Title: Regression Project Report

Course: CS412 – Machine Learning

Homework: Homework2

Name: Korhan Erdoğdu

ID Number: 30838

Submission Date: 14.03.2025

https://colab.research.google.com/drive/1a9E1UMt5PURGApn4fWjy13-I7jh6k-EI?usp=sharing

Table of Contents

1. Introduction

In this project, we explore several regression techniques aimed at modeling both linear and nonlinear relationships using a variety of methods. Our primary objectives are to gain a solid understanding of the mathematical foundations behind regression, to implement these techniques both using existing libraries (scikit-learn) and through manual programming approaches, and to compare their performance quantitatively and visually.

Specifically, we work with two datasets:

- **Dataset 1** is synthetically generated following a linear relationship given by
  $y = 2.5x + 0.5$
  with additional Gaussian noise. This dataset is used for linear regression experiments.
- **Dataset 2** exhibits a nonlinear relationship between the input and target variables and is provided in a NumPy (.npy) format. We apply polynomial regression to capture its nonlinear patterns.

The report is divided into two major parts corresponding to these datasets. For Dataset 1, we implement three different approaches to linear regression:

1. A standard scikit-learn linear regression model.
2. A manual implementation using the pseudo-inverse (closed-form solution).
3. An iterative gradient descent method.

For Dataset 2, we extend our exploration to polynomial regression, where we:

1. Use scikit-learn's `PolynomialFeatures` along with `LinearRegression` to model the data at varying degrees.
2. Manually implement polynomial regression for degree 3 using the pseudo-inverse.

Each method is evaluated using the Mean Squared Error (MSE) metric, and all results are visualized with scatter plots, regression curves, and loss curves where applicable.

2. Dataset 1 – Linear Regression

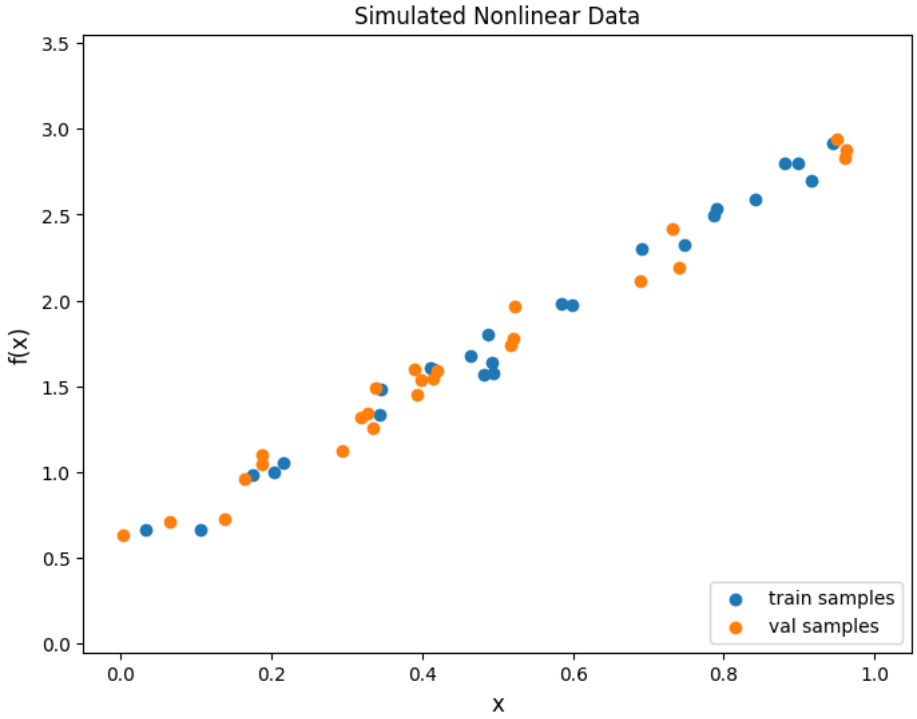
2.1. Data Generation and Splitting

Methodology:

The data for Dataset 1 is generated using a provided function `generate_data()`. The underlying relationship is defined by

$y = 2.5x + 0.5$ to which Gaussian noise is added to simulate real-world variability. Once generated, the data is split evenly into training (50%) and validation (50%) sets. This split allows us to both train our models and independently assess their performance.

