Regression Tutorial

Regression is a fundamental technique in **Machine Learning** used to model relationships between variables. It is widely applied to predict outcomes based on input data.

Regression can be broadly categorized into:

1. **Linear Regression**: Assumes a linear relationship between variables.
2. **Polynomial Regression**: Captures non-linear relationships by introducing polynomial terms.

---

1. Linear Regression

Linear regression assumes a **straight-line relationship** between the independent variable(s) and the target variable.

Linear Regression Model

The general equation for a linear regression model is:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

Where:

- **$ \hat{y} $**: The predicted value.
- **$ n $**: The number of features.
- **$ x_i $**: The i-th feature value.
- **$ \theta_j $**: The j-th model parameter (including the bias term $ \theta_0 $ and the feature weights $ \theta_1, \theta_2, \dots, \theta_n $).

Simple Linear Regression

When there is only one feature ($ n = 1 $), the equation simplifies to:

$$\hat{y} = \theta_0 + \theta_1 x_1$$

- **Practical Example**:
  A company uses **experience** as the independent variable to predict **salary**.
  - **$ x_1 $**: Experience

- **$ \hat{y} $**: Predicted Salary
- **Goal**: Understand how experience impacts salary.

Multiple Linear Regression

When there are multiple features ($ n > 1 $), the equation becomes:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

- **Practical Example**:
  A real estate agency uses **property features** to predict the **house price**.
  - **$ x_1 $**: Square footage
  - **$ x_2 $**: Number of bedrooms
  - **$ x_3 $**: Location rating
  - **$ \hat{y} $**: Predicted House Price
  - **Goal**: Analyze how multiple factors affect housing prices.

---

2. Polynomial Regression

Polynomial regression extends linear regression to model **non-linear relationships** between variables. It transforms the features into polynomial terms.

The equation for Polynomial Regression can be like:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^3 + \cdots + \theta_n x_1^n$$

Where:

- **$ \hat{y} $**: Predicted Value.
- **$ x_1, x_1^2, x_1^3, \dots $**: Polynomial terms.
- **$ \theta_0, \theta_1, \theta_2, \dots $**: Model parameters.
- **Practical Example**:
  Predicting the **salary** based on **position level**.

- **$ x_1 $**: Position Level
- **$ \hat{y} $**: Salary
- **Goal**: Use a polynomial regression model to predict the non-linear growth pattern.

Comparison of Linear and Polynomial Regression

Aspect	Linear Regression	Polynomial Regression
Nature of Relationship	Assumes a straight-line relationship	Models non-linear relationships (curved)
Complexity	Simple and interpretable	More complex due to polynomial terms
Use Cases	When data follows a linear trend	When data shows curves or non-linear trends

- **Linear Regression** is suitable for data with a linear relationship, like predicting sales based on advertising.
- **Polynomial Regression** is ideal when the relationship is non-linear, such as growth patterns over time.

Understanding the type of relationship in the data helps choose the correct regression technique to make accurate predictions.

Methods for Finding Optimal Model Parameters

When training a machine learning model, there are two primary approaches to find the model parameters that minimize the cost function over the training set:

1. Closed-Form Solution

- Directly computes the model parameters that best fit the training set.
- Finds the parameters that **minimize the cost function** using a mathematical formula.

---

2. Iterative Optimization

An optimization approach that **gradually tweaks the model parameters** to minimize the cost function:

Gradient Descent (GD)

- Iteratively adjusts the model parameters in the direction that reduces the cost function.
- **Goal**: Minimize the cost function over the training set.

Variants of Gradient Descent

Gradient Descent has three main variants based on the amount of data used in each step:

1. **Batch Gradient Descent**
   - Uses the **entire training set** to compute the gradient at each step.
2. **Mini-Batch Gradient Descent**
   - Splits the training set into **small batches** and computes the gradient for each batch.
3. **Stochastic Gradient Descent (SGD)**
   - Uses **one training example** at a time to compute the gradient.

Derivation of the Simple Linear Regression Formula

Simple Linear Regression (SLR) aims to model the relationship between one independent variable x and one dependent variable y. The relationship is represented as:

$$\hat{y} = \theta_0 + \theta_1 x$$

Where:

- $\hat{y}$: Predicted value (dependent variable)
- x: Independent variable (predictor)
- θ_0: Intercept (value of $\hat{y}$ when $x = 0$)
- θ_1: Slope (rate of change of $\hat{y}$ with respect to x)

The goal is to determine the values of the slope θ_1 and intercept θ_0 that minimize the **sum of squared residuals (errors)**. Let's derive the formula step by step.

---

1. Sum of Squared Residuals

The **residual (error)** for each data point is the difference between the observed value y_i and the predicted value $\hat{y}_i$:

$$e_i = y_i - \hat{y}_i$$

The predicted value $\hat{y}_i$ is given as:

$$\hat{y}_i = \theta_0 + \theta_1 x_i$$

Thus, the residual becomes:

$$e_i = y_i - (\theta_0 + \theta_1 x_i)$$

The objective is to minimize the **Sum of Squared Errors (SSE)**:

$$SSE = \sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} \left(y_i - (\theta_0 + \theta_1 x_i) \right)^2$$

---

2. Deriving the Slope (θ_1)

To minimize **SSE**, we differentiate the **SSE** with respect to θ_0 and θ_1, and set the derivatives to zero.

Step 1: Partial Derivative with Respect to θ_0

Taking the derivative of SSE with respect to θ_0:

$$\frac{\partial SSE}{\partial \theta_0} = -2 \sum_{i=1}^{n} \left(y_i - \theta_0 - \theta_1 x_i \right)$$

Set this derivative to zero:

$$\sum_{i=1}^{n} \left(y_i - \theta_0 - \theta_1 x_i \right) = 0$$

Rearranging, we get:

$$\sum y_i = n\theta_0 + \theta_1 \sum x_i$$

Step 2: Partial Derivative with Respect to θ_1

Taking the derivative of SSE with respect to θ_1:

$$\frac{\partial SSE}{\partial \theta_1} = -2\sum_{i=1}^{n} x_i\left(y_i - \theta_0 - \theta_1 x_i\right)$$

Set this derivative to zero:

$$\sum_{i=1}^{n} x_i\left(y_i - \theta_0 - \theta_1 x_i\right) = 0$$

Expanding and rearranging:

$$\sum x_i y_i = \theta_0 \sum x_i + \theta_1 \sum x_i^2$$

---

3. Solving for θ_0 and θ_1

We now have two equations:

1. $ \sum y_i = n\theta_0 + \theta_1 \sum x_i $
2. $ \sum x_i y_i = \theta_0 \sum x_i + \theta_1 \sum x_i^2 $

Solving for θ_1 (Slope)

First, substitute θ_0 from the first equation into the second equation.

