How can I perform polynomial regression using scikit-learn?

Authored by

stats writer

November 21, 2025

Performing polynomial regression within the scikit-learn framework is a streamlined process that bridges the gap between simple linear regression and complex non-linear relationships. The core mechanism involves leveraging the **PolynomialFeatures** class, which is housed within `sklearn.preprocessing`. This class generates new feature matrices consisting of all polynomial combinations of the features up to the specified degree.

Once the transformation object is initialized, it is used to elevate the input data dimensions, effectively transforming the original features into a higher-degree polynomial space. Critically, after this transformation, we can still utilize a standard **Linear Regression** model, as the relationship becomes linear in terms of the new, derived features. The model is then trained using the standard `fit()` method on this transformed data. Subsequently, the `predict()` method enables the generation of predictions based on the trained model, providing crucial insights into complex data patterns.

Understanding Polynomial Regression

Polynomial regression is a powerful form of analysis utilized when the underlying relationship between the **predictor variable** (or independent variable) and the **response variable** (or dependent variable) exhibits a curvilinear pattern, making standard linear regression unsuitable. While it appears to model a nonlinear relationship, it is fundamentally still a type of linear model because it is linear in the coefficients ($beta$ values).

This technique transforms the feature space by adding polynomial terms of the independent variables. By incorporating higher-order terms (like X^2, X^3, etc.), the model gains the flexibility required to capture dips, peaks, and curves in the data that a simple straight line cannot accommodate. This approach is essential in fields ranging from economics to physics where relationships are rarely perfectly straight.

Mathematically, polynomial regression models a relationship that takes the generalized form shown below, where h dictates the overall degree of the polynomial, which is the highest power of the predictor variable X:

Y = β0 + β1X + β2X2 + ... + βhXh + ε

The core challenge in implementing this in Python is efficiently generating these powered terms ($X^2, X^3, dots$) from the original feature X. Fortunately, the **scikit-learn** library provides specialized tools to handle this feature engineering automatically. The subsequent steps provide a detailed walkthrough of how to utilize the Python ecosystem, specifically **scikit-learn**, to effectively implement and visualize a polynomial regression model.

Step 1: Setting up the Environment and Generating Synthetic Data

Before fitting any model, the first crucial step is to prepare the dataset and ensure the necessary libraries are imported. For this demonstration, we rely on NumPy for efficient array manipulation and data creation, and **Matplotlib** for visualizing the data relationship. We will create a synthetic dataset where the relationship between the predictor variable (x) and the response variable (y) is visibly non-linear.

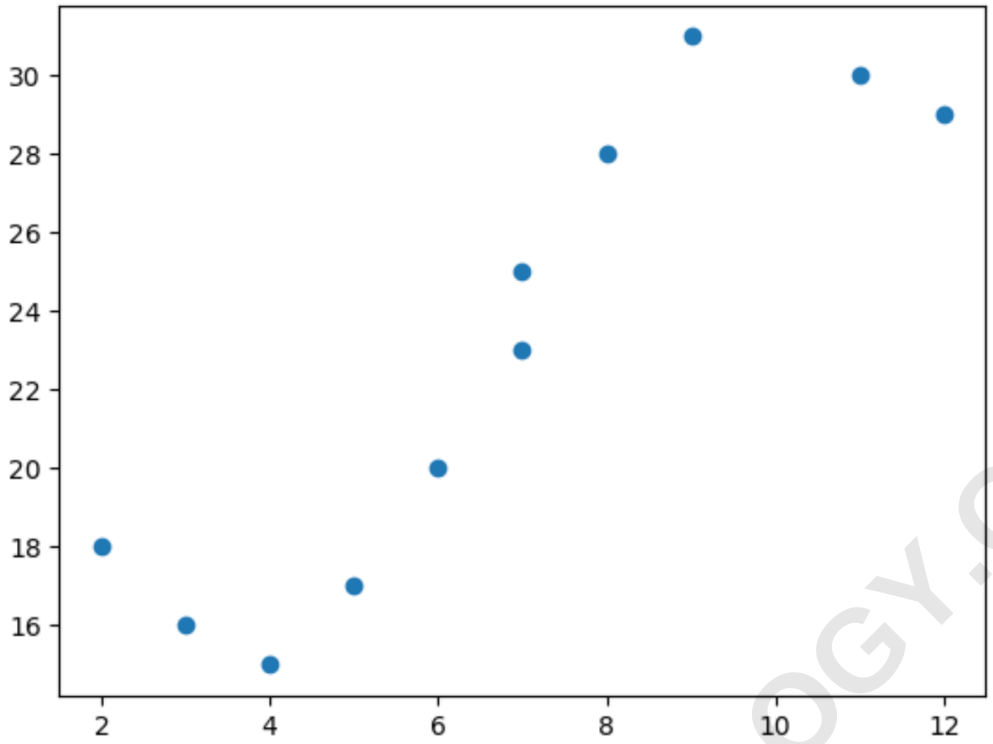
We initialize two **NumPy** arrays, `x` and `y`, representing our independent and dependent variables, respectively. These arrays will serve as the foundation for our regression analysis, simulating real-world observations where a simple linear fit would result in high error.

```
import matplotlib.pyplot as plt
import numpy as np

# Define predictor and response variables
x = np.array()
y = np.array()

# Create scatterplot to visualize initial relationship
plt.scatter(x, y)
```