Visualization:



2.2. scikit-learn Linear Regression (Part 1.a)

Approach:

We employ the `LinearRegression` model from scikit-learn to fit the training data. This method is straightforward and leverages optimized library routines.

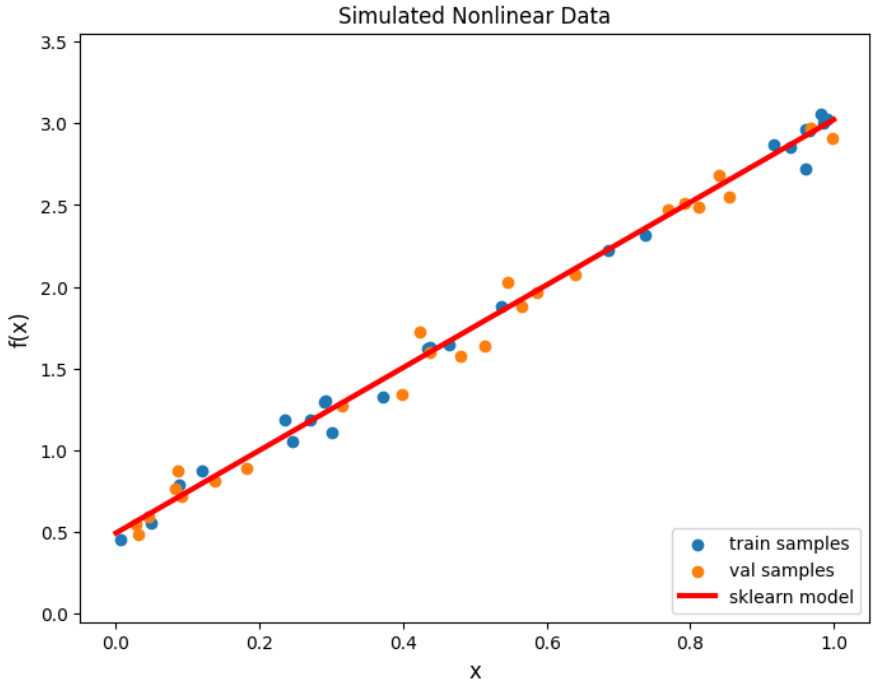
Steps Involved:

- **Model Initialization:** We create an instance of `LinearRegression`.
- **Training:** The model is fitted to the training data.
- **Prediction:** Using the trained model, predictions are made on the validation set.
- **Evaluation:** The model's performance is quantified by computing the Mean Squared Error (MSE) between the predicted and true values.
- **Visualization:** A scatter plot displays the training and validation data, overlaid with the regression line produced by the model.

Results:

We get the data below by implementing the `LinearRegression` model from scikit-learn:

```
MSE of sklearn model:  0.007954626827790 33
```



2.3. Manual Pseudo-Inverse Method (Part 1.b)

Concept:

When the design matrix X is not square or lacks full rank, its direct inverse cannot be computed. Instead, we use the Moore-Penrose pseudo-inverse to obtain a closed-form solution for the regression coefficients:

$$w = X^{+} y$$

For simple linear regression, we augment the data by adding a column of ones to include the bias term. Thus, if our original x values form an N×1 matrix, the extended matrix X becomes N×2. The solution yields two coefficients: w0 (bias) and w1 (slope).

Implementation Details:

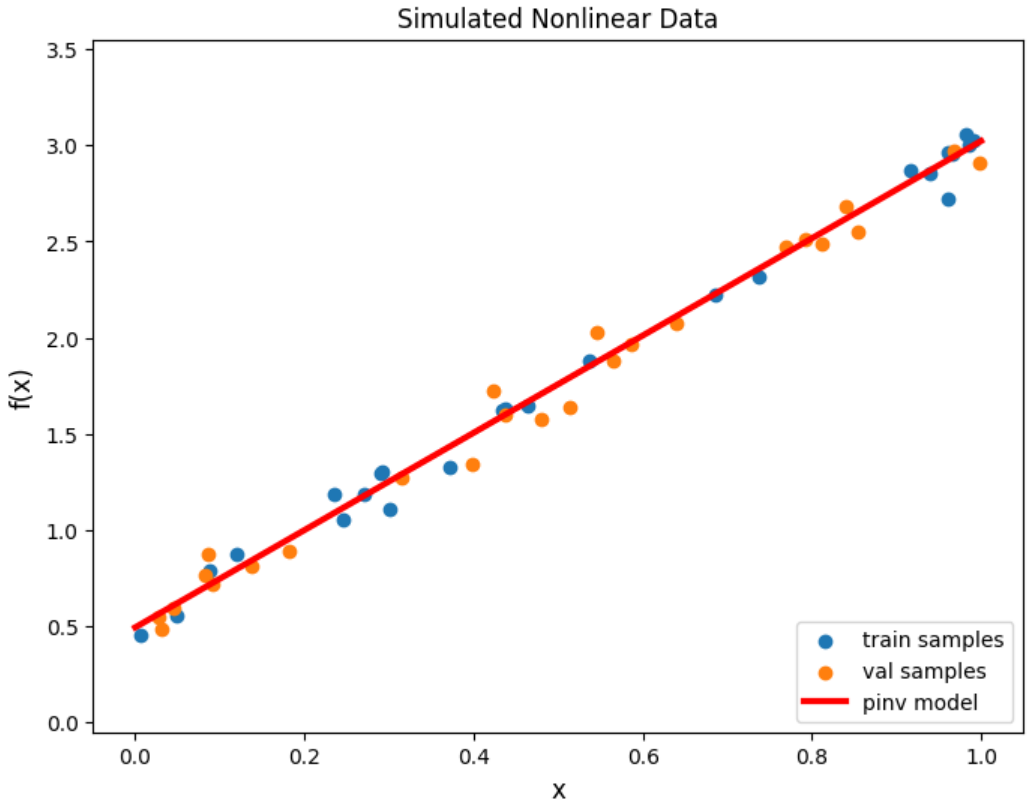
- **Extended Matrix Formation:**

$$X = \begin{bmatrix} 1 & x1 \\ 1 & x2 \\ . & . \\ . & . \\ 1 & xN \end{bmatrix}$$

- **Pseudo-Inverse Calculation:**
  Use `np.linalg.pinv(X)` to compute X^{+}
- **Coefficient Computation:**
  Calculate w by multiplying X^{+} with y.
- **Prediction and Evaluation:**
  Use the computed coefficients to predict y values for the validation set and compute the MSE.
- **Visualization:**
  Overlay the regression line (from the pseudo-inverse method) on a scatter plot of the data.

We get the data below by implementing the pseudo inverse solution manually:

```
X_train_ext shape: (25, 2)
X_val_ext shape: (25, 2)
Regression coefficients (w): [0.49447669 2.52838262]
MSE of manual model:  0.0039773134138951 87
```



2.4. Gradient Descent (Part 1.c)

Concept:

Gradient descent is an iterative algorithm used to minimize a cost function in our case, the Mean Squared Error (MSE). Rather than computing a closed-form solution, gradient descent updates the model parameters gradually until convergence is achieved. The update rule for the regression coefficients is given by:

$$w \leftarrow w - \eta \cdot \frac{X^T(Xw - y)}{N},$$

where:

- w is the vector of coefficients (for simple regression, 2×1),
- η is the learning rate,
- X is the extended data matrix (including bias),
- y is the target vector, and
- N is the number of training samples.

