How to Fit a Curve to a Set of Points Using a Library

Authored by

stats writer

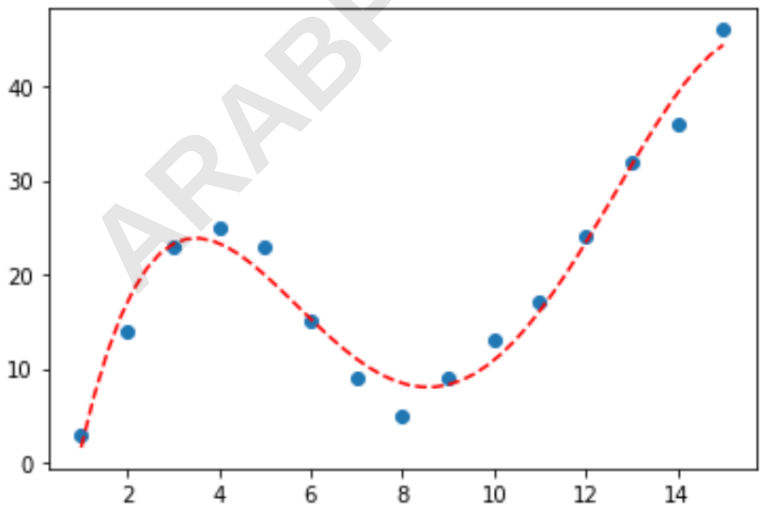
December 5, 2025

Yes, there is a specialized approach for implementing Curve fitting to a set of discrete data points. This methodology relies on robust mathematical libraries, often referred to as curve-fitting libraries, which employ a variety of sophisticated algorithms to generate a smooth, continuous curve that optimally minimizes the distance to the input data set. The utility of such capabilities is immense across diverse sectors, proving essential for tasks such as rigorous data analysis, advanced signal processing, and the development of statistical models within machine learning systems.

In modern computational environments, particularly using the Python programming language, the process of finding the best fit for a relationship between variables has been streamlined by powerful, open-source tools. This procedure moves beyond simple interpolation, aiming instead for generalization and prediction based on underlying trends within the data. Selecting the appropriate model--whether linear, exponential, or polynomial--is critical to ensure the resulting curve accurately reflects the phenomenon being studied without succumbing to pitfalls like **overfitting** or **underfitting**.

This comprehensive guide details the practical implementation of curve fitting in Python, utilizing industry-standard libraries like NumPy and Matplotlib. We will walk through the process of generating synthetic data, fitting multiple potential regression curves, evaluating their statistical validity using measures like the Adjusted R-squared, and ultimately visualizing the best predictive model.

The requirement to map complex relationships efficiently often arises when working with numerical datasets in Python. Unlike simple linear relationships, many real-world phenomena exhibit curved or non-linear patterns that demand higher-order models for accurate representation.



The subsequent, detailed example provides a practical, step-by-step methodology for executing curve fitting using core NumPy functions. Furthermore, it demonstrates how to rigorously apply

statistical metrics to determine which curve offers the statistically best fit for the analyzed dataset, ensuring the robustness and accuracy of the final model.

Data Initialization and Preliminary Visualization

The initial phase of any modeling task involves preparing the data environment and conducting exploratory visualization. For demonstration purposes, we begin by creating a synthetic dataset that clearly exhibits a non-linear, parabolic-like trajectory. This controlled setup allows us to easily test the performance of various fitting functions against known complexities. We utilize the Pandas library for structuring this data into a **DataFrame**, providing labeled access to our independent variable (x) and dependent variable (y).