Executing the visualization code provides immediate feedback on the data structure. The resulting scatterplot is essential for diagnosing the appropriate modeling technique. If the data points cluster around a curve, as they do here, it strongly suggests that methods capable of handling nonlinearity--such as polynomial regression--will outperform standard linear regression.



Analyzing the Data and Identifying Nonlinearity

Observation of the scatterplot confirms that the relationship between the predictor variable (x) and the response variable (y) is distinctly non-linear. The data initially decreases slightly, then begins a clear upward trajectory that seems to accelerate, resembling the shape of a quadratic or cubic curve. Attempting to fit a simple straight line (a degree 1 polynomial) to this data would introduce significant systematic errors, known as **bias**, because the model inherently lacks the complexity to follow the data trend.

This visual confirmation justifies the decision to employ polynomial regression. By introducing higher-order terms, we are engineering features that allow the model to bend and conform to the observed curvature. The selection of the polynomial degree is a critical aspect of this process; a degree that is too low will lead to underfitting (high bias), while a degree that is too high might lead to overfitting (high variance), where the model fits the noise rather than the underlying pattern.

For the current dataset, based on the smooth S-like curve observed, a relatively low degree, such as degree 3, is often a suitable starting point. A cubic model (X^3) provides enough flexibility to capture a single inflection point, which aligns well with the visual pattern displayed in the scatterplot. The next step in our process focuses on transforming our single feature X into three features: X^1, X^2, and X^3, using specialized tools provided by scikit-learn.

Step 2: Transforming Features using PolynomialFeatures

The core innovation provided by scikit-learn for polynomial regression lies in the `PolynomialFeatures` class, located within the `preprocessing` module. This transformer object is responsible for generating the necessary polynomial features. It takes a matrix of original features and outputs a new matrix containing the original features raised to all powers up to the specified **degree**.

In the code snippet below, we initialize the PolynomialFeatures object, explicitly setting the `degree` to 3. This tells the transformer to create terms up to X^3. We also set `include_bias=False`. If set to `True` (the default), the transformer would add an intercept feature, which is usually unnecessary as the `LinearRegression` class handles the intercept ($beta_0$) automatically during the fitting process.

It is crucial to note that **scikit-learn** expects input data (features) to be in a 2D array format (samples x features). Since our original variable `x` is a 1D array, we must reshape it using `x.reshape(-1, 1)` before passing it to the `fit_transform()` method. This transformation step converts the 11 data points, each having one feature (x), into 11 data points, each having three features (x, x^2, x^3).

from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

# Specify degree of 3 for polynomial regression model
# include_bias=False means we do not force the y-intercept to equal zero in the feature matrix
poly = PolynomialFeatures(degree=3, include_bias=False)

# Reshape data to work properly with sklearn (must be 2D)
poly_features = poly.fit_transform(x.reshape(-1, 1))

Step 3: Fitting the Linear Regression Model to Transformed Data

Once the features have been transformed into their polynomial representations (`poly_features`), the problem reverts back to a standard linear regression task. This is the key insight of polynomial regression: we are fitting a linear relationship between the response variable Y and the newly created polynomial features (X^1, X^2, X^3).

We instantiate the standard `LinearRegression` object from `sklearn.linear_model`. This model employs the Ordinary Least Squares (OLS) method to find the optimal coefficients ($beta_1, beta_2, dots$) that minimize the sum of squared residuals between the predicted and actual response values.

