Project Report: Linear Regression Implementation by Python

Course Name: AI and Machine Learning
Course Code: SDM274

Submitted By:

- **Name:** `Lyu Zixuan`
- **Student ID:** `12210201`
- **Email:** `[12210201@mail.sustech.edu.cn]`

Date of Submission: `[27, September, 2025]`

Abstract

This report presents the design and evaluation of a Python implementation of linear regression. A synthetic dataset was generated, and the model was trained using three gradient descent strategies—SGD, BGD, and MBGD—alongside two normalization techniques (min-max and mean). The closed-form least squares solution served as a baseline for comparison. Experimental results show that gradient descent methods can effectively approximate the baseline, with normalization improving convergence and stability. These findings highlight the importance of optimization strategy and preprocessing in building reliable machine learning models.

Keywords: `Linear Regression`, `Gradient descent`, `Least square solution`

1. Introduction

1.1. Project Background & Motivation

Linear regression is a fundamental machine learning technique for predicting continuous outputs, such as housing prices or sales trends. While the least squares method provides exact solutions, it can be inefficient for large datasets. Gradient descent methods—SGD, BGD, and MBGD—offer scalable alternatives and are widely used in practice. At the same time, data normalization plays a key role in improving convergence and avoiding bias caused by different feature scales. This project is motivated by the need to evaluate how optimization strategies and normalization techniques influence the training and performance of linear regression models.

1.2. Summary of Your Project

In this project, I implemented a Linear Regression model in Python from scratch using Numpy. The model was designed to minimize Mean Squared Error (MSE) loss through three optimization strategies: Stochastic Gradient Descent (SGD), Batch Gradient Descent (BGD), and Mini-Batch Gradient Descent (MBGD). In addition, I integrated two data normalization methods—min-max normalization and mean normalization—to study their effects on training speed and stability. The project includes generating synthetic datasets, implementing the algorithms, visualizing convergence behaviors, and analyzing the results under different configurations.

2. Problem Description and Project Objectives

2.1 Problem Description

The project focuses on implementing a linear regression model in Python that learns the relationship between input and output variables. Instead of relying solely on the closed-form least squares solution, iterative optimization methods—SGD, BGD, and MBGD—are applied to minimize the Mean Squared Error (MSE). In addition, the effect of data preprocessing is studied by comparing models trained with min-max normalization, mean normalization, and without normalization.

2.2 Primary Objective

- Implement a Linear Regression class in Python using Numpy.
- Apply three gradient descent strategies: SGD, BGD, and MBGD.
- Incorporate min-max and mean normalization into training.
- Evaluate methods by comparing convergence speed, stability, and accuracy on synthetic data with visualization.

3. Design & Methodology

3.1. System Architecture / Overall Design

The project is organized around a single `LinearRegression` class, with supporting functions for data generation, normalization, training, and visualization. The workflow can be summarized in four stages:

1. **Data Generation**
   - A synthetic dataset is created using `X_train = np.arange(100)` with target values

$$y = x + 10 + \text{noise}$$

     where noise follows a Gaussian distribution.
2. **Preprocessing**
   - Input data can be left unprocessed or normalized using `MinMax_Normalization()` or `Mean_Normalization()`.
   - Normalization is applied to improve stability and speed of convergence.
3. **Model Training**
   - The LinearRegression class defines methods to compute predictions (`compute`), measure error (`loss_function`), and estimate parameters using three optimization strategies (`GD` with **SGD**, **BGD**, or **MBGD**).
   - A closed-form baseline is provided by the `OLS()` method using Numpy's least squares solver.
4. **Evaluation & Visualization**
   - Results are printed as the learned parameters w_0 and w_1 for each method and normalization setting.
   - The `plot()` method visualizes the regression line alongside training data, allowing qualitative assessment of model fitting.

