Implementation of Linear Regression from Scratch

This report documents the implementation of Linear Regression from scratch using Gradient Descent. The model was tested on the California Housing dataset to predict median house values based on multiple input features.

Name: Sarthak Narnor

Roll No: 1220

Objective: The goal is to implement and understand the Linear Regression model for predicting continuous values using Gradient Descent.

---

1. Data Characteristics

Dataset Properties

- Instances: 20,640
- Features: 8 numerical, no categorical features
- Target: Continuous (Median house value)
- Source: 1990 U.S. Census data (California districts)

Feature(X) Information

1.	MedInc	median income in block group
2.	HouseAge	median house age in block group
3.	AveRooms	average number of rooms per household
4.	AveBedrms	average number of bedrooms per household
5.	Population	block group population
6.	AveOccup	average number of household members
7.	Latitude	block group latitude
8.	Longitude	block group longitude

Target (y): MedHouseVal (Median house value)

---

2. Methodology

This report details the implementation of a Linear Regression model from scratch to predict housing prices using the California Housing dataset. The model was trained using the **Gradient Descent** optimization algorithm to find the optimal weights (w) and bias (b) that minimize the **Mean Squared Error (MSE)** cost function.

Implementation steps

1. Load Dataset: The process begins by loading the California Housing dataset using the fetch_california_housing function from Scikit-learn.
2. Split Data: The train_test_split function is used to partition the data into an 80% training set (X_train, y_train) and a 20% testing set (X_test, y_test) for model evaluation.

3. Standardize Features: The StandardScaler class is used for feature scaling. To prevent data leakage, the scaler is fitted only on the training data (X_train) and then used to transform both the training and testing sets. This ensures all features have a mean of 0 and a standard deviation of 1.
4. Initialize Model: An instance of our custom LinearRegression class is created, setting key hyperparameters like the learning_rate (e.g., 0.01) and the number of n_iterations (e.g., 1000).
5. Train Model: The model.fit(X_train_scaled, y_train) method is called. This executes the from-scratch Gradient Descent loop, which iteratively adjusts the model's weights and bias to minimize the Mean Squared Error (MSE) on the training data.
6. Evaluate Model: The trained model's performance is assessed on the unseen test set. Predictions are made using the model.predict(X_test_scaled) method, and these are compared against the true values (y_test) to calculate the Mean Squared Error (MSE) and the R-squared (R^2) score.
7. Visualize Results: Finally, matplotlib is used to generate and save key visualizations for analysis:

   - The Learning Curve, showing the cost reduction over iterations.

   - The Actual vs. Predicted plot, which visually assesses the model's predictive accuracy.

Understanding the Linear Regression Algorithm

The Linear Regression we are using here is essentially an algorithm that tries to find the **best possible straight-line relationship** between a set of features (like house age, median income) and a target value (the house price).

Linear Regression fits a linear model to a set of data points. The model's prediction is a weighted sum of the input features:

$$\hat{y} = \mathbf{w}^T\mathbf{x} + b = w_1x_1 + \cdots + w_nx_n + b$$

If we only had one feature, it would be the familiar equation of a line:

price = (weight * feature) + bias

Since we have multiple features, the formula is just an extension of that:

Price = (w1 * feature1) + (w2 * feature2) + ... + b

- The weights (w) are numbers the model learns that represent how much importance or influence each feature has on the final price.

- The bias (b) acts as a baseline or starting price for a house before any features are considered.

The goal is to find the optimal weights (w) and bias (b) by minimizing a cost function, typically the Mean Squared Error (MSE), using Gradient Descent. Based on this error, it slightly adjusts all the weights and the bias to try and make the error a little smaller. It repeats this process thousands of times, making small corrections in each step until it finds the set of weights and bias that produce the lowest possible error.