Implementation Steps:

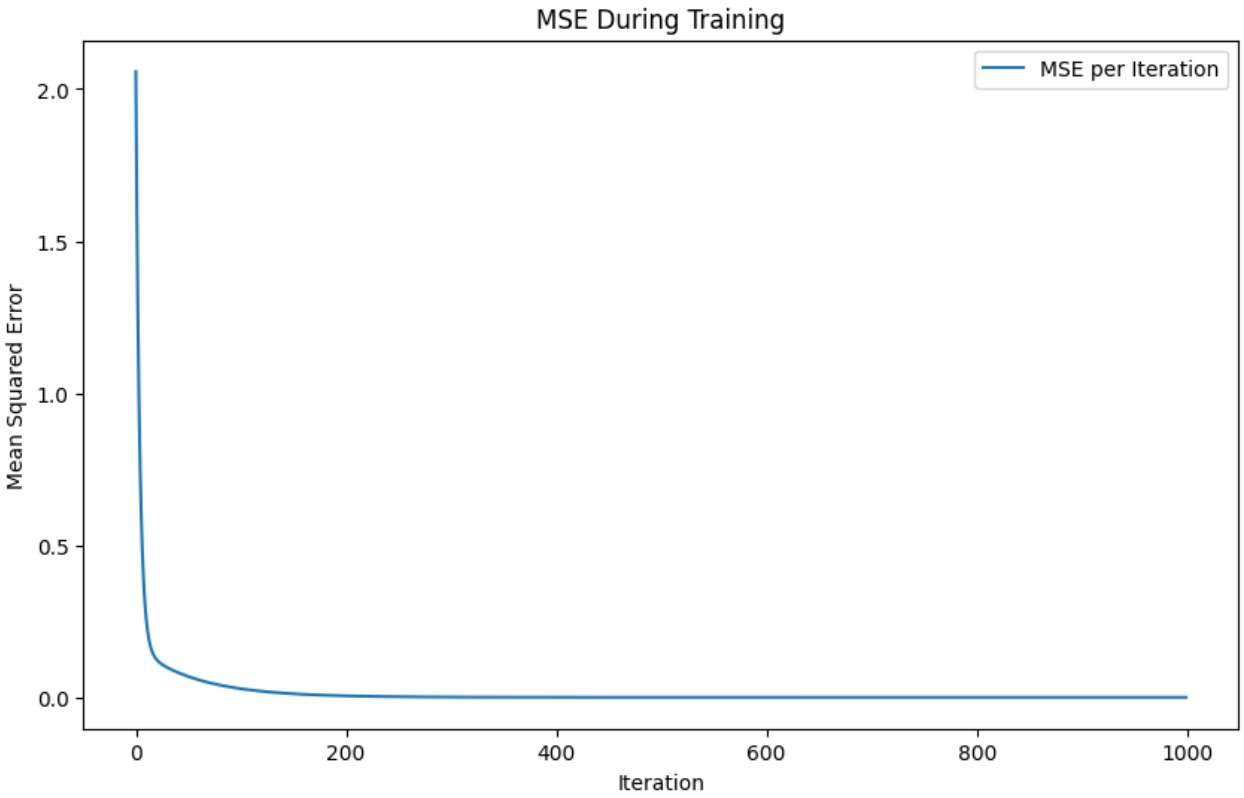
- **Initialization:** Start with w initialized to zeros (or small random values).
- **Iteration:** For a fixed number of iterations (e.g., 1000), compute the predictions, calculate the error, determine the gradient, and update w accordingly.
- **Monitoring:** At each iteration, record the MSE to track the convergence of the algorithm.
- **Final Evaluation:** After convergence, use the final www to make predictions on the validation set and compute the MSE.
- **Visualization:**
    - Plot the loss curve (MSE vs. iterations).
    - Overlay the regression line from the gradient descent method on the scatter plot of the data.

