

## 4.5 Regression



Suppose we have a sample with two measurands: (1) the force  $F$  through a spring and (2) its displacement  $X$  (not from equilibrium).

We would like to determine an analytic function that relates the variables, perhaps for prediction of the force given another displacement.

There is some variation in the measurement. Let's say the following is the sample.

```
X_a = 1e-3 * np.array(
    [10, 21, 30, 41, 49, 50, 61, 71, 80, 92, 100]
) # m
F_a = np.array(
    [50.1, 50.4, 53.2, 55.9, 57.2, 59.9, 61.0, 63.9, 67.0, 67.9, 70.3]
) # N
```

Let's take a look at the data.

```
fig, ax = plt.subplots()
p = ax.plot(X_a * 1e3, F_a, '.b', markersize=15)
ax.set_xlabel(r'$X$ (mm)')
ax.set_ylabel(r'$F$ (N)')
ax.set_xlim([0, np.max(X_a * 1e3)])
ax.grid(True)
plt.draw()
```

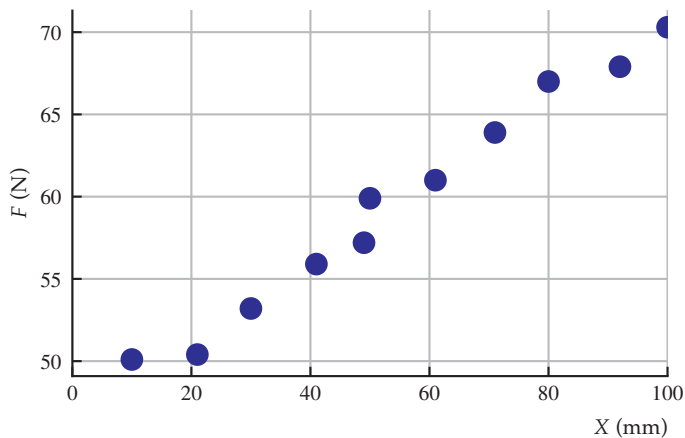


Figure 4.13. Force  $F$  as a function of displacement  $X$ .

How might we find an analytic function that agrees with the data? Broadly, our strategy will be to assume a general form of a function and use the data to set

the parameters in the function such that the difference between the data and the function is minimal.

Let  $y$  be the analytic function that we would like to fit to the data. Let  $y_i$  denote the value of  $y(x_i)$ , where  $x_i$  is the  $i$ th value of the random variable  $X$  from the sample. Then we want to minimize the differences between the force measurements  $F_i$  and  $y_i$ .

From calculus, recall that we can minimize a function by differentiating it and solving for the zero-crossings (which correspond to local maxima or minima).

First, we need such a function to minimize. Perhaps the simplest, effective function  $D$  is constructed by squaring and summing the differences we want to minimize, for sample size  $N$ :

$$D(x_i) = \sum_{i=1}^N (F_i - y_i)^2$$

(recall that  $y_i = y(x_i)$ , which makes  $D$  a function of  $x$ ).

Now the form of  $y$  must be chosen. We consider only  $m$ th-order polynomial functions  $y$ , but others can be used in a similar manner:

$$y(x) = a_0 + a_1x + a_2x^2 + \dots + a_mx^m.$$

If we treat  $D$  as a function of the polynomial coefficients  $a_j$ , i.e.

$$D(a_0, a_1, \dots, a_m),$$

and minimize  $D$  for each value of  $x_i$ , we must take the partial derivatives of  $D$  with respect to each  $a_j$  and set each equal to zero:

$$\frac{\partial D}{\partial a_0} = 0, \quad \frac{\partial D}{\partial a_1} = 0, \quad \dots, \quad \frac{\partial D}{\partial a_m} = 0.$$

This gives us  $N$  equations and  $m + 1$  unknowns  $a_j$ . Writing the system in matrix form,

$$\underbrace{\begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^m \\ 1 & x_2 & x_2^2 & \dots & x_2^m \\ 1 & x_3 & x_3^2 & \dots & x_3^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_N & x_N^2 & \dots & x_N^m \end{bmatrix}}_{A_{N \times (m+1)}} \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix}}_{\mathbf{a}_{(m+1) \times 1}} = \underbrace{\begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ \vdots \\ F_N \end{bmatrix}}_{\mathbf{b}_{N \times 1}}.$$

Typically  $N > m$  and this is an *overdetermined system*. Therefore, we usually can't solve by taking  $A^{-1}$  because  $A$  doesn't have an inverse!

Instead, we either find the *Moore-Penrose pseudo-inverse*  $A^\dagger$  and have  $\mathbf{a} = A^\dagger \mathbf{b}$  as the solution, which is *inefficient* (even with NumPy's `linalg.pinv()` function)—or we

can approximate  $\mathbf{b}$  with an algorithm such as that used in the *least-squares* method, which has Numpy function `linalg.lstsq()`. We'll use the latter method.

### Example 4.5

Use Numpy to find the least-squares polynomial fit for the sample. There's the sometimes-difficult question, "What order should we fit?" Let's try out several and see what the squared-differences function  $D$  gives.

