"].to_numpy()
```

3.6 Feature Scaling

For multivariable linear regression, feature scaling is a very important process since all the features have values in a scale of their own. For example, the number of rooms ranges from 1 to 10 while the median income must be of the order 10^6. This can result in a lot of variation in the weights and may make the model inaccurate.

3.6.1 Z-score Normalization

Z-score normalization is a technique which rescales features so that they behave like standard normal distribution. It also does not alter the shape of the data, makes the mean 0 and standard deviation 1.

It is especially useful for algorithms like **gradient descent** since they are sensitive to scale of the features.

For a given feature x, each value is normalized using the formula:

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

Python Implementation of Z-Score Normalization .

```python
def normalize(X):
    for i in range(len(X[0])):
        mu = mean([x[i] for x in X])
        sigma = std([x[i] for x in X])
        # mean and std are already defined
        for j in range(len(X)):
            X[j][i] = (X[j][i] - mu) / sigma
    return X

X_norm = normalize(X)
```

4 Model Implementations

4.1 Gradient Descent Theory

4.1.1 Weights and Bias

In a multivariable linear regression model, our main goal is to find weights and bias. The model (which is inherently a multivariable function) is defined as:

$$f_{\mathbf{w},b}(\mathbf{x}) = \textstyle\sum_{i=1}^{n} w_i x_i + b$$

where x is the feature vector $[x_1, x_2, ..., x_n]$ and w is the weight vector $[w_1, w_2, ..., w_n]$.

Our objective is to learn the optimal parameters w and b that minimize the error between the model's predictions and the true target values.

4.1.2 Cost Function

To bring this error into perspective, we use something called as a loss function (sometimes called cost function). It is a function of the weight vector and the bias.

For algorithms like gradient descent, it is commonly the **Mean Squared Error (MSE)** which is defined as:

$$J(\mathbf{w}, b) = \tfrac{1}{2m} \textstyle\sum_{j=1}^{m} \left(f_{\mathbf{w},b}(\mathbf{x}^{(j)}) - y^{(j)}\right)^2$$

To minimize this cost function, we apply an algorithm known as gradient descent. As its name suggests, in the algorithm we descend down to the minima using the gradient of the cost function.

4.1.3 Gradient Descent Equations

Gradient Descent is an iterative optimization algorithm. In each iteration, we update the weights and bias using the gradients (commonly known as partial derivatives) of the loss function with respect to the weights and bias.

After each iteration, weights and bias are updated using these equations

$$w_i := w_i - \alpha \tfrac{\partial J}{\partial w_i}$$

$$b := b - \alpha \tfrac{\partial J}{\partial b}$$

where α is the learning rate. It controls the step-size of each update. Repeated application of these updates gradually make the model parameters (weights and bias) converge. Since, the loss function used has only one minima, for this application the convergence will always be at the global minima. For above mentioned J,

$$\tfrac{\partial J}{\partial w_k} = \tfrac{1}{m} \textstyle\sum_{i=1}^{m} \left(f^{(i)} - y^{(i)}\right) \cdot x_k^{(i)}$$

$$\tfrac{\partial J}{\partial b} = \tfrac{1}{m} \textstyle\sum_{i=1}^{m} \left(f^{(i)} - y^{(i)}\right)$$

4.2 Implementations

4.2.1 Pure Python

I opted to write an object-oriented implementation of the Linear Regression model using gradient descent.

```python
class LinearRegression_pure:
    def __init__(self, num_features):
        self.num_features = num_features
        self.weights = [0 for i in range(self.num_features)]
        self.bias = 0
        self.costs = []

    def gradient_descent(self, X, y, alpha, epochs):
        m = len(y)
        for epoch in range(epochs):
            dj_dw = [0 for i in range(self.num_features)]
            dj_db = 0

            for i in range(m):
                f_i = self.predict(X[i])
                dj_db += f_i - y[i]
                for k in range(self.num_features):
                    dj_dw[k] += (f_i - y[i]) * X[i][k]

            dj_db = dj_db / m
            dj_dw = [dj_dw[k] / m for k in range(self.num_features)]

            self.weights = [self.weights[k] - alpha * dj_dw[k] for k in range(self.num_features)]
            self.bias = self.bias - alpha * dj_db
            self.costs.append(self.cost(X, y))