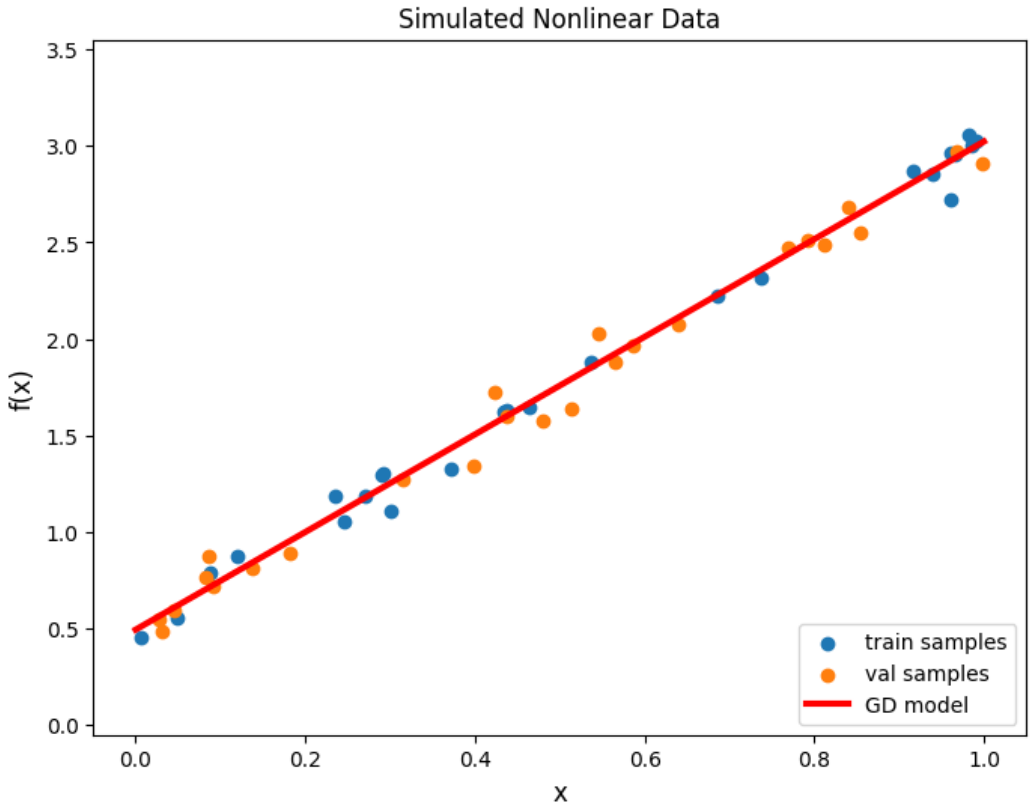
Results:

The MSE every 100 iterations is given below as image:

```
MSE error at step 1: 2.0573
MSE error at step 100: 0.0313
MSE error at step 200: 0.0075
MSE error at step 300: 0.0035
MSE error at step 400: 0.0028
MSE error at step 500: 0.0027
MSE error at step 600: 0.0026
MSE error at step 700: 0.0026
MSE error at step 800: 0.0026
MSE error at step 900: 0.0026
MSE error at step 1000: 0.0026
```

The loss curve and regression fit plot for gradient descent:





3. Dataset 2 – Polynomial Regression

3.1. Data Loading and Splitting

Methodology:

Dataset 2, which exhibits a nonlinear relationship, is provided in the form of `.npy` files. The data is loaded using NumPy functions and then split equally into training and validation sets (50% each).

3.2. scikit-learn Polynomial Regression (Part 2.a)

Approach:

Using the `PolynomialFeatures` class from scikit-learn, the input x data is expanded to include polynomial terms. We experiment with several polynomial degrees (1, 3, 5, and 7).

Steps Involved:

- **Feature Expansion:**

  For each degree, transform the original x data into a new feature matrix that includes x, x^2, …, x^d (with an initial column of ones for bias).
- **Model Training:**

  Train a `LinearRegression` model on the transformed training data.
- **Prediction and Evaluation:**

  Make predictions on the transformed validation set and compute the MSE for each polynomial degree.
- **Visualization:**