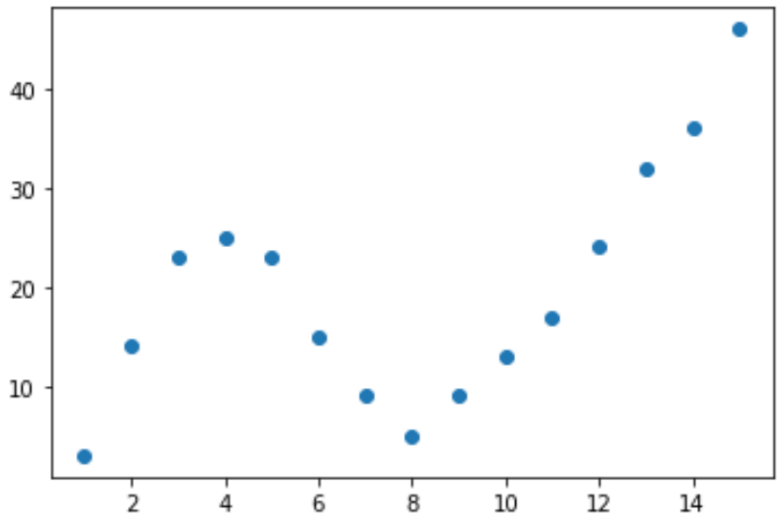
A crucial aspect of preliminary data analysis is visualization, which provides instantaneous insight into the underlying distribution and potential relationship between variables. By generating a scatterplot using Matplotlib, we can visually inspect the data points and hypothesize about the appropriate model structure. This visual inspection helps confirm that a linear model would be insufficient and justifies the use of higher-order polynomial regression techniques for better representation.

The provided code block below initializes the environment by importing the necessary libraries, defining the 15 data points (x and y), and subsequently generating the scatterplot visualization. Observe how the structure of the data necessitates a curve, rather than a straight line, to effectively capture the changing slope across the range of x-values.

```
import pandas as pd
import matplotlib.pyplot as plt

#create DataFrame
df = pd.DataFrame({'x': ,
'y': })

#create scatterplot of x vs. y
plt.scatter(df.x, df.y)
```



Implementing Polynomial Regression Models

Following the initial visual assessment, the next logical step is to fit multiple polynomial regression models to the data. Polynomial regression is a form of linear regression in which the relationship between the independent variable (x) and the dependent variable (y) is modeled as an n-th degree polynomial. By testing models ranging from degree 1 (simple linear) up to degree 5, we can systematically explore the complexity required to accurately capture the variance in the data.

The core functionality for performing these fits is provided by NumPy. Specifically, the `numpy.polyfit()` function calculates the coefficients that minimize the sum of squared errors between the observed data and the polynomial model, while `numpy.poly1d()` creates a callable polynomial object from those coefficients, allowing us to easily generate the curve values for plotting.

We must evaluate a range of degrees to avoid common modeling errors. A first- or second-degree polynomial might result in **underfitting**, failing to capture the inflection points in the data, thus yielding high bias. Conversely, a very high-degree polynomial might result in **overfitting**, capturing noise in the specific sample data rather than the true underlying trend, leading to poor generalization on new, unseen data. The goal is to find the optimal trade-off in complexity.

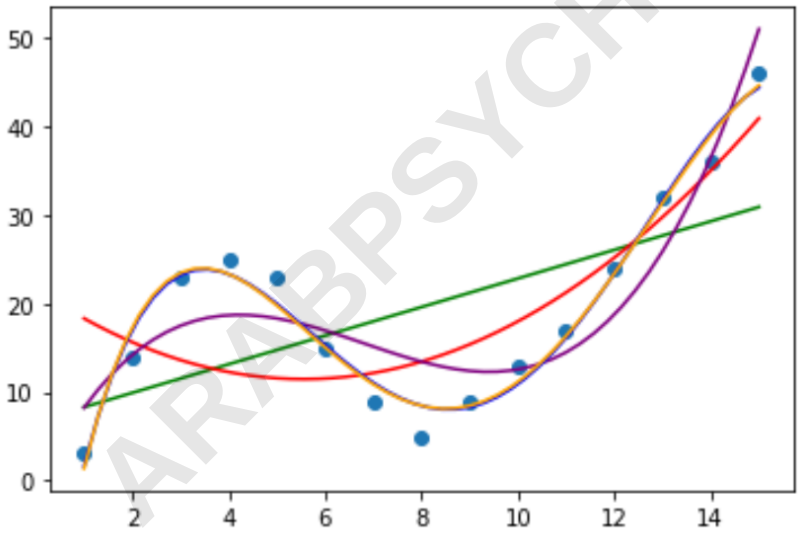
The code below executes the fitting process for five separate models. We then overlay the resulting curves onto the original scatterplot using Matplotlib's `plot()` function. The `numpy.linspace` function is used to create 50 evenly spaced points between 1 and 15, ensuring the fitted lines are smooth and continuous across the domain of the data.