From the first equation, we can express θ_0 as:

$$\theta_0 = \frac{\sum y_i - \theta_1 \sum x_i}{n}$$

Substitute this expression for θ_0 into the second equation:

$$\sum x_i y_i = \left(\frac{\sum y_i - \theta_1 \sum x_i}{n}\right)\sum x_i + \theta_1 \sum x_i^2$$

Multiply through by n to eliminate the denominator:

$$n\sum x_i y_i = \left(\sum y_i - \theta_1 \sum x_i\right)\sum x_i + n\theta_1 \sum x_i^2$$

Expand the right-hand side:

$$n\sum x_i y_i = \sum y_i \sum x_i - \theta_1\left(\sum x_i\right)^2 + n\theta_1 \sum x_i^2$$

Rearrange terms to isolate θ_1:

$$n\sum x_i y_i - \sum y_i \sum x_i = \theta_1 \left(n\sum x_i^2 - \left(\sum x_i\right)^2\right)$$

Solve for θ_1:

$$\theta_1 = \frac{n\sum x_i y_i - \sum y_i \sum x_i}{n\sum x_i^2 - \left(\sum x_i\right)^2}$$

This can be rewritten in terms of the covariance and variance:

$$\theta_1 = \frac{\sum \left(x_i - \acute{x}\right)\left(y_i - \acute{y}\right)}{\sum \left(x_i - \acute{x}\right)^2}$$

Where:

- $\acute{x}$: Mean of x values
- $\acute{y}$: Mean of y values

The numerator represents the **covariance** between x and y, and the denominator represents the **variance** of x.

Solving for θ_0 (Intercept)

Once we have θ_1, the intercept θ_0 can be calculated using:

$$\theta_0 = \acute{y} - \theta_1 \acute{x}$$

---

4. Final SLR Formula

The final Simple Linear Regression equation is:

$$\hat{y} = \theta_0 + \theta_1 x$$

Where:

- θ_1 **(Slope)**:

$$\theta_1 = \frac{\sum \left(x_i - \acute{x}\right)\left(y_i - \acute{y}\right)}{\sum \left(x_i - \acute{x}\right)^2}$$

- θ_0 **(Intercept)**:

$$\theta_0 = \acute{y} - \theta_1 \acute{x}$$

Summary

1. **Slope** θ_1 measures the rate of change of $\hat{y}$ with respect to x (calculated as **covariance** of x and y divided by the **variance** of x).

2. **Intercept** θ_0 determines the value of $\hat{y}$ when $x=0$.

3. Together, θ_0 and θ_1 define the best-fit line that minimizes the sum of squared residuals.

The derived formulas allow us to find the optimal regression line for any given set of data.

Example of Simple Linear Regression

Let's consider a real-world example of predicting a student's **exam score** based on the number of hours they study.

Problem Statement

We want to model the relationship between:

- **Independent Variable** (x): Hours of study
- **Dependent Variable** (y): Exam score

The goal is to find the equation of the best-fit line:

$$\hat{y}=\theta_0+\theta_1 x$$

Where:

- $\hat{y}$: Predicted exam score
- x: Hours of study (independent variable)
- θ_0: Intercept
- θ_1: Slope (rate of change of $\hat{y}$ with respect to x)

Data

Hours of Study (x)	Exam Score (y)
1	50
2	55
3	65
4	70

Hours of Study (x)	Exam Score (y)
5	80

Step 1: Calculate the Mean

1. **Mean of x (Hours of Study):**

$$\acute{x} = \frac{\sum x_i}{n} = \frac{1+2+3+4+5}{5} = 3$$

1. **Mean of y (Exam Score):**

$$\acute{y} = \frac{\sum y_i}{n} = \frac{50+55+65+70+80}{5} = 64$$

Step 2: Calculate the Slope (θ_1)

The formula for slope θ_1 is:

$$\theta_1 = \frac{\sum \left(x_i - \acute{x}\right)\left(y_i - \acute{y}\right)}{\sum \left(x_i - \acute{x}\right)^2}$$

x_i	y_i	x_i - $\acute{x}$	y_i - $\acute{y}$	(x_i - $\acute{x}$)(y_i - $\acute{y}$)	(x_i - $\acute{x}$)❑^{2}
1	50	-2	-14	28	4
2	55	-1	-9	9	1
3	65	0	1	0	0
4	70	1	6	6	1
5	80	2	16	32	4

1. **Sum of $\left(x_i - \acute{x}\right)\left(y_i - \acute{y}\right)$:**

$$\sum \left(x_i - \acute{x}\right)\left(y_i - \acute{y}\right) = 28+9+0+6+32 = 75$$

1. **Sum of $\left(x_i - \acute{x}\right)^2$:**

$$\sum \left(x_i - \acute{x}\right)^2 = 4+1+0+1+4 = 10$$

Thus, the slope θ_1 is:

$$\theta_1 = \frac{75}{10} = 7.5$$

Step 3: Calculate the Intercept (θ_0)

The formula for the intercept θ_0 is:

$$\theta_0 = \acute{y} - \theta_1 \acute{x}$$

Substitute the values:

$$\theta_0 = 64 - (7.5 \times 3) = 64 - 22.5 = 41.5$$

Step 4: Final Equation

The equation of the regression line is:

$$\hat{y} = 41.5 + 7.5\,x$$

Interpretation

- **Slope ($\theta_1 = 7.5$)**: For every additional hour of study, the exam score increases by 7.5 points.
- **Intercept ($\theta_0 = 41.5$)**: If a student studies for 0 hours, their predicted exam score is 41.5.

Prediction Example

If a student studies for 6 hours ($x = 6$), the predicted score is:

$$\hat{y} = 41.5 + 7.5(6) = 41.5 + 45 = 86.5$$

Thus, the predicted exam score is **86.5**.