    def cost(self, X, y):
        m = len(y)
        total_cost = 0
        for i in range(m):
            f_i = self.predict(X[i])
            total_cost += (f_i - y[i])**2
        return total_cost / (2 * m)

    def predict(self, x):
        f = 0
        for k in range(self.num_features):
            f += self.weights[k] * x[k]
        f += self.bias
        return f
```

Code Walk-through

1. **Structure** A linear regressor object needs to be created by passing in the number of features in the linear regression model. Next, the data is fit and the parameters are adjusted using the `gradient_descent` method which takes in the argument of training data, learning rate (α) and the number of epochs.

   The object created contains attributes like weights, bias and costs which are regression metrics and can be accessed. Object oriented implementation provides a reusable and clean looking code.

   The training data needs to be in the following format

   - **`X`** A python list of lists, each element containing the feature vector.
   - **`y`** A python list each element containing the value of target variable.

2. **Initialization** The constructor take input the number of features in the linear regression model. It initializes all the weights and bias to zero. It also initializes a list `costs` to store the cost over each epoch. This list will later help us in plotting the convergence of the cost function as epochs go by.

3. **Gradient Descent** Although, Stochastic Gradient Descent (SGD) is faster, I have implemented batch gradient descent for now.

   In each epoch,

   - The partial derivative of cost with respect to weights ($\alpha\frac{\partial J}{\partial \mathbf{w}}$) list `dj_dw` and the partial derivative of cost with respect to bias ($\alpha\frac{\partial J}{\partial b}$) `dj_db` are initialized as zero.
   - Then, the algorithm loops over entire dataset and keeps calculating $\frac{\partial J}{\partial w_k}$ for all weights and also $\frac{\partial J}{\partial b}$ using above mentioned equations. After the loop ends, finally mean is taken by dividing by the total sample size.
   - Then, weights and bias are updated again by using the above mentioned equations.
   - The cost is also calculated in each epoch and appended to the list.

4. **Cost** The cost function computes Mean Squared Error (MSE)

5. **Predict** It computes the value of f using $f_{\mathbf{w},b}(\mathbf{x}) = \sum_{i=1}^{n} w_i x_i + b$

4.2.2 Numpy Vectorized

The implementation of the same algorithm with similar methods and attributes using `numpy` is as follows

```python
class LinearRegression_np:
    def __init__(self, num_features):
        self.num_features = num_features
        self.weights = np.zeros(self.num_features)
        self.bias = 0
        self.costs = []

    def gradient_descent(self, X, y, alpha, epochs):
        m = len(y)
        for epoch in range(epochs):
            dj_dw = np.zeros(self.num_features)
            dj_db = 0

            f = self.predict(X)
            dj_db = np.mean(f - y)
            dj_dw = np.dot(X.T, (f - y)) / m

            self.weights = self.weights - alpha * dj_dw
            self.bias = self.bias - alpha * dj_db
            self.costs.append(self.cost(X, y))

    def cost(self, X, y):
        m = len(y)
        total_cost = 0
        f = self.predict(X)
        return np.sum((f - y)**2) / (2 * m)

    def predict(self, x):
        return np.dot(x, self.weights) + self.bias
```

As it is clear from the code length, the `numpy` implementation is much shorter and looks cleaner because of `numpy`'s good wrapping abilities. This implementation is also faster thanks to `numpy`'s computation efficiency.

How Numpy makes it fast? .

`Numpy` achieves better performance mainly through **vectorization**. It mainly removes for loops with array-wide operations. Most of the work in the numpy implementation is done through matrix multiplication and `numpy` seems to have mastered it fully utilizing the CPU operations.

If we assume, that X is a matrix of shape (m, n) where m is the number of samples and n is the number of features then, the gradients for the whole batch can be easily computed at once using:

$$\frac{\partial J}{\partial \mathbf{w}} = \frac{1}{m} X^T (X\mathbf{w} + b - y)$$

Similarly, predictions and cost fucntion computations can be reduced to matrix operation making everything many folds faster.

4.2.3 Scikit-learn Implementation