import numpy as np

```
#fit polynomial models up to degree 5
model1 = np.poly1d(np.polyfit(df.x, df.y, 1))
model2 = np.poly1d(np.polyfit(df.x, df.y, 2))
model3 = np.poly1d(np.polyfit(df.x, df.y, 3))
model4 = np.poly1d(np.polyfit(df.x, df.y, 4))
model5 = np.poly1d(np.polyfit(df.x, df.y, 5))

#create scatterplot
polyline = np.linspace(1, 15, 50)
plt.scatter(df.x, df.y)

#add fitted polynomial lines to scatterplot
plt.plot(polyline, model1(polyline), color='green')
plt.plot(polyline, model2(polyline), color='red')
plt.plot(polyline, model3(polyline), color='purple')
plt.plot(polyline, model4(polyline), color='blue')
plt.plot(polyline, model5(polyline), color='orange')
plt.show()
```



Evaluating Model Fitness Using Adjusted R-squared

Once multiple curves have been fitted, the critical task is model selection--determining which equation provides the most statistically justifiable representation of the data. While the standard R-squared value is often used, it suffers from a significant drawback: it always increases as more predictor variables (or higher polynomial degrees) are added to the model, even if those variables

are statistically insignificant. This tendency leads to the selection of overly complex models.

To counteract this bias, we must rely on the Adjusted R-squared statistic. The Adjusted R-squared imposes a penalty for including unnecessary terms in the model. It modifies the standard R-squared calculation by taking into account the number of predictors (p) and the number of data points (n), ensuring that a new term is only accepted if it improves the model fit substantially enough to offset the loss of degrees of freedom.

The value derived from the Adjusted R-squared quantifies the percentage of the variance in the dependent variable (y) that is explained by the independent predictor variable(s) in the model, after adjusting for the complexity introduced by the number of predictors. A higher Adjusted R-squared, therefore, indicates a better-performing model that balances explanatory power with parsimony.

To calculate this metric for our polynomial models, we define a custom function, `adjR`, which takes the data (x, y) and the degree of the polynomial as inputs. Internally, this function calculates the Sum of Squares Regression (SSREG) and the Total Sum of Squares (SSTOT) to derive the standard R-squared, which is then mathematically corrected for the degrees of freedom (n-1) and the number of parameters (degree+1).

#define function to calculate adjusted r-squared
def adjR(x, y, degree):
results = {}
coeffs = np.polyfit(x, y, degree)
p = np.poly1d(coeffs)
yhat = p(x)
ybar = np.sum(y)/len(y)
ssreg = np.sum((yhat-ybar)2)**
sstot = np.sum((y - ybar)2)**
results = 1- (((1-(ssreg/sstot))*(len(y)-1))/(len(y)-degree-1))

return results

#calculated adjusted R-squared of each model
adjR(df.x, df.y, 1)
adjR(df.x, df.y, 2)
adjR(df.x, df.y, 3)
adjR(df.x, df.y, 4)
adjR(df.x, df.y, 5)

{'r_squared': 0.3144819}
{'r_squared': 0.5186706}

{'r_squared': 0.7842864}
{'r_squared': 0.9590276}
{'r_squared': 0.9549709}

Upon reviewing the output generated by the Adjusted R-squared calculations, a clear optimal model emerges. The model corresponding to the fourth-degree polynomial yields the highest Adjusted R-squared value, specifically **0.9590276**. This indicates that approximately 95.9% of the variance in the y-variable can be accurately explained by the fourth-degree polynomial relationship with x, making it the most appropriate and parsimonious model among the tested candidates. It is important to note that the fifth-degree polynomial, while slightly more complex, actually resulted in a marginally lower Adjusted R-squared (0.9549709), demonstrating the effectiveness of this metric in penalizing unnecessary complexity.

Step 3: Visualize the Optimal Curve

Having statistically identified the fourth-degree polynomial as the best fit, the final step involves generating a dedicated visualization of this optimal curve against the raw data points. This visualization serves as a crucial confirmation, allowing us to see how well the selected mathematical model adheres to the observed patterns across the entire dataset range. It verifies that the model selected based on the Adjusted R-squared metric does not visually exhibit signs of overfitting or underfitting.