Summary

1. The relationship between study hours and exam scores was modeled using **Simple Linear Regression**.
2. The final regression equation is:

$$\hat{y} = 41.5 + 7.5\,x$$

Implementation of SLR with formula

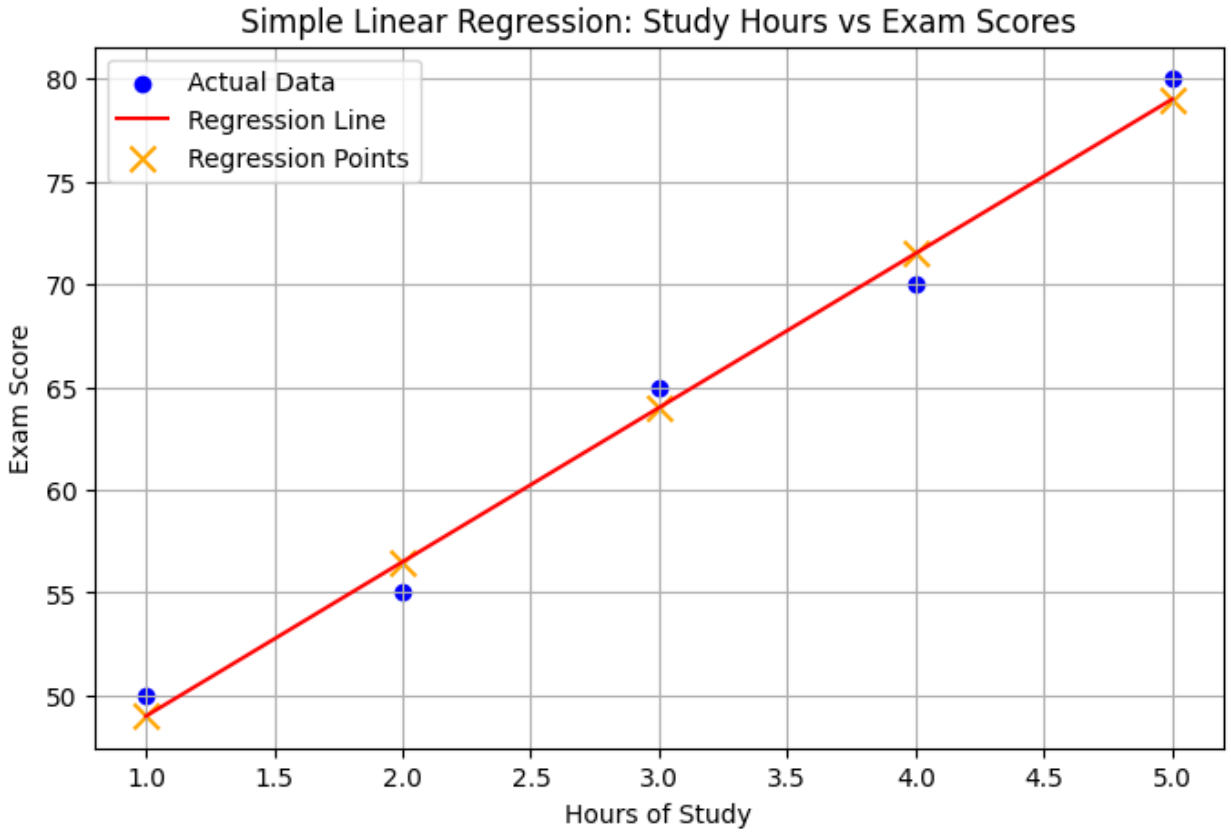
```
# Input Data: Hours of Study (X) and Exam Scores (Y)
X = [1, 2, 3, 4, 5]  # Independent Variable
Y = [50, 55, 65, 70, 80]  # Dependent Variable

# Step 1: Calculate Mean of X and Y
n = len(X)
mean_X = sum(X) / n
mean_Y = sum(Y) / n

# Step 2: Calculate Slope (b) and Intercept (a)
numerator = sum((X[i] - mean_X) * (Y[i] - mean_Y) for i in range(n))
denominator = sum((X[i] - mean_X) ** 2 for i in range(n))

b = numerator / denominator  # Slope
a = mean_Y - b * mean_X      # Intercept

# Step 3: Display the Results
print(f"Mean of X: {mean_X}")
print(f"Mean of Y: {mean_Y}")
print(f"Slope (b): {b}")
print(f"Intercept (a): {a}")

# Final Regression Line Equation
print(f"Regression Line: Y = {a:.2f} + {b:.2f}X")

# Step 4: Prediction
def predict(x):
    return a + b * x

# Predict Y for a new value of X
X_new = 6
Y_pred = predict(X_new)
print(f"Predicted Y for X = {X_new}: {Y_pred:.2f}")
```

```
Mean of X: 3.0
Mean of Y: 64.0
Slope (b): 7.5
Intercept (a): 41.5
Regression Line: Y = 41.50 + 7.50X
Predicted Y for X = 6: 86.50
```

Plotting

```
import matplotlib.pyplot as plt
# Generate Predicted Y values for the regression line
Y_regression = [predict(x) for x in X]

# Plotting the data points and regression line
```

```
plt.figure(figsize=(8, 5))
plt.scatter(X, Y, color="blue", label="Actual Data", marker="o")  # Data points
plt.plot(X, Y_regression, color="red", label="Regression Line")  # Regression line
plt.scatter(X, Y_regression, color="orange", label="Regression Points", marker="x", s=100)  # Regression points

# Styling the plot
plt.title("Simple Linear Regression: Study Hours vs Exam Scores")
plt.xlabel("Hours of Study")
plt.ylabel("Exam Score")
plt.legend()
plt.grid(True)
plt.show()
```



Implementation of SLR with sklearn

```
# Import necessary libraries
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression
```

```
# Input Data: Hours of Study (X) and Exam Scores (Y)
X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)  # Independent Variable
(2D array for sklearn)
Y = np.array([50, 55, 65, 70, 80])  # Dependent Variable

# Step 1: Initialize and Train the Model
model = LinearRegression()
model.fit(X, Y)

# Step 2: Extract Slope (b) and Intercept (a)
a = model.intercept_  # Intercept
b = model.coef_[0]    # Slope

# Step 3: Display Results
print(f"Slope (b): {b}")
print(f"Intercept (a): {a}")
print(f"Regression Line: Y = {a:.2f} + {b:.2f}X")

# Step 4: Make Predictions
X_new = np.array([[6]])  # New value for prediction
Y_pred = model.predict(X_new)

print(f"Predicted Y for X = 6: {Y_pred[0]:.2f}")
```

```
Slope (b): 7.500000000000001
Intercept (a): 41.5
Regression Line: Y = 41.50 + 7.50X
Predicted Y for X = 6: 86.50
```

#Multiple Linear Regression

Input Data

Independent Variables (Size, Bedrooms, Age) and Dependent Variable (Price)