The model is trained using the `fit()` method, taking the transformed features (`poly_features`) as the independent variables and the original response array (`y`) as the dependent variable. After the training completes, the model object (`poly_reg_model`) stores the optimized intercept ($beta_0$) and the array of coefficients ($beta_1, beta_2, beta_3$).

 # Fit polynomial regression model
poly_reg_model = LinearRegression()
poly_reg_model.fit(poly_features, y)

# Display model coefficients and intercept
print(poly_reg_model.intercept_, poly_reg_model.coef_)

33.62640037532282

The output reveals two critical components: the intercept ($beta_0$) and an array of coefficients. The intercept is 33.626, and the coefficients for X^1, X^2, and X^3 are approximately -11.839, 2.256, and -0.109, respectively. These values define the fitted curve and quantify the influence of each polynomial term on the response variable.

Interpreting Model Coefficients and the Regression Equation

The coefficient values derived from the fitted model allow us to explicitly write out the finalized cubic regression equation. This equation is the mathematical description of the relationship discovered by the model and is the primary tool for making future predictions.

Based on the intercept (33.626) and the coefficients (), the fitted polynomial regression equation is structured as follows:

y = -0.109x3 + 2.256x2 - 11.839x + 33.626

This equation now serves as our predictive function. By substituting any value for the predictor variable x into this formula, we can calculate the corresponding expected value for the response variable y. For instance, if we wanted to predict the response when $x=4$, we would calculate:

y = -0.109(4)3 + 2.256(4)2 - 11.839(4) + 33.626

Calculating this yields a predicted response value of approximately 15.39. This calculation demonstrates how the model uses the weighted contributions of the linear, quadratic, and cubic terms to arrive at a predicted outcome that aligns with the established non-linear trend in the data.

Important Consideration: The complexity of the model is entirely determined by the `degree` argument passed to the PolynomialFeatures function. Should the initial visualization or subsequent

residual analysis suggest that a quadratic (degree 2) or quartic (degree 4) model would be more appropriate, modifying only this single parameter is necessary to retrain the entire model architecture within **scikit-learn**. Selecting the optimal degree is often an iterative process involving cross-validation and minimizing prediction error.

Step 4: Making Predictions and Visualizing the Fit

To confirm the effectiveness of our trained polynomial model, the final step involves generating predictions across our input features and overlaying the resulting curve onto the original scatterplot. This visual inspection provides the most intuitive confirmation that the model has successfully captured the non-linear relationship.

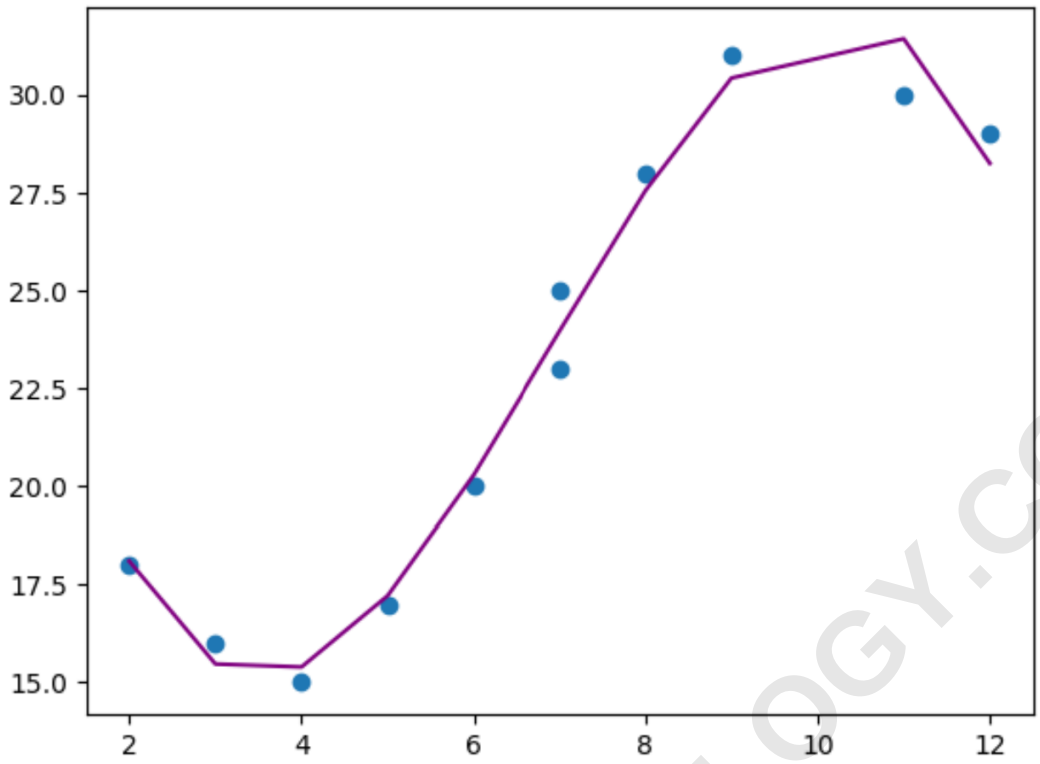
Using the `predict()` method on the `poly_reg_model`, we calculate the expected response values (`y_predicted`) using the same transformed features (`poly_features`) that were used during training. The predictions are then plotted as a continuous line using **Matplotlib**, typically distinguished by a contrasting color, such as purple, against the raw data points.