We recalculate the fourth-degree model specifically, ensuring we use the coefficients derived from `numpy.polyfit` in NumPy. The visualization utilizes the scatterplot for the original data and the `plot()` function for the fitted curve. To distinguish the final model clearly, we apply specific formatting, such as a dashed line style and a prominent color like red. This clear visual presentation is essential for communicating the result of the curve fitting exercise to stakeholders.

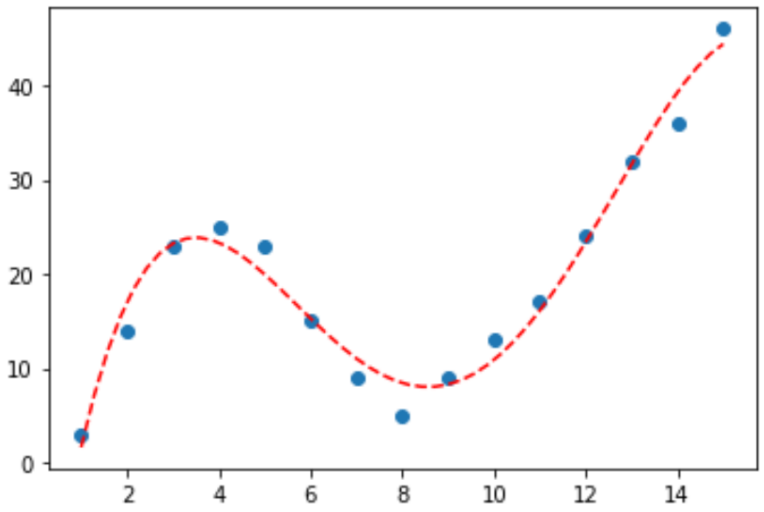
This final plot confirms that the fourth-degree curve successfully passes through the general trend of the data, accurately capturing the initial upward slope, the flattening around the midpoint, and the final sharp acceleration, confirming its suitability for prediction tasks across the defined domain.

#fit fourth-degree polynomial
model4 = np.poly1d(np.polyfit(df.x, df.y, 4))

#define scatterplot
polyline = np.linspace(1, 15, 50)
plt.scatter(df.x, df.y)

#add fitted polynomial curve to scatterplot
plt.plot(polyline, model4(polyline), '--', color='red')

```
plt.show()
```



Interpretation and Predictive Modeling

Beyond visual confirmation, the primary utility of polynomial regression lies in the derived equation, which encapsulates the mathematical relationship found by the fitting algorithm. We can easily retrieve this equation, including the calculated coefficients, using the `print()` function on the `poly1d` object generated by NumPy.

The printed output displays the coefficients associated with each power of x, starting from the highest degree (x4) down to the intercept. This provides a formal, quantitative description of the model. Understanding these coefficients is vital for interpreting the direction and magnitude of the influence of the independent variable on the outcome, although in high-degree polynomial models, direct coefficient interpretation can become mathematically complex.

print(model4)

4 3 2
-0.01924 x + 0.7081 x - 8.365 x + 35.82 x - 26.52

Translating the output from the NumPy object into standard algebraic notation, the equation of the optimal fourth-degree curve is formally expressed as:

y = **-0.01924x4 + 0.7081x3 - 8.365x2 + 35.82x - 26.52**

This derived equation is the operational output of the entire curve fitting process. It allows us to

perform precise prediction or estimation for any value of x within the observed range (interpolation) or even slightly outside it (cautious extrapolation). By substituting a specific value for the independent variable (x) into this formula, we can calculate the corresponding estimated value for the dependent variable (y). For instance, if we input $x = 4$ into the equation, we predict that the response variable *y* will be approximately **23.32**. This capability forms the foundation for applying these models in areas like forecasting and scientific estimation.

y = -0.0192(4)4 + 0.7081(4)3 - 8.365(4)2 + 35.82(4) - 26.52 = 23.32

Conclusion and Further Applications

The methodology outlined demonstrates a robust process for identifying the optimal non-linear fit for complex data structures using standard Python libraries. The key takeaway is the importance of balancing model complexity (polynomial degree) with model explanatory power, a balance effectively managed by the Adjusted R-squared metric.

While we focused on polynomial models, the same principles of evaluation and visualization apply to other non-linear techniques, such as exponential, logarithmic, or trigonometric fitting functions, often implemented using tools within the SciPy ecosystem. Mastering these fundamental data analysis techniques is essential for accurate modeling in fields ranging from quantitative finance to computational physics.