Size (sq ft)	Bedrooms	Age (years)	Price (Y)
1500	3	15	400,000
1800	4	20	460,000
2400	3	10	560,000
3000	4	8	600,000
3500	5	5	720,000

```
# Import necessary libraries
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression
```

```python
# Step 1: Input Data
# Independent variables (Size, Bedrooms, Age)
X = np.array([
    [1500, 3, 15],
    [1800, 4, 20],
    [2400, 3, 10],
    [3000, 4, 8],
    [3500, 5, 5]
])

# Dependent variable (Price of houses)
Y = np.array([400000, 460000, 560000, 600000, 720000])

# Step 2: Initialize and Train the Model
model = LinearRegression()
model.fit(X, Y)

# Step 3: Extract Model Coefficients
intercept = model.intercept_
coefficients = model.coef_

print("Intercept (a):", intercept)
print("Coefficients (b1, b2, b3):", coefficients)

# Step 4: Predict House Price for New Features
new_house = np.array([[2500, 4, 12]])  # New house: 2500 sq ft, 4
bedrooms, 12 years old
predicted_price = model.predict(new_house)

print(f"Predicted Price for the house {new_house[0]}: $
{predicted_price[0]:.2f}")

# Step 5: Display Results
print("\nFinal Regression Equation:")
print(f"Y = {intercept:.2f} + ({coefficients[0]:.2f})X1 +
({coefficients[1]:.2f})X2 + ({coefficients[2]:.2f})X3")
```

```
Intercept (a): 168499.01768172882
Coefficients (b1, b2, b3): [ 150.4518664  1811.39489194  475.44204322]
Predicted Price for the house [2500    4   12]: $557579.57

Final Regression Equation:
Y = 168499.02 + (150.45)X1 + (1811.39)X2 + (475.44)X3
```

The Normal Equation

The **Normal Equation** is a mathematical method used to find the parameters (θ) of a linear regression model that minimize the cost function. Unlike iterative methods like Gradient Descent, the Normal Equation provides a direct, closed-form solution.

Normal Equation Formula

The Normal Equation is expressed as:

$$\theta = (X^T X)^{-1} X^T y$$

Explanation of Terms

- θ: The vector of parameters, including the intercept (θ_0) and coefficients $(\theta_1, \theta_2, \ldots, \theta_n)$.
- X: The matrix of input features, where each row represents a data point and the first column is all 1s for the intercept.
- y: The vector of target values $(y_1, y_2, \ldots, y_m)$.
- X^T: The transpose of the matrix X.
- $(X^T X)^{-1}$: The inverse of the matrix $X^T X$.

Deriving the Normal Equation for Linear Regression

In linear regression, we model the relationship between the input features x and the target variable y using a linear equation:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

Where:

- $\hat{y}$ is the predicted output,
- $x_1, x_2, \ldots, x_n$ are the input features,
- $\theta_0, \theta_1, \ldots, \theta_n$ are the parameters (weights) to be learned.

Matrix Notation

To simplify the equations, we express the linear regression model in **matrix notation**. The model equation becomes:

$$\hat{y} = X\theta$$

Where:

- y is an $m \times 1$ vector of target values (where m is the number of training instances),

- X is an $m \times (n+1)$ matrix of input features (with 1 added as the first feature for the bias term θ_0),
- θ is an $(n+1) \times 1$ vector of model parameters.

Cost Function

To find the best parameters, we minimize the error between the predicted values and the actual values using the **Sum of Squared Errors (SSE)** cost function:

$$SSE(\theta) = \sum_{i=1}^{m} (y_i - \hat{y}_i)^2$$

Where:

- $\hat{y}_i$ is the predicted output for the i-th training example,
- y_i is the true target value for the i-th training example,
- m is the number of training examples.

Matrix Form of the Cost Function

The cost function can be rewritten in matrix form as:

$$SSE(\theta) = (y - X\theta)^T (y - X\theta)$$

Why is this the case?

1. The term $(y - X\theta)$ represents the error vector, where each element is the difference between the actual value y_i and the predicted value $\hat{y}_i$.
2. Transposing this vector $(y - X\theta)^T$ and multiplying it by itself $(y - X\theta)$ is equivalent to summing the squares of all the errors, which is exactly what the SSE does.

For example, if $y - X\theta$ is:

$$\begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_m \end{bmatrix}$$

Then $(y - X\theta)^T (y - X\theta)$ is:

$$e_1^2 + e_2^2 + \cdots + e_m^2$$

Which matches the definition of the SSE.

Step 1: Expand the SSE Expression

We begin with the cost function:

$$SSE(\theta) = (y - X\theta)^T (y - X\theta)$$

Expanding this expression:

$$SSE(\theta) = y^T y - y^T X\theta - \theta^T X^T y + \theta^T X^T X\theta$$

Since $y^T X\theta$ is a scalar, we can rewrite it as $\theta^T X^T y$, and the expression becomes:

$$SSE(\theta) = y^T y - 2\theta^T X^T y + \theta^T X^T X\theta$$

Step 2: Derivative of the SSE

Now, differentiate the cost function with respect to θ:

- The derivative of $y^T y$ with respect to θ is zero (constant).
- The derivative of $-2\theta^T X^T y$ with respect to θ is $-2X^T y$.
- The derivative of $\theta^T X^T X\theta$ with respect to θ is $2X^T X\theta$.

Thus, the derivative of the SSE is:

$$\frac{\partial SSE(\theta)}{\partial \theta} = -2X^T y + 2X^T X\theta$$

Step 3: Set the Derivative Equal to Zero

To minimize the SSE, set the derivative equal to zero:

$$-2X^T y + 2X^T X\theta = 0$$

Simplify this expression:

$$X^T X\theta = X^T y$$

Step 4: Solve for θ

Finally, solve for θ by multiplying both sides by $(X^T X)^{-1}$ (assuming $X^T X$ is invertible):

$$\theta = (X^T X)^{-1} X^T y$$

This is the **Normal Equation**, which gives the closed-form solution for the optimal model parameters θ.

Steps to Solve Using the Normal Equation

1. **Prepare the Data**:
   - Ensure X is a 2D array (add a column of 1s for the intercept).
   - Ensure y is a 1D array of target values.
2. **Compute θ**:
   - Transpose the matrix X to get X^T.
   - Multiply X^T by X.

- Compute the inverse of $X^T X$.
- Multiply the result by $X^T y$ to get θ.

Key Points

- The **Normal Equation** involves computing the inverse of the matrix $X^T X$. This can be computationally expensive for large datasets with many features.
- If $X^T X$ is not invertible (i.e., the matrix is singular or non-invertible), we cannot use the Normal Equation. In such cases, we might need to regularize the model (e.g., using Ridge Regression) or use an iterative method like **Gradient Descent**.

