Polynomial Regression - Complete Breakdown

1. In-Depth and Specific Intuitive Understanding

What is Polynomial Regression?

Polynomial Regression is an **extension of Linear Regression** that allows modeling of **non-linear relationships** by introducing polynomial terms of the input features.

Key Idea

- In **Linear Regression**, we assume a straight-line relationship:

  $$y = w_0 + w_1 x_1 + w_2 x_2 + ... + w_n x_n$$

- In **Polynomial Regression**, we introduce polynomial terms:

  $$y = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + ... + w_d x^d$$

  where d is the **degree** of the polynomial.

- We **still solve it using Linear Regression techniques**, but we first **transform** the features into polynomial form.

Example

If we have a single feature x, we transform:

$$X_{\text{transformed}} = [x, x^2, x^3, ..., x^d]$$

This allows **Linear Regression to fit a non-linear curve.**

2. When Polynomial Regression is Used and When It Should Be Avoided

✅ When to Use It

- When the relationship between X and y is **non-linear but smooth**.
- When **Linear Regression underfits the data**.
- When a **low-degree polynomial captures the trend**.

❌ When to Avoid It

- If **the dataset is noisy**, high-degree polynomials may overfit.
- If **the number of features is already large**, polynomial transformation increases feature space exponentially.
- If **the degree of the polynomial is too high**, it can lead to **poor generalization**.

3. When It Fails to Converge and How to Avoid That

When Polynomial Regression Fails

- Overfitting with high-degree polynomials.
- Multicollinearity in polynomial features.
- Numerical instability in Normal Equation if $X^T X$ is non-invertible.

How to Ensure Convergence

✅ **Feature Scaling** (Standardization).
✅ **Regularization (Ridge/Lasso Regression)**.
✅ **Cross-validation** to select the best polynomial degree.

When Polynomial Regression Always Converges

- If we use **Gradient Descent with a proper learning rate**.
- If we use **Normal Equation (but only when the number of features is manageable)**.
- If we **apply regularization**.

4. Advantages and Disadvantages

Advantages

✅ **Can model non-linear relationships** using simple linear regression techniques.

✅ **Easy to implement.**

✅ **Computationally efficient for low-degree polynomials.**

Disadvantages

❌ **Overfitting risk** with high-degree polynomials.

❌ **Exponential feature growth.**

❌ **Unstable for large-degree models.**

5. Intuitive Algorithm / Pseudo Code

1. **Transform input features**: Convert x into $x, x^2, x^3, ..., x^d$.
2. **Train using Linear Regression**:
   - Compute Normal Equation or use Gradient Descent.
3. **Make predictions using the polynomial model**.

plaintext Copy Edit

```
1. Choose a polynomial degree d
2. Convert each feature into its polynomial terms (x → x, x^2, ..., x^d)
3. Train a Linear Regression model on the transformed dataset
4. Use the trained model to make predictions
```

6. Mathematical and Logical Breakdown

Feature Transformation

Given an input feature matrix:

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

We transform it into:

$$X_{\text{poly}} = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ 1 & x_3 & x_3^2 & x_3^3 \end{bmatrix}$$

where:

- The first column is **bias term (intercept)**.
- The remaining columns are **higher-order polynomial terms**.

Training (Using Normal Equation)

The optimal weights are:

$$w = (X_{\text{poly}}^T X_{\text{poly}})^{-1} X_{\text{poly}}^T y$$

Training (Using Gradient Descent)

Compute gradient:

$$\frac{\partial J}{\partial w} = \frac{1}{m} X_{\text{poly}}^T (X_{\text{poly}} w - y)$$

Update weights:

$$w := w - \alpha \frac{\partial J}{\partial w}$$

```python
import numpy as np
class PolynomialRegression:
    def __init__(self, degree=2, learning_rate=0.01, epochs=1000, method="gd"):
        """
        Initialize the Polynomial Regression model.
        Parameters:
        - degree: Polynomial degree || - learning_rate: Step size for gradient descent || - epochs: Number of iterations for gradient descent
        - method: "gd" for gradient descent, "ne" for normal equation
        """
        self.degree = degree
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.method = method
        self.weights = None

    def _polynomial_features(self, X):
        """Generate polynomial features up to the given degree."""
        poly_X = np.ones((X.shape[0], self.degree + 1))  # Initialize matrix with ones (for bias term)
        for d in range(1, self.degree + 1):
            poly_X[:, d] = X[:, 0] ** d  # Compute x^d for each degree
        return poly_X

    def fit(self, X, y):
        """Train the model using either gradient descent or normal equation."""
        X_poly = self._polynomial_features(X)
        m, n = X_poly.shape

        if self.method == "gd":  # Gradient Descent
            self.weights = np.zeros(n)

            for _ in range(self.epochs):
                predictions = np.dot(X_poly, self.weights)
                error = predictions - y
                gradient = (1/m) * np.dot(X_poly.T, error)
                self.weights -= self.learning_rate * gradient

        elif self.method == "ne":  # Normal Equation
            self.weights = np.linalg.pinv(X_poly.T @ X_poly) @ (X_poly.T @ y)  # Compute w = (X^T X)^-1 X^T y

    def predict(self, X):
        """Make predictions using the trained model."""
        X_poly = self._polynomial_features(X)
        return np.dot(X_poly, self.weights)
```

8. Scikit-Learn Implementation (Fully Commented)

python

Copy Edit

```python
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_regression

# Generate synthetic dataset
X, y = make_regression(n_samples=100, n_features=1, noise=10, random_state=42)

# Split dataset into training (80%) and testing (20%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

# Create a pipeline that first transforms features into polynomial terms, then appl
degree = 3  # Choose polynomial degree
model = make_pipeline(PolynomialFeatures(degree), LinearRegression())

# Train the model on the transformed data
model.fit(X_train, y_train)

# Predict on the test set
y_pred = model.predict(X_test)

# Print model coefficients
print("Coefficients:", model.named_steps['linearregression'].coef_)
```

Final Summary

- Polynomial Regression is Linear Regression with polynomial feature transformations.
- It allows fitting non-linear relationships using simple linear regression techniques.
- Feature expansion can lead to overfitting, so careful tuning of the polynomial degree is essential.
- Regularization (Ridge/Lasso) helps stabilize high-degree models.
- Use the Normal Equation when the dataset is small, and Gradient Descent when it is large.