```python
from sklearn import linear_model
LinearRegression_sk = linear_model.LinearRegression()
LinearRegression_sk.fit(X_norm, y)
```

5 Visualization

5.1 Comparative Plot

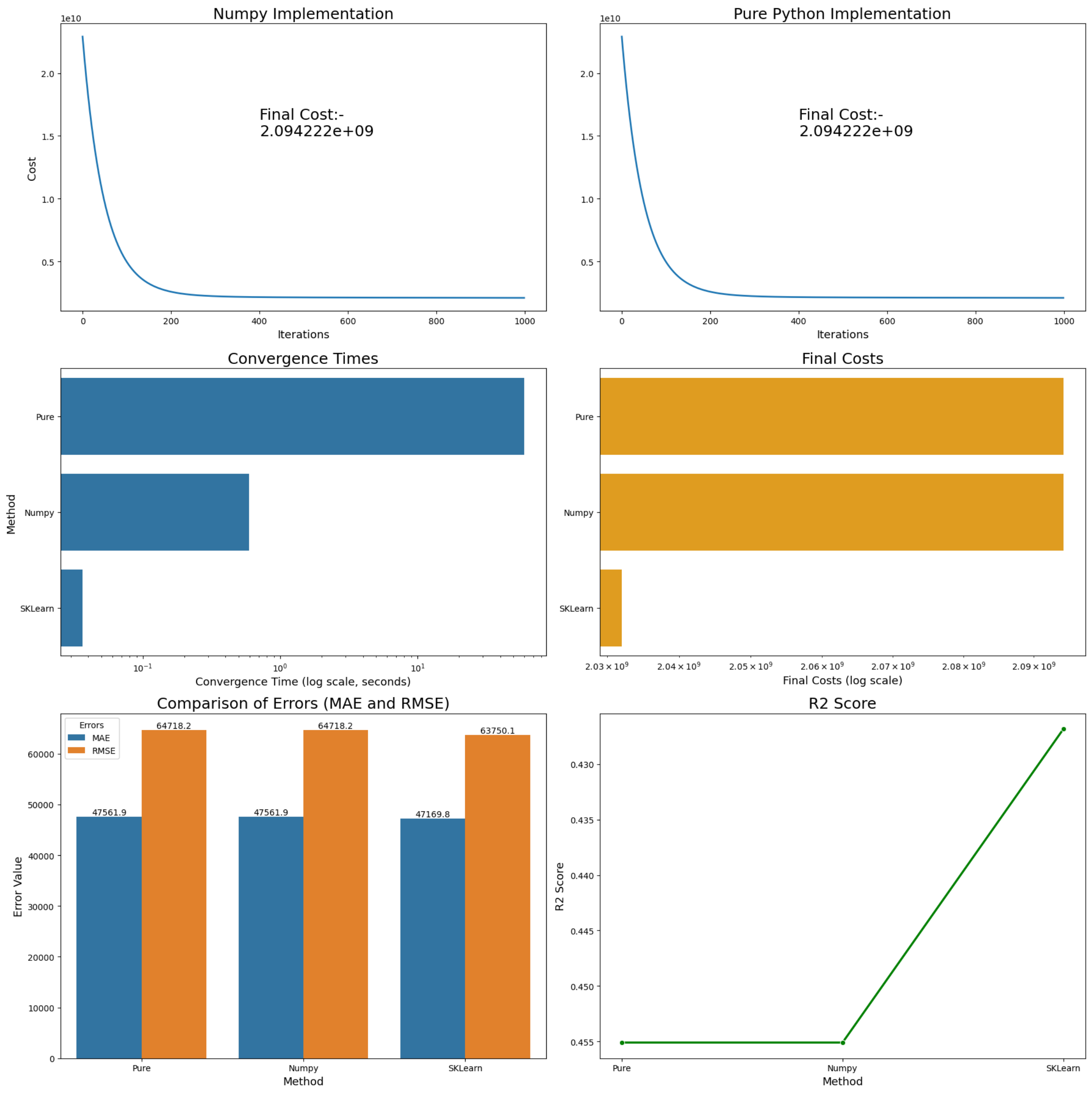

Figure 5: Plots summarizing comparative analysis

5.2 Code for the plot

Refer end of document

6 Analysis and Discussion

6.1 Convergence Times and Accuracies

- `Numpy` based implementation converged faster than pure Python due to above explained vectorization and efficient operation benefits. While, `sklearn` based implementation was the fastest and provided better scores, it has underlying scalability issues as discussed further.
- The accuracy of Python and Numpy implementation is exactly same because of the same underlying algorithm for the two, The only differing factors are the optimization strategy. `sklearn` on the other hand does not use **Gradient Descent** at all. It implements **Ordinary Least Squares (OLS)**.

What is OLS? It is a closed-form algorithm (while gradient descent is iterative optimization algorithm). The solution is found analytically by solving mathematical equations. Our main goal is to minimize the cost function. To do so, the total derivative of the cost function is computed and put as 0. The bias term is also put in the weight vector itself and need not be computed separately, The mathematical equation is as follows

$$\frac{\partial J}{\partial \mathbf{w}} = \frac{1}{m}\mathbf{X}^T(\mathbf{X}\mathbf{w} - \mathbf{y}) = 0 \Rightarrow \mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

This gives more accurate solution for minimization of J but obviously does not guarantee complete fitting of the model.

6.2 Scalability

Out of the 3 implementations, the numpy implementation of Gradient Descent is the most scalable.