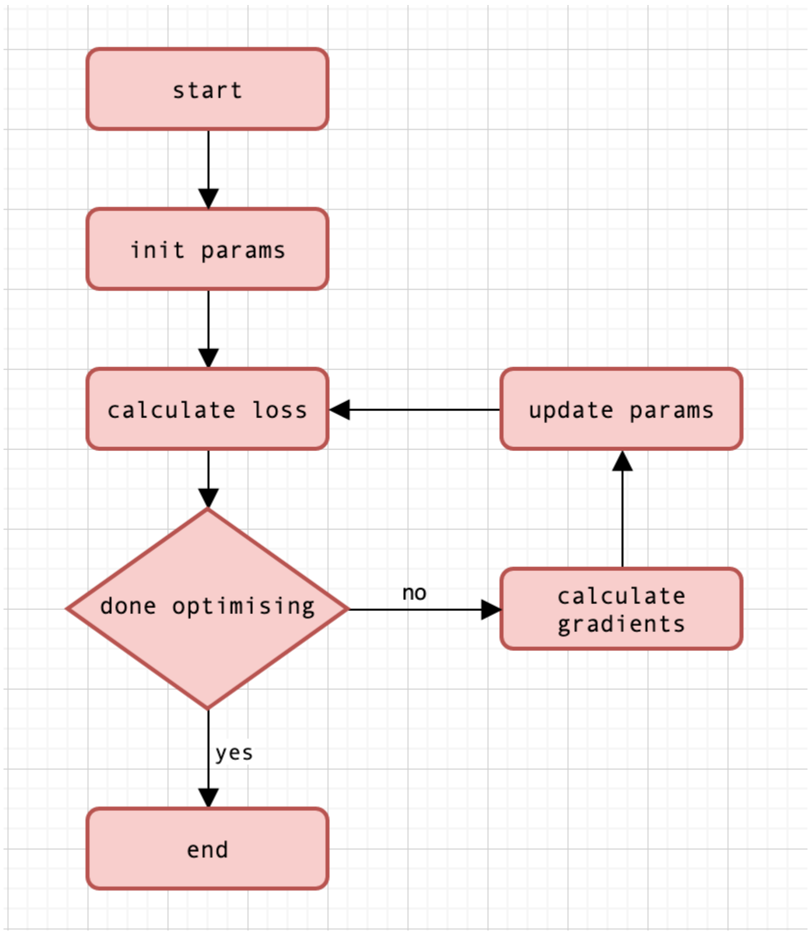
The update rules for the parameters at each iteration are:

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

Where α is the learning rate and m is the number of training examples.

The implementation is encapsulated within a Python class, LinearRegression, which contains methods for initialization (__init__), training (fit), and prediction (predict). The dataset was first split into an **80%** training set and a **20%** testing set. All input features of the dataset were then pre-processed using **standardization**, scaling them to have a mean of 0 and a standard deviation of 1. This step is crucial for ensuring efficient convergence of the **Gradient Descent Algorithm**. The model was then trained on the scaled training data.

Gradient Decent Algorithm



Gradient Decent Algorithm

Gradient Descent is an iterative optimization algorithm used to find the minimum of a function. In machine learning, its purpose is to adjust a model's parameters (e.g., weights and bias) to minimize the cost function, which measures the model's error.

The algorithm operates by repeatedly calculating the gradient of the cost function with respect to each parameter. The gradient indicates the direction of steepest ascent. To minimize the cost, the parameters are updated by taking a step in the opposite direction of the gradient.

The magnitude of this step is determined by a hyperparameter called the learning rate (α). This iterative process systematically guides the parameters towards values that result in the lowest possible cost, thus finding the optimal fit for the data.

---

3. Results and Evaluation

After training, the model's performance was evaluated on the unseen test set to assess its predictive accuracy. The key quantitative and visual results are presented below.

Quantitative Metrics

The primary evaluation metrics calculated on the test set are summarized in the table below.

Table 1: Model Performance Metrics

Metric	Testing Value
Mean Squared Error (MSE)	0.5546
R-squared (R^2) Score	0.5768

The **Mean Squared Error (MSE)** represents the average of the squared differences between the model's predicted house prices and the actual prices. A lower value is better. The **Mean Squared Error (MSE)** of **0.5546** quantifies the model's average prediction error. The **RMSE** is approximately **0.745**. Since the target variable is in units of $100,000s, this suggests that, on average, the model's predictions are off by about **$74,500**.

An R^2 score of **0.5768** means that approximately **57.7%** of the variability in California housing prices can be explained by the features in your model. This indicates a **moderately strong** relationship. While there is still **42.3%** of the variance that the model fails to capture, it shows that the model is significantly better than simply guessing the average house price.