Begin by writing a function that takes the sample data and the order of the polynomial fit and returns the coefficients of the polynomial.

```
def poly_fit(X, F, order):
    A = np.vander(X, order + 1, increasing=True) # Vandermonde matrix
    # This is the matrix A in the system of equations
    return np.linalg.lstsq(A, F, rcond=None)[0] # Coefficients
```

Fit the data with polynomials of orders 1, 3, 5, 7, and 9.

```
orders = [1, 3, 5, 7, 9]
coefficients = [poly_fit(X_a, F_a, order) for order in orders]
```

Now we can plot the data and the fitted polynomials.

```
fig, ax = plt.subplots()
p = ax.plot(X_a * 1e3, F_a, '.b', markersize=15)
x = np.linspace(np.min(X_a), np.max(X_a), 100)
for i, order in enumerate(orders):
    y = np.polyval(coefficients[i][::-1], x)
    ax.plot(x * 1e3, y, label=f'Order {order}')
ax.set_xlabel(r'$X$ (mm)')
ax.set_ylabel(r'$F$ (N)')
ax.legend()
plt.draw()
```

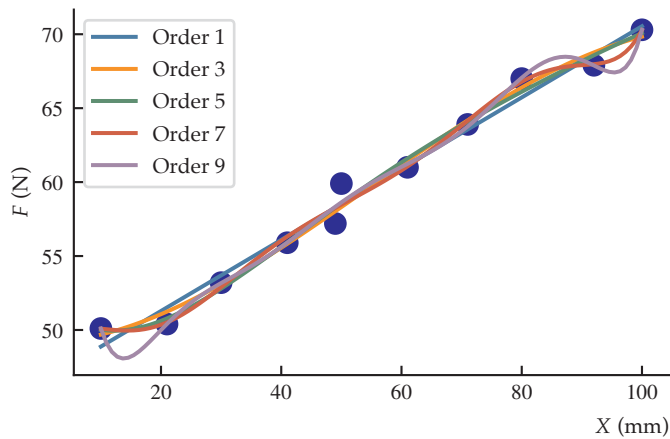


Figure 4.14. Data and fitted polynomials of different orders.

The plot shows the data points and the fitted polynomials of different orders. The higher-order polynomials seem to fit the data better, but they may be overfitting. We can quantify the goodness of fit by calculating the sum of squared differences  $D$  for each order.

```
D = []
for i, order in enumerate(orders):
    y = np.polyval(coefficients[i][::-1], X_a)
    D.append(np.sum((F_a - y) ** 2))
```

Let's plot the sum of squared differences as a function of the order of the polynomial.

```
fig, ax = plt.subplots()
p = ax.plot(orders, D, '-b')
ax.set_xlabel('Order of polynomial')
ax.set_ylabel(r'$D(a_0, a_1, \dots, a_m)$')
ax.set_xticks(orders)
plt.show()
```

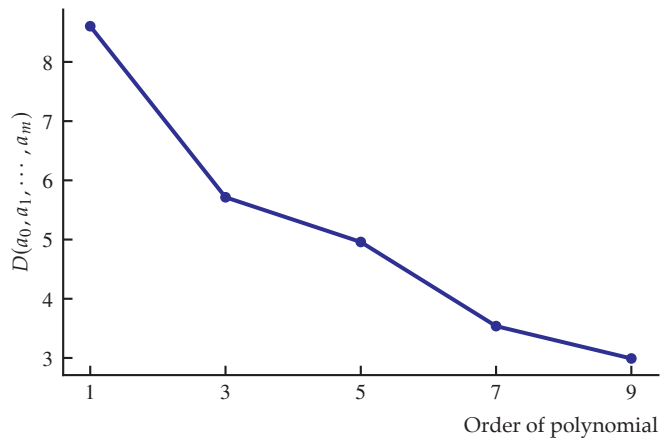



Figure 4.15. Sum of squared differences as a function of polynomial order.

The plot shows that the sum of squared differences decreases with the order of the polynomial. However, the decrease is less pronounced for higher-order polynomials. This suggests that the higher-order polynomials are overfitting the data. The optimal order of the polynomial is the one that gives the best fit without overfitting.

## 4.6 Problems



**Problem 4.1**  BREW You need to know the duration of time a certain stage of a brewing process takes. You set up an automated test environment that repeats the test 100 times, recorded in the following JSON<sup>1</sup> data file: <https://math.ricopic.one/bt>. Perform the following analysis.

- Download and parse the JSON file (it contains a single array).
- Estimate the duration of the process from the sample.
- Choose and justify an assumed probability density function for the random variable duration.
- Use this PDF model to compute a 99 percent confidence interval for your duration estimate.
- Compute your duration confidence interval for the range of confidence values [85, 99.99] percent.<sup>2</sup>
- Plot the confidence intervals over the range of confidence in said intervals.

**Problem 4.2**  LABORATORIUM Use linear regression techniques to find the values of  $a$ ,  $b$ ,  $c$ , and  $d$ , in a cubic function of the form,

$$f(x) = ax^3 + bx^2 + cx + d,$$

using the data below.

$x$	$f(x)$
-2.0	-4.7
-1.5	-1.9
-1.0	1.5
-0.5	1.5
0.0	1.4
0.6	0.3
1.1	-1.5
1.6	0.0
2.1	0.6
2.6	4.2

1. JSON is a simple and common programming language-independent data format. For parsing it with Matlab, see `jsondecode` here: <https://math.ricopic.one/75>. For parsing it with Python, see the module `json` here: <https://math.ricopic.one/jb>.

2. Consider using a  $z$ - or  $t$ -score inverse CDF lookup function like `t.ppf` from `scipy.stats`.