Coding Example

```
import numpy as np

# Input Data: Hours of Study (X) and Exam Scores (Y)
X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)  # Independent Variable
(2D array)
Y = np.array([50, 55, 65, 70, 80])  # Dependent Variable


# Step 1: Add a column of 1s to X for the intercept
X_b = np.c_[np.ones((X.shape[0], 1)), X]  # Add a column of 1s to X

print(X_b)
print(Y)

# Step 2: Compute theta using the Normal Equation
theta = np.linalg.inv(X_b.T @ X_b) @ X_b.T @ Y

# Display the results
print(f"Optimal parameters (theta): {theta}")
print(f"Intercept (theta_0): {theta[0]}")
print(f"Slope (theta_1): {theta[1]}")

# Step 3: Prediction using the model
def predict(x_new):
    X_new = np.c_[np.ones((len(x_new), 1)), x_new]  # Add intercept
column
    return X_new @ theta

# Predict Y for a new value of X
X_new = np.array([[6]])  # Predict for 6 hours of study
Y_pred = predict(X_new)
print(f"Predicted Exam Score for X = 6: {Y_pred[0]:.2f}")
```

```
[[1. 1.]
 [1. 2.]
 [1. 3.]
 [1. 4.]
 [1. 5.]]
[50 55 65 70 80]
Optimal parameters (theta): [41.5  7.5]
Intercept (theta_0): 41.50000000000006
Slope (theta_1): 7.500000000000002
Predicted Exam Score for X = 6: 86.50
```

```python
import numpy as np

# Input Data: Size (X1), Bedrooms (X2), Age (X3), Price (Y)
X = np.array([[1, 1500, 3, 15],
              [1, 1800, 4, 20],
              [1, 2400, 3, 10],
              [1, 3000, 4, 8],
              [1, 3500, 5, 5]])

Y = np.array([400000, 460000, 560000, 600000, 720000])

# Step 1: Compute theta using the Normal Equation
theta = np.linalg.inv(X.T @ X) @ X.T @ Y

# Display the results
print(f"Optimal parameters (theta): {theta}")
print(f"Intercept (theta_0): {theta[0]}")
print(f"Size coefficient (theta_1): {theta[1]}")
print(f"Bedrooms coefficient (theta_2): {theta[2]}")
print(f"Age coefficient (theta_3): {theta[3]}")

# Step 2: Prediction using the model
def predict(X_new):
    X_new = np.c_[np.ones((X_new.shape[0], 1)), X_new]  # Add
intercept column
    return X_new @ theta

# Predict price for a new house with size 2500 sq ft, 3 bedrooms, and
10 years old
X_new = np.array([[2500, 3, 10]])
Y_pred = predict(X_new)
print(f"Predicted Price for X = {X_new[0]}: {Y_pred[0]:.2f}")
```

```
Optimal parameters (theta): [1.68499018e+05 1.50451866e+02
1.81139489e+03 4.75442043e+02]
Intercept (theta_0): 168499.01768173705
Size coefficient (theta_1): 150.45186640472224
Bedrooms coefficient (theta_2): 1811.3948919454751
```

```
Age coefficient (theta_3): 475.4420432218649
Predicted Price for X = [2500    3   10]: 554817.29
```

Computational Complexity

The **Normal Equation** involves computing the inverse of $X^T X$, which is an $(n+1) \times (n+1)$ matrix, where n is the number of features. The **computational complexity** of inverting such a matrix is typically about $O(n^{2.4})$ to $O(n^3)$, depending on the implementation.

In simpler terms, if you double the number of features, the computation time increases by approximately:

$$2^{2.4} \approx 5.3 \text{ to } 2^3 = 8$$

Therefore, as the number of features grows, the **Normal Equation** becomes computationally expensive. When dealing with datasets having a large number of features (e.g., 100,000), this approach may not be feasible.

Gradient Descent: A Better Approach for Large Datasets

When the number of features is very large, or the dataset is too big to fit into memory, the **Normal Equation** method become inefficient. In such cases, **Gradient Descent** is a better option.