Upon completion of the training process, the Gradient Descent algorithm converged on a final set of optimal parameters for the model. These parameters, the bias and the weights, collectively define the linear equation used for making predictions.

The Bias (or Intercept)

The Learned Bias (Intercept) was found to be **2.0719**. This value represents the predicted house price for a perfectly average house, where all features (like income, age, etc.) are zero (or their mean value). In simple terms, it's the b in the line equation **y = mx + b**.

- Model's Bias (2.0719): This is the base price prediction before the specific features of a house are considered.

The Weights (or Coefficients)

The Learned Weights (Coefficients) for the eight features are as follows: **[0.858, 0.150, -0.252, 0.281, 0.007, -0.043, -0.683, -0.654]**.

Each weight corresponds to a specific feature in the dataset and quantifies its impact on the final prediction.

- **Sign:** A positive weight indicates a positive correlation with the house price (e.g., the first feature with a weight of 0.858), while a negative weight indicates a negative correlation (e.g., the seventh feature with a weight of -0.683).
- **Magnitude:** The absolute value of the weight signifies the strength of its influence. Features with larger absolute weights, such as the first and seventh, are the most significant predictors in this model.

Visual Results

The model's training process and predictive performance are visualized in the following plots.

Figure 1: The learning curve, showing the MSE cost decreasing over 1000 iterations. This visualizes the convergence of the Gradient Descent algorithm.

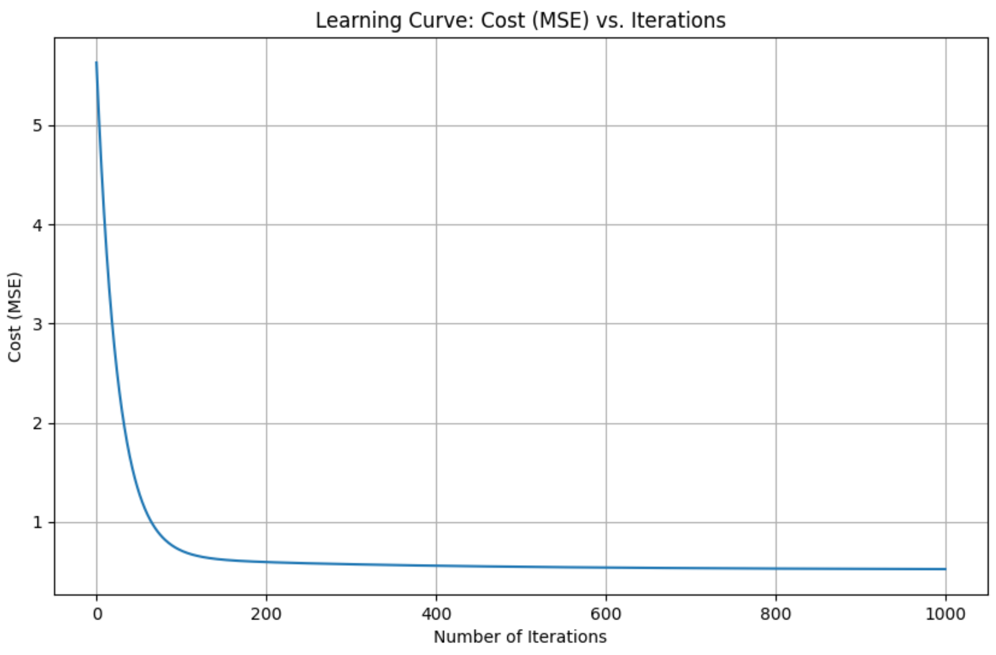


Figure 2: A scatter plot of the actual vs. predicted values on the test set.