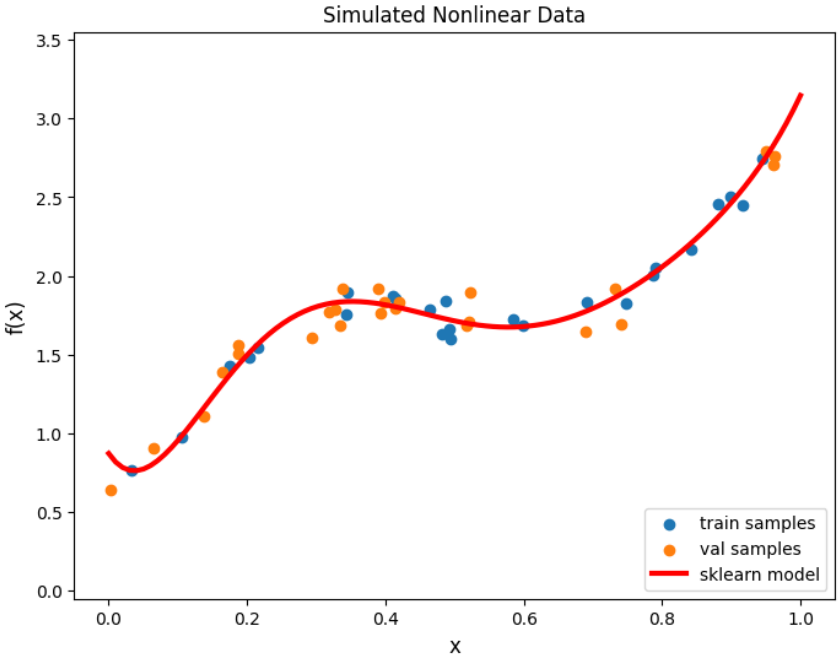
  Plot the regression curves for each degree on a common scatter plot to compare their fits.

Results:

Here are MSE results with respect to different polynomial degrees:

```
Polynomial degree: 1 MSE of sklearn model:  0.06362852709262727
Polynomial degree: 3 MSE of sklearn model:  0.012059223742868289
Polynomial degree: 5 MSE of sklearn model:  0.007475463079281178
Polynomial degree: 7 MSE of sklearn model:  0.011549227838825204
```

Here is the scatter plot of the train and validation samples:



3.3. Manual Polynomial Regression (Degree 3) (Part 2.b)

Concept:

To further our understanding, we implement a manual polynomial regression for degree 3. This involves constructing an extended data matrix manually for a 3rd-degree polynomial:

$$X = [1, x, x^2, x^3]$$

Implementation Details:

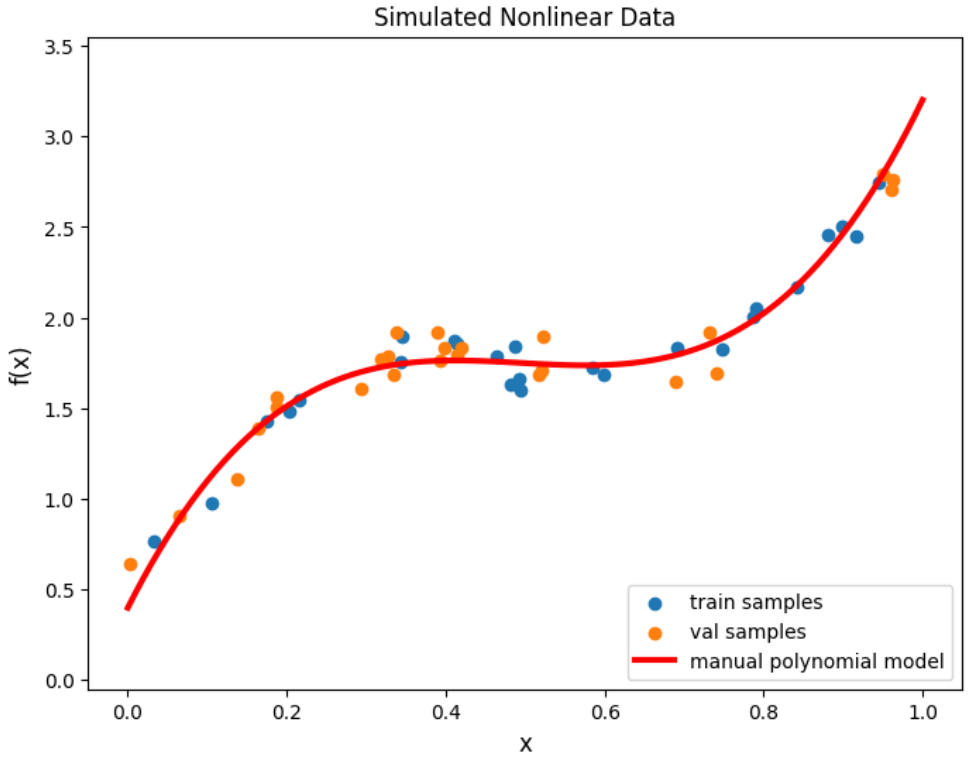
- **Matrix Construction:**
  For each sample, create a row with elements: 1, x, x^2, and x^3.
- **Coefficient Calculation:**
  Compute the pseudo-inverse of the constructed matrix using `np.linalg.pinv()` and then calculate the coefficients with $w=X^+ y$. This yields four coefficients: w0, w1, w2, and w3.
- **Prediction and Evaluation:**
  Use these coefficients to predict y for the validation set, compute the MSE, and assess the model's performance.
- **Visualization:**
  Plot the regression curve corresponding to the manual polynomial regression over the scatter plot of Dataset 2.