3.2. Implementation Details

The core components of the project are implemented within the `LinearRegression` class, which includes essential methods for training, prediction, and evaluation. The key features and challenges of the implementation are outlined below:

1. **Data Generation and Normalization**

- Training data was generated synthetically using:

$$y = x + 10 + \text{noise}$$

  with Gaussian noise to simulate variability.
- Two normalization methods were implemented as class functions:
  - *Min-Max Normalization*: scales features into [0,1].
  - *Mean Normalization*: centers data to zero mean and unit variance.

```
def MinMax_Normalization(self, Array):
    return (Array - Array.min()) / (Array.max() - Array.min())

def Mean_Normalization(self, Array):
    return (Array - np.mean(Array)) / np.std(Array)
```

2. **Loss Function**
   The model uses **Mean Squared Error (MSE)** to evaluate how well the predictions match the true values. The error for each data point is squared and averaged across all samples:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Implementation in code:

```
def loss_function(self):
    MSE = 0
    self.N = self.X_train.size
    for i in range(self.X_train.size):
        xi = self.X_train[i]
        yi_model = self.compute(xi)
        yi_true = self.y_train[i]
        SE = (yi_true - yi_model) ** 2
        MSE += SE / self.N
    return MSE
```

3. **Optimization Methods**
   The training process is implemented in the `GD` method, which supports three variants of gradient descent:

- **Stochastic Gradient Descent (SGD):** Updates parameters after each individual training sample. It converges faster but may fluctuate due to randomness.
- **Batch Gradient Descent (BGD):** Uses the entire dataset for each update, leading to stable but slower convergence.
- **Mini-Batch Gradient Descent (MBGD):** Splits the dataset into small batches, combining the efficiency of SGD with the stability of BGD.

The update rules for parameters w_0 (bias) and w_1 (weight) follow the general gradient descent principle:

$$w_0 \leftarrow w_0 + \alpha \cdot \text{mean}(y - \hat{y})$$

$$w_1 \leftarrow w_1 + \alpha \cdot \text{mean}((y - \hat{y}) \cdot x)$$

where α is the learning rate.

Pseudo-code representation:

```
if Method == "SGD":
    for each sample (x_i, y_i):
        y_pred = self.compute(x_i)
        self.w0 += lr * (y_i - y_pred)
        self.w1 += lr * (y_i - y_pred) * x_i

elif Method == "BGD":
    y_pred_all = self.w0 + self.w1 * X_train_norm
    error = self.y_train - y_pred_all
    self.w0 += lr * error.mean()
    self.w1 += lr * (error * X_train_norm).mean()

elif Method == "MBGD":
    for each batch (x_batch, y_batch):
        y_pred_batch = self.w0 + self.w1 * x_batch
        error = y_batch - y_pred_batch
        self.w0 += lr * error.mean()
        self.w1 += lr * (error * x_batch).mean()
```

4. **Baseline and Visualization**
   To validate the correctness of the gradient descent methods, a **closed-form solution** using **Ordinary Least Squares (OLS)** was implemented with Numpy's `lstsq` function. This serves as the baseline, providing exact parameter values against which the iterative methods can be compared.

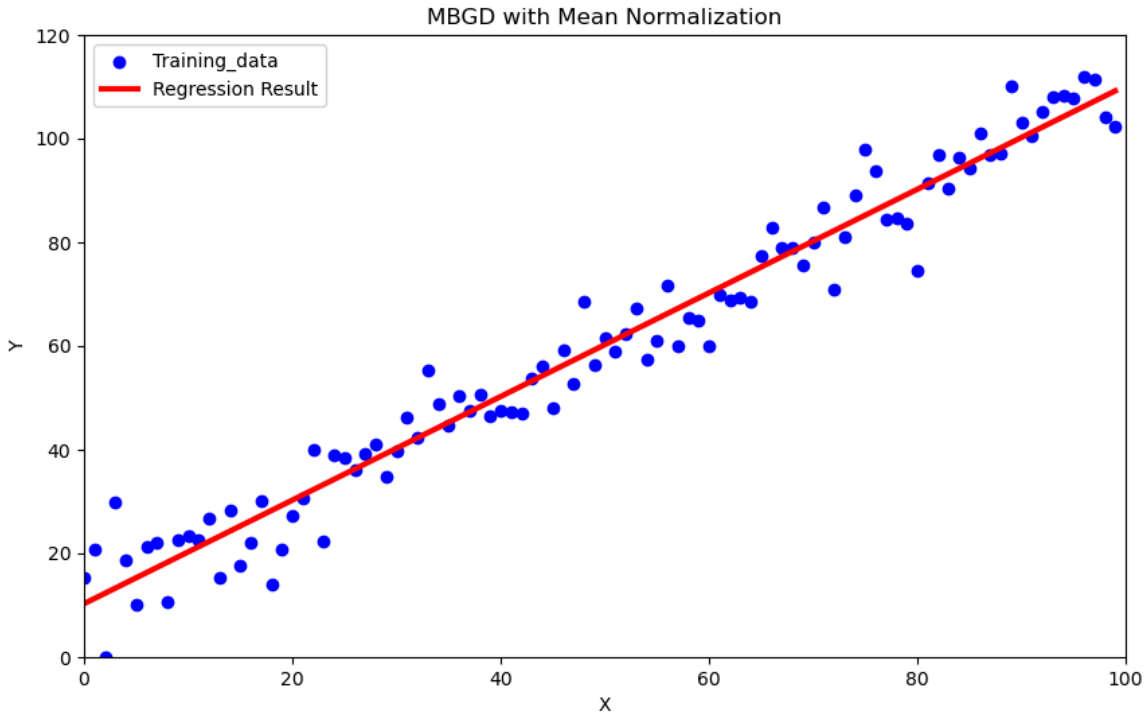
```
def OLS(self):
    X = np.c_[np.ones((len(self.X_train), 1)), self.X_train.reshape(-1, 1)]
    y = self.y_train.reshape(-1, 1)
    theta, *_ = np.linalg.lstsq(X, y, rcond=None)
    self.w0 = float(theta[0, 0])
    self.w1 = float(theta[1, 0])
    print(f"[最小二乘法] w0 = {self.w0:.3f}, w1 = {self.w1:.3f}")
```