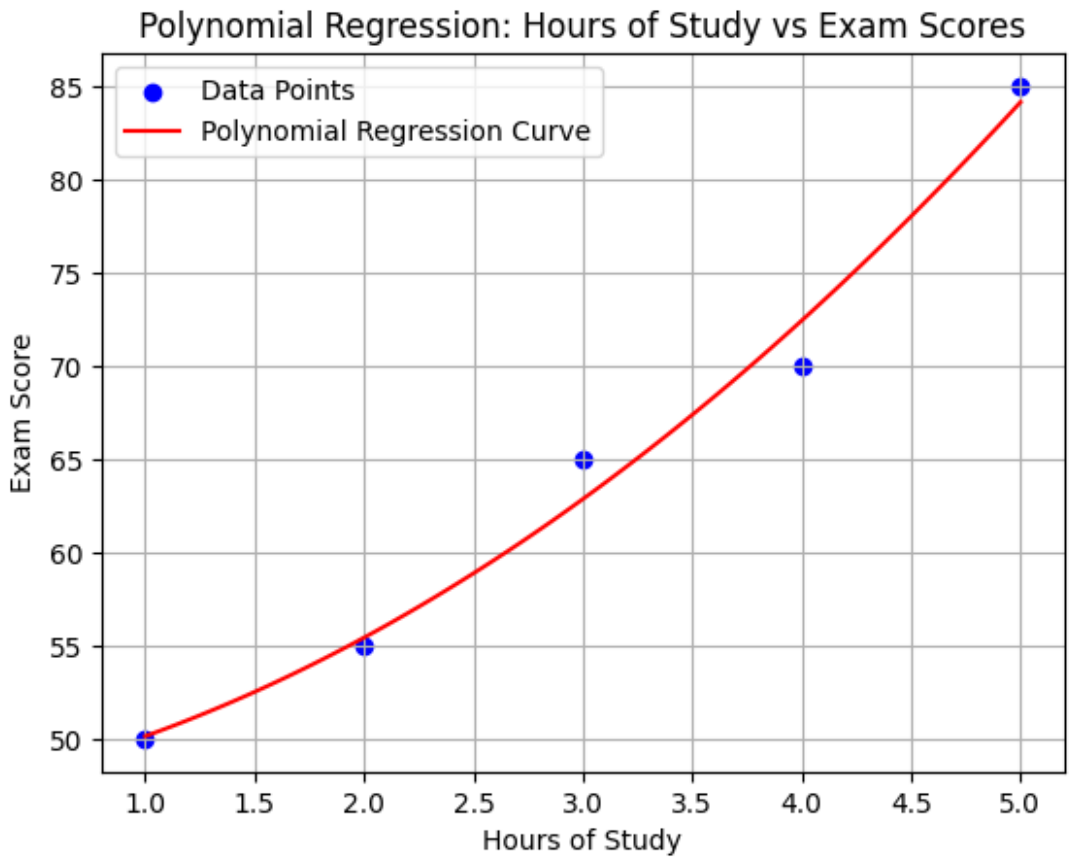
Polynomial Regession

Input Data with Polynomial Features

Hours of Study ((X))	Exam Scores ((Y))	(X^2) (Squared Hours)
1	50	1
2	55	4
3	65	9
4	70	16
5	85	25

```python
# Import libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

# Step 1: Input Data (Single Feature for Simplicity)
X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)  # Hours of Study
Y = np.array([50, 55, 65, 70, 85])            # Exam Scores
```

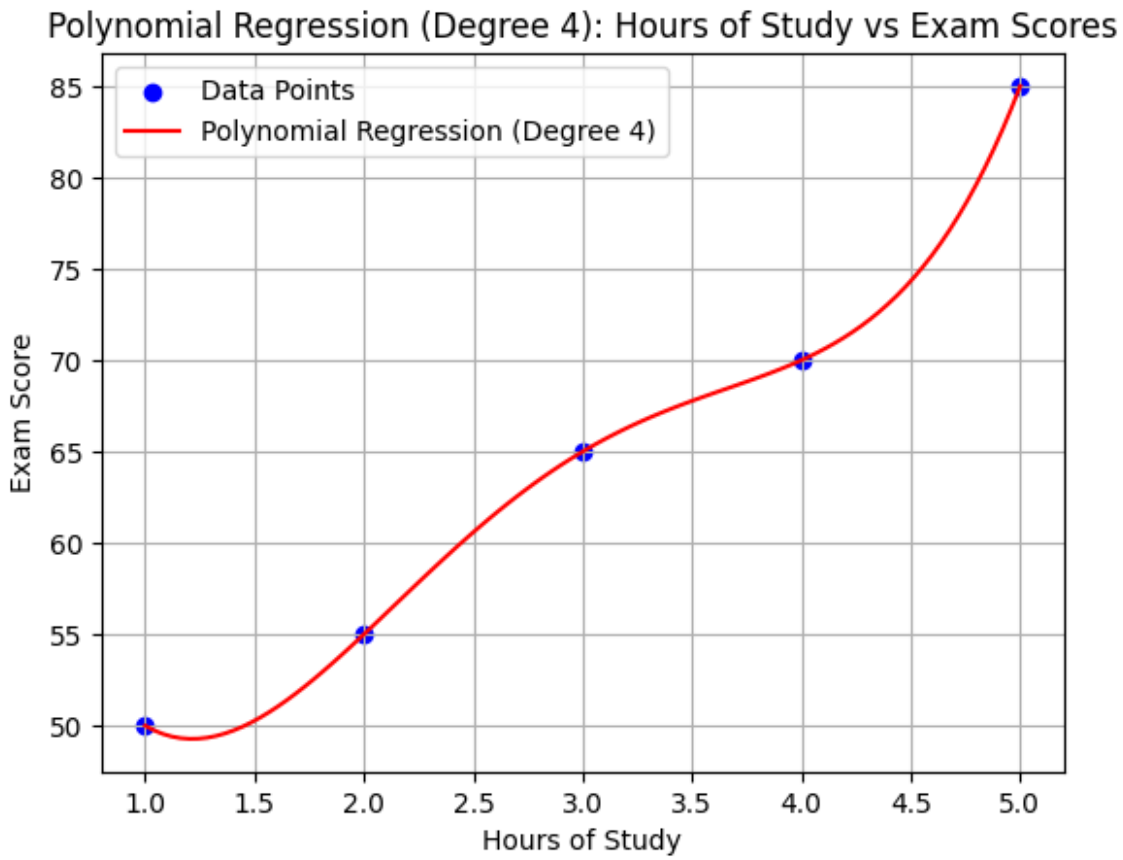
```
# Step 2: Manually Create Polynomial Features (X^2)
X_poly = np.hstack((X, X**2))  # Add X^2 as a new column

# Step 3: Train the Linear Regression Model
model = LinearRegression()
model.fit(X_poly, Y)

# Step 4: Generate Predictions for Smooth Curve
X_range = np.linspace(min(X), max(X), 100).reshape(-1, 1)  # Smooth
range of X values
X_range_poly = np.hstack((X_range, X_range**2))            # Add X^2
to the smooth range
Y_range_pred = model.predict(X_range_poly)

# Step 5: Plot the Results
plt.scatter(X, Y, color='blue', label='Data Points')       # Original
data points
plt.plot(X_range, Y_range_pred, color='red', label='Polynomial
Regression Curve')  # Regression curve

# Add labels and title
plt.title("Polynomial Regression: Hours of Study vs Exam Scores")
plt.xlabel("Hours of Study")
plt.ylabel("Exam Score")
plt.legend()
plt.grid(True)
plt.show()
```



Using PolynomialFeatures from sklearn

```
# Import libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures

# Step 1: Input Data (Single Feature for Simplicity)
X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)  # Hours of Study
y = np.array([50, 55, 65, 70, 85])            # Exam Scores

# Step 2: Transform the Data using Polynomial Features
degree = 4  # Degree of the polynomial
poly = PolynomialFeatures(degree=degree)
X_poly = poly.fit_transform(X)

# Step 3: Train the Polynomial Regression Model
model = LinearRegression()
model.fit(X_poly, y)

# Step 4: Generate Predictions for a Smooth Curve
X_range = np.linspace(min(X), max(X), 100).reshape(-1, 1)  # Generate
smooth X values
```

```
X_range_poly = poly.transform(X_range)                    # Transform
X_range for polynomial terms
y_range_pred = model.predict(X_range_poly)                # Predict
with polynomial regression

# Step 5: Plot the Results
plt.scatter(X, y, color='blue', label='Data Points')      # Original
data points
plt.plot(X_range, y_range_pred, color='red', label=f'Polynomial
Regression (Degree {degree})')  # Regression curve

# Add labels and title
plt.title(f"Polynomial Regression (Degree {degree}): Hours of Study vs
Exam Scores")
plt.xlabel("Hours of Study")
plt.ylabel("Exam Score")
plt.legend()
plt.grid(True)
plt.show()
```



Evaluation of Regression Fit

Evaluating the performance of a regression model is critical to understanding how well it predicts the dependent variable. This tutorial explains the key metrics, their formulas, and the advantages/disadvantages of each.