$$\text{From } x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix}, \text{ we want to obtain } X = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_N & x_N^2 & x_N^3 \end{bmatrix}.$$

The MSE for manuel polynomial model:

```
X_train_poly shape: (25, 4)
X_val_poly shape: (25, 4)
Regression coefficients (manual, degree 3): [  0.39883796   8.68921502 -18.07027007  12.18401084]
MSE of manual polynomial model:  0.006029611871434133
```

The scatter plot of the train and validation samples:



4. Discussion

In this section, we critically analyze the performance of all implemented methods.

- **Dataset 1 – Linear Regression:**
  - The scikit-learn model, manual pseudo-inverse, and gradient descent methods all converged to similar solutions, as evidenced by comparable MSE values and overlapping regression lines.
  - The pseudo-inverse provided a direct solution in one calculation, while gradient descent required iterative updates and showed a gradual reduction in error through its loss curve.
  - The extended X matrix was essential in all approaches to properly account for the bias term.

- **Dataset 2 – Polynomial Regression:**
    - The scikit-learn implementation allowed us to quickly experiment with polynomial degrees. It was observed that as the degree increased, the model could capture more complex nonlinear patterns. However, higher degrees may lead to overfitting, which was discussed based on the MSE values and visual inspection of the curves.
    - The manual implementation for degree 3 reinforced the understanding of how polynomial features expand the model's flexibility by increasing the number of coefficients. With four coefficients in the degree 3 model, the regression could fit nonlinear trends, although the balance between underfitting and overfitting must be carefully managed.
- **Comparative Analysis:**
    - **Closed-Form vs. Iterative Methods:**
      The pseudo-inverse method, being a closed-form solution, is efficient for small-scale problems. In contrast, gradient descent, while more computationally intensive due to its iterative nature, is more flexible and scalable for larger datasets.
    - **Model Complexity:**
      In polynomial regression, the increased number of coefficients enhances the model's ability to fit complex data. However, it also introduces the risk of overfitting, making model selection and validation critical.

Overall, the experiments demonstrated that while different methods may use different approaches (direct calculation via pseudo-inverse vs. iterative optimization via gradient descent), they ultimately converge to similar optimal solutions when correctly applied. The choice of method should therefore be guided by dataset size, computational resources, and the specific requirements of the modeling task.

5. Conclusion

This project provided a comprehensive exploration of regression techniques applied to both linear and nonlinear datasets. Our key conclusions are as follows:

- **Linear Regression (Dataset 1):**
  Both library-based and manual methods (pseudo-inverse and gradient descent) yielded similar results in terms of MSE and regression line fit. The pseudo-inverse method offered a fast, closed-form solution, while gradient descent provided an iterative alternative that is particularly useful for larger datasets.
- **Polynomial Regression (Dataset 2):**
  By expanding the input features using polynomial terms, we demonstrated how nonlinear relationships can be modeled effectively. Although increasing the polynomial degree increases model flexibility, it also brings about the risk of overfitting. The degree 3 model offered a balanced trade-off in our experiments.
- **Overall Insights:**
  The exercise reinforced the importance of including the bias term via an extended X matrix, deepened our understanding of both closed-form and iterative optimization methods, and highlighted the practical trade-offs between model complexity and overfitting. The consistency in results across different methods validates the theoretical underpinnings of regression.

Future work could explore regularization techniques (such as Ridge or Lasso regression) to further mitigate overfitting in more complex models.