- The pure python implementation is beautiful but is not scalable due to poor resource management and computational inefficiency. Numpy not only makes the code cleaner but also makes it faster.
- The sklearn implementation which uses OLS as the underlying algorithm is way faster (for smaller dataset) compared to Numpy but fails in some cases. Some of them are
  1. **Large datasets** Computation of the inverse matrix is a very intensive task and can easily get out of hand when the dataset is very large
  2. **Non-linear models** Gradient descent can be easily extended to train non-linear models while OLS is strictly for linear models.

6.3 Influence on convergence

Initial Parameters In my implementation, I have initialized all parameters as 0. For gradient descent using MSE as cost function, convergence is guaranteed to only one local minima (assuming appropriate α is chosen). But, the initial value will heavily impact the convergence time. If weights that are already very close to the ideal weights are chosen, then convergence will obviously be reached in less number of epochs.

Learning Rate (α) The learning rate needs to be appropriately chosen since it determines how big of a step we will take in each epoch. Consider the following scenarios

- **Very Large α** Due to large value of alpha, the minima will be missed entirely and the cost function will start showing uprise as shown below

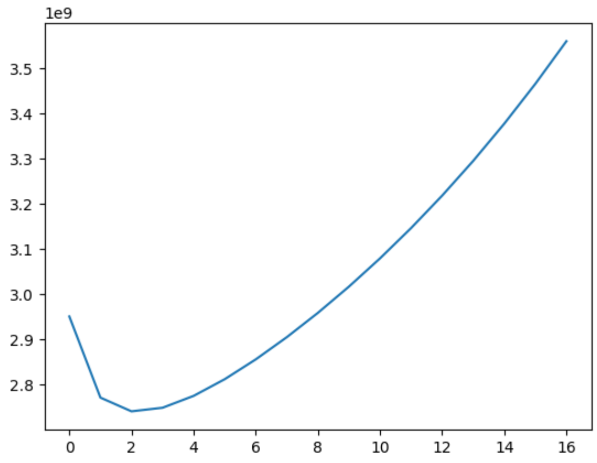

Figure 6: Very Large α

- **Very Small α** Due to small value of alpha, the minima might be missed entirely and it might take a lot more epochs and hence a lot more time for the cost function to converge

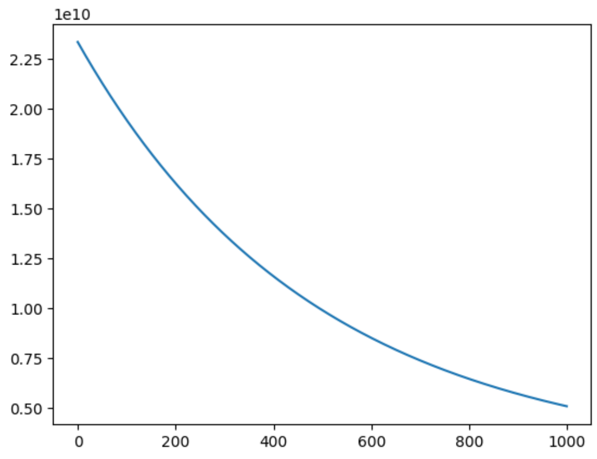

Figure 7: Very Small α

7 Code for the Plot

7.0.1 Calculating metrics