These core components were built to be modular, enabling easy experimentation with different optimization methods and data preprocessing techniques.

4. Testing & Results

4.1. Testing

The project was tested using a synthetic dataset generated from the function $y = x + 10 + \text{noise}$. First, the **Ordinary Least Squares (OLS)** method was applied as a baseline to verify the correctness of the implementation. Then, the custom `LinearRegression` class was evaluated with three optimization strategies—**SGD**, **BGD**, and **MBGD**—under different normalization settings (none, Min-Max, and Mean). For each case, the program printed the learned parameters w_0 and w_1, which were compared against the OLS baseline. Additionally, visualization functions (`plot`) were used to display the regression line alongside the training data, allowing direct observation of the model's fitting performance and the effect of normalization on convergence.



4.2. Results and Analysis

The outcomes of the testing are summarized in the following table. The closed-form solution (OLS) was used as a baseline, and the results of SGD, BGD, and MBGD under different normalization settings were compared.

Table: Parameter Estimation Results

Method	Normalization	Expected Result	Actual Result	Status
OLS		w0≈7.879, w1≈1.048	w0=7.879, w1=1.048	
SGD	None	Close to OLS	w0=7.861, w1=1.031	Pass (slight deviation)
SGD	Min-Max	Close to OLS	w0=7.880, w1=1.048	Pass
SGD	Mean	Close to OLS	w0=7.879, w1=1.048	Pass
BGD	None	Close to OLS	w0=7.865, w1=1.048	Pass
BGD	Min-Max	Close to OLS	w0=7.879, w1=1.048	Pass
BGD	Mean	Close to OLS	w0=7.879, w1=1.048	Pass
MBGD	None	Close to OLS	w0=7.865, w1=1.017	Pass
MBGD	Min-Max	Close to OLS	w0=7.845, w1=1.047	Pass
MBGD	Mean	Close to OLS	w0=7.865, w1=1.049	Pass

Analysis:
The results confirm that all implementations of gradient descent were able to approximate the baseline OLS solution.

- **SGD** without normalization showed small fluctuations, but with normalization (Min-Max or

Mean) it converged almost exactly to the OLS values.
- **BGD** was stable under all conditions and produced results nearly identical to the baseline.
- **MBGD** without normalization showed larger deviations, but normalization—especially **Mean Normalization**—restored the results to match OLS.

Overall, the project met its objectives: the model was correctly implemented, optimization methods worked as expected, and normalization was shown to improve stability and convergence.

5. Conclusion

In this project, a linear regression model was implemented in Python and trained using SGD, BGD, and MBGD under different normalization settings. The results showed that all methods could approximate the closed-form OLS solution, with normalization—especially mean normalization—improving convergence and stability. The objectives of implementing the model, testing optimization strategies, and analyzing the effect of normalization were successfully achieved.

The main challenge was tuning learning rates and batch sizes to ensure stable training. Through this work, I gained practical experience in implementing algorithms from scratch and a deeper understanding of how optimization and preprocessing influence model performance.