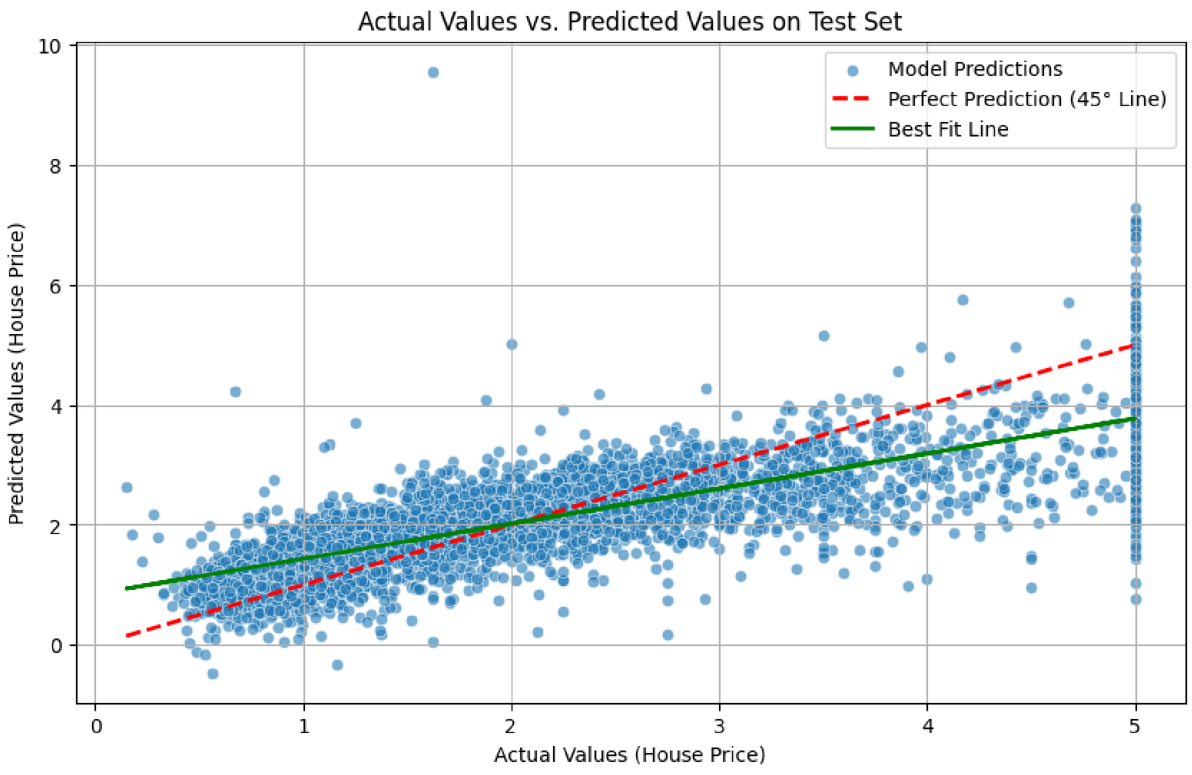
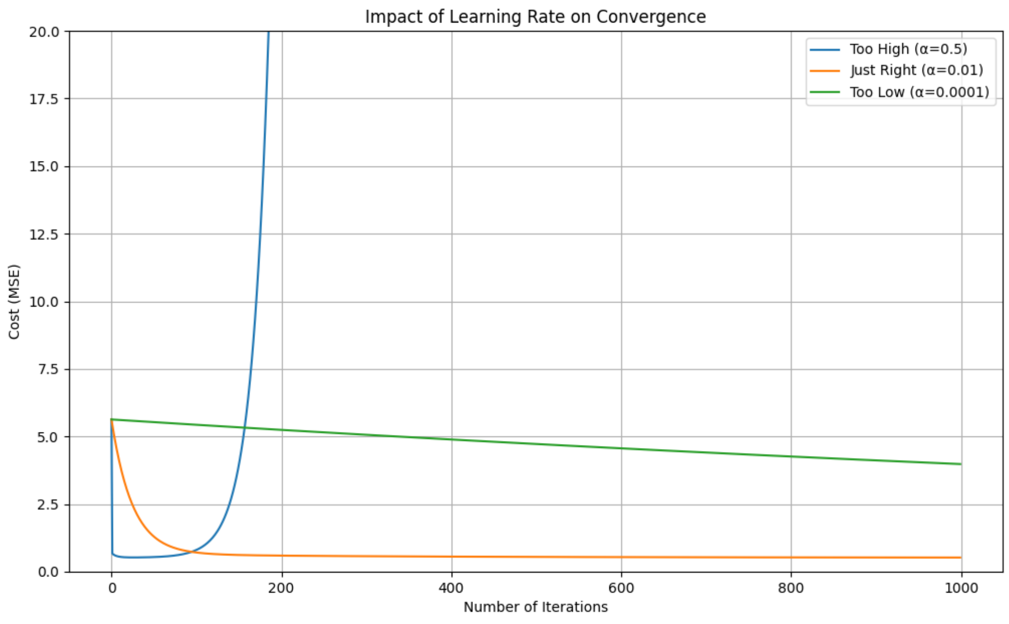


Figure 3: A multiple line plots showing the impact of learning rate(α) on convergence.