```python
skl_cost = get_cost(LinearRegression_sk.coef_,
↪  LinearRegression_sk.intercept_, np.array(X_norm), np.array(y))
pred_pure = [lg.predict(x) for x in X_test]
pred_np = [lg_np.predict(x) for x in X_test]
pred_skl = LinearRegression_sk.predict(X_test)

metrics = pd.DataFrame({
    "Method":["Pure", "Numpy", "SKLearn"],
    "Time": [pure_time, np_time, skl_time],
    "Final Cost": [lg.costs[-1], lg_np.costs[-1], skl_cost],
    "MAE": [mae(pred_pure, X_test, y), mae(pred_np, X_test, y),
    ↪  mae(pred_skl, X_test, y)],
    "R2": [r2(pred_pure, X_test, y), r2(pred_np, X_test, y),
    ↪  r2(pred_skl, X_test, y)],
    "RMSE": [rmse(pred_pure, X_test, y), rmse(pred_np, X_test, y),
    ↪  rmse(pred_skl, X_test, y)]
    })
metrics.set_index("Method")
```

7.0.2 Plotting

```python
import matplotlib.gridspec as gridspec

fig = plt.figure(figsize=(18, 18))
gs = gridspec.GridSpec(3, 2, height_ratios=[1, 1, 1.2])

ax1 = fig.add_subplot(gs[0, 0])
ax2 = fig.add_subplot(gs[0, 1])
ax3 = fig.add_subplot(gs[1, 0])
ax4 = fig.add_subplot(gs[1, 1])
ax5 = fig.add_subplot(gs[2, 0])
ax6 = fig.add_subplot(gs[2, 1])

ax1.set_title("Numpy Implementation", fontsize="18")
ax1.set_ylabel("Cost", fontsize="13")
ax1.set_xlabel("Iterations", fontsize="13")
sns.lineplot(lg_np.costs, ax=ax1, linewidth=2)
ax1.text(400, 1.5e10, f"Final Cost:- \n{lg_np.costs[-1]:e}",
↪  fontsize=18)
```

```python

ax2.set_title("Pure Python Implementation", fontsize="18")
ax2.set_xlabel("Iterations", fontsize="13")
sns.lineplot(lg.costs, ax=ax2, linewidth=2)
ax2.text(400, 1.5e10, f"Final Cost:- \n{lg.costs[-1]:e}", fontsize=18)


ax3.set_title("Convergence Times", fontsize="18")
sns.barplot(x="Time", y="Method", data=metrics, ax=ax3)
ax3.set_xscale("log")
ax3.set_xlabel("Convergence Time (log scale, seconds)", fontsize="13")
ax3.set_ylabel("Method", fontsize="13")

ax4.set_title("Final Costs", fontsize="18")
sns.barplot(x='Final Cost', y='Method', data=metrics, ax=ax4, color="orange")
ax4.set_xscale("log")
ax4.set_xlabel("Final Costs (log scale)", fontsize="13")
ax4.set_ylabel("")

# melted dataframe for grouped barplot
melted = metrics[["Method", "MAE", "RMSE"]].melt(id_vars="Method", var_name="Metric", value_name="Value")
ax5.set_title("Comparison of Errors (MAE and RMSE)", fontsize="18")
sns.barplot(data=melted, x="Method", y="Value", hue="Metric", ax=ax5)
ax5.bar_label(ax5.containers[-1], label_type="edge")
ax5.bar_label(ax5.containers[-2], label_type="edge")
ax5.set_xlabel("Method", fontsize="13")
ax5.set_ylabel("Error Value", fontsize="13")
ax5.legend(title="Errors")

ax6.set_title("R2 Score", fontsize="18")
sns.lineplot(data=metrics, x="Method", y="R2", marker="o", ax=ax6, color="green", linewidth=2.5)
ax6.set_xlabel("Method", fontsize="13")
ax6.set_ylabel("R2 Score", fontsize="13")
ax6.invert_yaxis()


plt.tight_layout()
plt.show()
```