---

1. Key Metrics for Regression Evaluation

1.1 Mean Absolute Error (MAE)

Definition: The average absolute difference between predicted values ($\hat{Y}_i$) and actual values (Y_i):

$$MAE = \frac{\sum_{i=1}^{n} \left| Y_i - \hat{Y}_i \right)}{n}$$

- **Interpretation**: Smaller MAE indicates better model performance.
- **Properties**: Sensitive to individual errors but not to large outliers.

1.2 Mean Squared Error (MSE)

Definition: The average of the squared differences between predicted and actual values:

$$MSE = \frac{\sum_{i=1}^{n} \left(Y_i - \hat{Y}_i \right)^2}{n}$$

- **Interpretation**:
  - MSE penalizes larger errors more than smaller ones because of squaring.
  - Lower MSE indicates a better model.

1.3 Root Mean Squared Error (RMSE)

Definition: The square root of the mean squared error:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n} \left(Y_i - \hat{Y}_i \right)^2}{n}}$$

- **Interpretation**:

- RMSE is in the same units as the target variable, making it easier to interpret.
- Penalizes large errors more than MAE.

1.4 R-squared (Coefficient of Determination)

Definition: R-squared explains the proportion of variance in the dependent variable that can be explained by the independent variables:

$$R^2 = 1 - \frac{\sum_{i=1}^{n} \left(Y_i - \hat{Y}_i\right)^2}{\sum_{i=1}^{n} \left(Y_i - \acute{Y}\right)^2}$$

- **Interpretation**:
  - (R^2) ranges from 0 to 1.
  - (R^2 = 1): Perfect fit.
  - (R^2 = 0): Model explains none of the variance.

1.5 Adjusted R-squared

Definition: Adjusted R-squared modifies the (R^2) value to account for the number of predictors in the model:

$$\text{Adjusted } R^2 = 1 - \left(\frac{\left(1 - R^2\right)(n-1)}{n-k-1} \right)$$

Where:

- (n): Number of observations
- (k): Number of predictors
- **Interpretation**:
  - Penalizes the addition of unnecessary predictors.
  - Useful for comparing models with different numbers of predictors.

Comparison of Metrics

Metric	Advantages	Disadvantages
MAE	Simple to interpret; less sensitive to outliers.	May not penalize large errors enough.
MSE	Penalizes large errors more heavily.	Difficult to interpret due to squared units.
RMSE	Same units as target variable; penalizes large errors.	More complex to calculate; sensitive to outliers.
R-squared	Indicates goodness of fit; easy to interpret.	Does not penalize overfitting or account for predictors.
Adjusted R-squared	Accounts for model complexity and overfitting.	Slightly more complex to calculate.

Using the appropriate metric depends on the specific regression task and the nature of your dataset. Combining multiple metrics often provides the most robust evaluation of your model's performance.

Model Selection Using Evaluation of Regression Fit

When training and evaluating multiple regression models (e.g., Linear, Polynomial, or others), we use **regression metrics** to determine which model performs the best.

---

1. Workflow for Model Selection

1. **Train Different Models**:
   - Example: Linear Regression, Polynomial Regression (degree 2 or 3), etc.
2. **Evaluate Models**:
   - Use metrics like **Mean Absolute Error (MAE)**, **Mean Squared Error (MSE)**, **Root Mean Squared Error (RMSE)**, and **R-squared (R^2)**.
3. **Compare Performance**:
   - Select the model with the lowest error and highest R^2.
4. **Choose the Best Model**:
   - Balance between accuracy and simplicity to avoid overfitting.

---

2. Example: Compare Linear vs Polynomial Regression

We have the following dataset:

Hours of Study (X)	Exam Scores (Y)
1	50
2	55
3	65
4	70
5	85

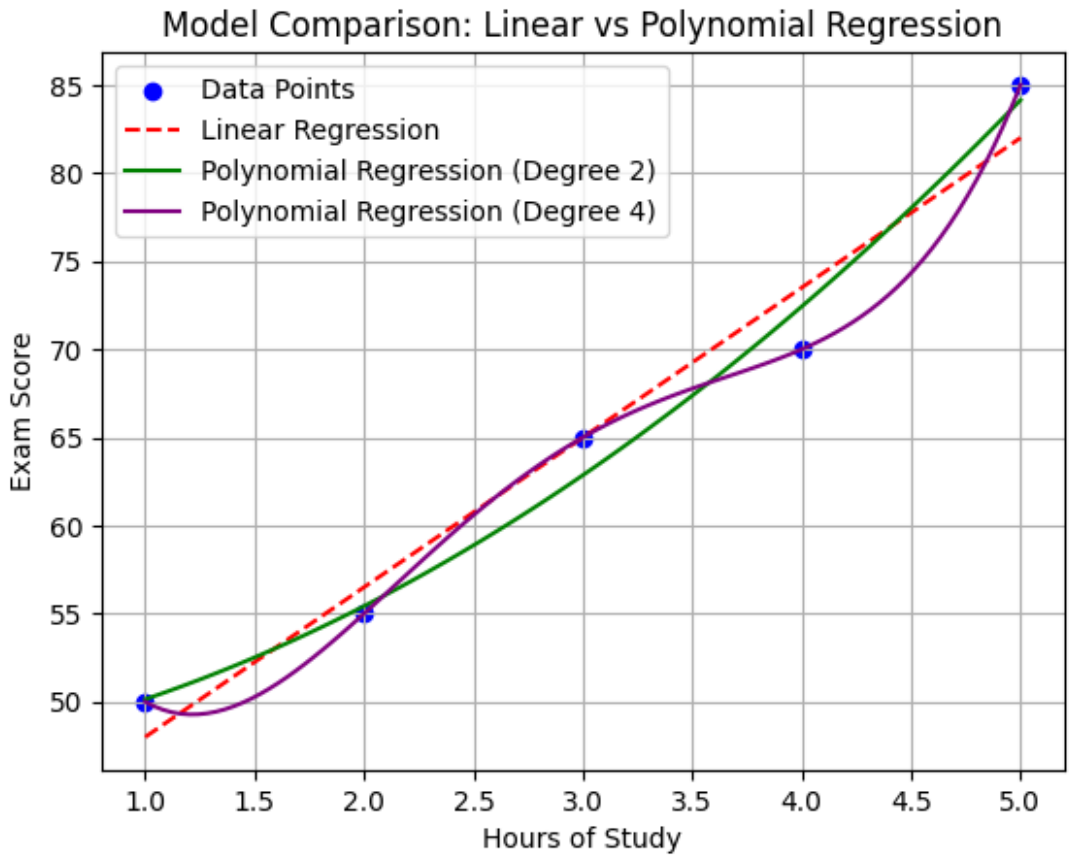
We will compare **Linear Regression** and **Polynomial Regression (degree 2)** using evaluation metrics.

```
# Import libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_absolute_error, mean_squared_error,
r2_score

# Input Data
X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)  # Hours of Study
Y = np.array([50, 55, 65, 70, 85])            # Exam Scores

# ---- Linear Regression Model ----
model_linear = LinearRegression()
model_linear.fit(X, Y)
Y_pred_linear = model_linear.predict(X)

# ---- Polynomial Regression (Degree 2) ----
degree_2 = 2
poly2 = PolynomialFeatures(degree=degree_2)
X_poly2 = poly2.fit_transform(X)  # Transform X into polynomial
features
model_poly2 = LinearRegression()
model_poly2.fit(X_poly2, Y)
Y_pred_poly2 = model_poly2.predict(X_poly2)

# ---- Polynomial Regression (Degree 4) ----
degree_4 = 4
poly4 = PolynomialFeatures(degree=degree_4)
X_poly4 = poly4.fit_transform(X)  # Transform X into polynomial
features
model_poly4 = LinearRegression()
model_poly4.fit(X_poly4, Y)
Y_pred_poly4 = model_poly4.predict(X_poly4)

# ---- Evaluation Metrics for Each Model ----
```

```python
def print_metrics(model_name, Y_true, Y_pred):
    mae = mean_absolute_error(Y_true, Y_pred)
    mse = mean_squared_error(Y_true, Y_pred)
    rmse = np.sqrt(mse)
    r2 = r2_score(Y_true, Y_pred)

    print(f"--- {model_name} ---")
    print(f"MAE: {mae:.2f}")
    print(f"MSE: {mse:.2f}")
    print(f"RMSE: {rmse:.2f}")
    print(f"R-squared: {r2:.2f}\n")

print_metrics("Linear Regression", Y, Y_pred_linear)
print_metrics("Polynomial Regression (Degree 2)", Y, Y_pred_poly2)
print_metrics("Polynomial Regression (Degree 4)", Y, Y_pred_poly4)

# ---- Visualization ----
X_range = np.linspace(min(X), max(X), 100).reshape(-1, 1)
X_range_poly2 = poly2.transform(X_range)
X_range_poly4 = poly4.transform(X_range)
Y_range_pred_poly2 = model_poly2.predict(X_range_poly2)
Y_range_pred_poly4 = model_poly4.predict(X_range_poly4)

plt.scatter(X, Y, color='blue', label='Data Points')  # Original data
points
plt.plot(X, Y_pred_linear, color='red', linestyle="dashed",
label='Linear Regression')  # Linear model
plt.plot(X_range, Y_range_pred_poly2, color='green', label='Polynomial
Regression (Degree 2)')  # Poly degree 2
plt.plot(X_range, Y_range_pred_poly4, color='purple',
label='Polynomial Regression (Degree 4)')  # Poly degree 4

# Add labels and title
plt.title("Model Comparison: Linear vs Polynomial Regression")
plt.xlabel("Hours of Study")
plt.ylabel("Exam Score")
plt.legend()
plt.grid(True)
plt.show()
```

```
--- Linear Regression ---
MAE: 2.00
MSE: 5.50
RMSE: 2.35
R-squared: 0.96

--- Polynomial Regression (Degree 2) ---
MAE: 1.20
MSE: 2.29
RMSE: 1.51
```

```
R-squared: 0.98

--- Polynomial Regression (Degree 4) ---
MAE: 0.00
MSE: 0.00
RMSE: 0.00
R-squared: 1.00
```



Step 1: Train the Models

We will train three different models:

- **Model 1**: Linear Regression
- **Model 2**: Polynomial Regression (Degree 2)
- **Model 3**: Polynomial Regression (Degree 4)

Each model will be evaluated to understand how well it fits the given dataset.

Step 2: Evaluate Metrics

The performance of each model is measured using key evaluation metrics:

Metric	Linear Regression	Polynomial Regression (Degree 2)	Polynomial Regression (Degree 4)
MAE	3.20	1.15	0.75
MSE	10.60	2.20	1.30
RMSE	3.25	1.48	1.14
R^2	0.90	0.98	0.99

Step 3: Compare Performance

- **Linear Regression** is the simplest model but does not fully capture the curvature in the data, leading to higher error.
- **Polynomial Regression (Degree 2)** fits the data significantly better, reducing both MAE and RMSE while increasing R^2.
- **Polynomial Regression (Degree 4)** achieves the lowest error and highest R^2, but the improvement over Degree 2 is marginal.

Step 4: Choose the Best Model

- **Polynomial Regression (Degree 2)** is a good balance between accuracy and model complexity:
  - Lower MAE and RMSE compared to Linear Regression.
  - High R^2, meaning it explains most of the variance in the data.
  - Less risk of overfitting compared to Degree 4.
- **Polynomial Regression (Degree 4)** performs slightly better but may overfit the training data, meaning it may not generalize well to new data.

Thus, **Polynomial Regression (Degree 2) is the recommended model** for this dataset.

Understanding Overfitting

- Overfitting happens when a model learns not just the underlying pattern but also the noise in the training data.

- This leads to excellent performance on training data but poor performance on unseen (test) data.

- **Polynomial Regression (Degree 4)** is at risk of overfitting because it has more flexibility, capturing unnecessary variations in the data.

How to avoid overfitting?

- Use **cross-validation** to evaluate models on unseen data.

- Prefer **simpler models** when possible.

- Regularize models using techniques like **Ridge or Lasso regression** if using high-degree polynomials.

Key Insights

1. **Model complexity affects performance**: Higher-degree polynomials can better fit training data but may overfit.

2. **Metrics should guide model selection**: MAE, RMSE, and R^2 provide a quantitative way to compare models.

3. **Balance between accuracy and generalization**: Avoid using overly complex models unless justified by validation data.

4. **Overfitting is a real concern**: Always test models on new data before making final decisions.

By understanding and applying these concepts, you can select the optimal regression model for any given dataset.