 # Use model to make predictions on response variable
y_predicted = poly_reg_model.predict(poly_features)

# Create scatterplot of original data (x vs. y)
plt.scatter(x, y)

# Add line to show the fitted polynomial regression model
plt.plot(x, y_predicted, color='purple')

The resulting visualization clearly shows the cubic curve fitting snugly around the original data points. Unlike a straight line that would miss the lower values and overestimate or underestimate the peak values, the polynomial curve adapts to the changing direction of the data.



Examination of the plot confirms that the degree 3 polynomial regression model offers an excellent fit, successfully minimizing the residuals across the range of the dataset. This high degree of conformity indicates that the steps taken--from data creation and non-linear feature transformation via PolynomialFeatures to the final fitting using **Linear Regression**--were executed correctly and yielded a robust predictive model for this non-linear relationship.

Advanced Considerations: Overfitting and Feature Scaling

While polynomial regression is highly effective for modeling complex curves, practitioners must be wary of two major pitfalls: **overfitting** and the necessity of **feature scaling**. Overfitting occurs primarily when an excessively high polynomial degree is chosen (e.g., degree 10 for a small dataset). Such a complex model will fit the training data almost perfectly, including random noise, but will perform poorly when generalizing to new, unseen data. To mitigate this, techniques like cross-validation should be employed to select the optimal degree that balances bias and variance.

Furthermore, when working with polynomial features, **feature scaling** becomes exceptionally important. Consider an original feature X with values ranging from 1 to 10. The squared term X^2 will range from 1 to 100, and the cubic term X^3 will range from 1 to 1000. These vastly different scales can destabilize the training process for the **Linear Regression** model, particularly if regularization methods like Ridge or Lasso are used.

To ensure numerical stability and convergence, it is best practice to standardize or normalize the original predictor variable X **before** applying the PolynomialFeatures transformation. Standard scaling, which transforms data to have a mean of zero and a unit variance, ensures that all generated polynomial features remain within a manageable scale, leading to a more stable and reliable model fit. scikit-learn offers the `StandardScaler` class within `sklearn.preprocessing` for this purpose.

Conclusion: Leveraging Scikit-learn for Non-Linear Modeling

The step-by-step process demonstrated how scikit-learn simplifies the implementation of complex statistical models like polynomial regression. By utilizing the **PolynomialFeatures** transformer, we can effortlessly generate the necessary higher-order terms, allowing the standard **Linear Regression** estimator to effectively model non-linear data patterns. This approach maintains the computational efficiency and interpretability inherent in linear models while expanding their descriptive power significantly.

The key takeaway is understanding the modularity of the **scikit-learn** pipeline: transformation (preprocessing) is decoupled from estimation (fitting). This separation allows users to chain multiple preprocessing steps--such as scaling, feature generation, and dimensionality reduction--before feeding the data into a final estimator. This flexibility is what makes scikit-learn the industry standard for machine learning in Python.

To deepen your expertise in advanced data modeling techniques using **scikit-learn**, consider exploring tutorials on related concepts. These methods often involve similar preprocessing pipelines:

Fitting models to handle multiple input variables (Multivariate Regression).
Implementing regularization techniques (Ridge or Lasso regression) to manage model complexity and prevent overfitting in high-degree polynomial models.
Using pipelines to automate feature engineering and model training processes seamlessly.

For comprehensive details on the transformation tools used here, the complete documentation for the `PolynomialFeatures` function is the authoritative source for understanding all available parameters and methods.