4. Analysis and Observations

The evaluation metrics indicate a moderately effective model. An **R^2 score of 0.5768** suggests that our model can explain approximately **57.7% of the variance** in California housing prices, which is a reasonable result for a simple linear model on this complex dataset. The **MSE of 0.5546** represents the average squared difference between the actual and predicted prices.

The learning curve (Figure 1) clearly demonstrates the successful convergence of the Gradient Descent algorithm. The cost decreases sharply in the initial iterations and then gradually flattens, indicating that the model has found a stable minimum for the cost function within the specified 1000 iterations. The majority of the reduction in cost (MSE) occurs within the first **200-300 iterations**. Beyond this point, the curve flattens into a plateau, indicating that further training yields **diminishing returns** and negligible improvement to the model's performance.

Therefore, while running for 1000 iterations confirms complete convergence, the training process could be stopped much earlier—for instance, around 400 iterations—to achieve a nearly identical result while being significantly **more computationally efficient**.

The Actual vs. Predicted plot (Figure 2) shows a **positive linear relationship** between the model's predictions and the true values, as the data points (blue dots) generally follow the direction of the 45-degree "Perfect Prediction" line. However, the significant spread of these points around the line illustrates the model's **notable prediction errors**, which is consistent with a moderate R^2 score.

The addition of the green "Best Fit Line," which shows the actual trend of the predictions, reveals a more specific bias in the model. This green line is visibly **less steep than the red reference line**. This indicates that the model tends to:

- **Underestimate the price of expensive homes** (the green line is below the red line for high actual values).
- **Slightly overestimate the price of inexpensive homes** (the green line is above the red line for low actual values).

Furthermore, the plot highlights a key data limitation. The vertical cluster of predictions for homes with an actual value of 5.0 shows that the model struggles significantly with this **price-capping effect** in the dataset, contributing to poor accuracy for the most expensive properties.

The experimenting with learning rate(α) (Figure 3) compares the training progress for three different learning rates, highlighting why the choice of this hyperparameter is crucial.

- **"Too High" (α=0.5):** The blue line shows that a high learning rate causes the cost to decrease for a few iterations before exploding upwards. This behavior is known as **divergence**. The steps the algorithm takes are so large that they repeatedly overshoot the minimum of the cost function, leading to a complete failure to learn.
- **"Too Low" (α=0.0001):** The green line illustrates **slow convergence**. While the cost does decrease, it does so very gradually. After 1000 iterations, the model is still far from the optimal solution found by the "Just Right" rate. This makes the training process highly inefficient.
- **"Just Right" (α=0.01):** The orange line represents an **optimal learning rate** for this problem. It achieves a balance between speed and stability, decreasing the cost rapidly in the early stages and then smoothly settling at the lowest point.

In conclusion, this experiment confirms that an appropriate learning rate is essential for training the model effectively.

5. Conclusion:

This assignment successfully demonstrated the from-scratch implementation of a Linear Regression model using Gradient Descent. The final model achieved an R-squared score of **0.5768** on the California Housing dataset, indicating a moderately effective predictive performance.

The analysis confirmed successful model convergence via the learning curve and highlighted the critical impact of the learning rate hyperparameter on training. While the linear model served as a strong baseline, its limitations were evident in the unexplained variance. This project provided key practical insights into the fundamentals of regression, model evaluation, and hyperparameter tuning.

Future work could involve applying more advanced techniques like **regularization**(L1/L2) to improve predictive accuracy.

---

6. Resources Used

1. Scikit-learn documentation for dataset loading and preprocessing.
2. Pandas documentation for understanding data statistics.
3. Matplotlib documentation for plotting graphs.
4. W3Schools for OOPs concepts.
5. GeeksforGeeks for Gradient Descent implementation.