# Polynomial Regression

## Contents



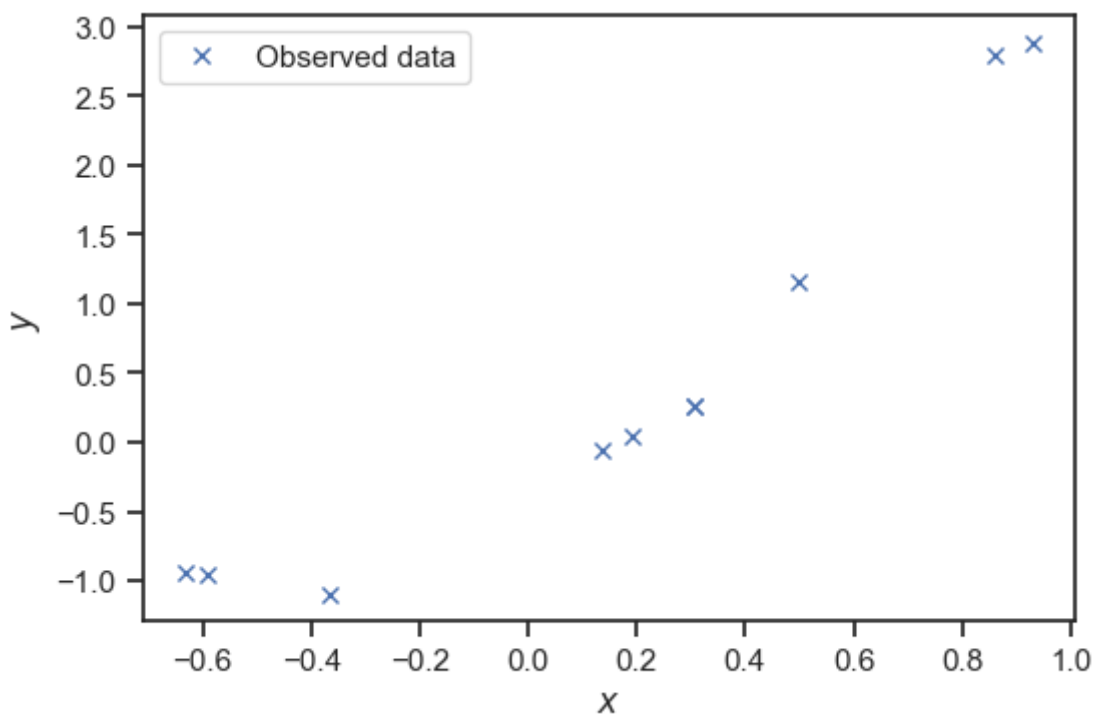
```
import numpy as np
np.set_printoptions(precision=3)
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
sns.set(rc={"figure.dpi":100, "savefig.dpi":300})
sns.set_context("notebook")
sns.set_style("ticks")
```

We take up where we left in the previous section. Recall that we tried to fit a linear regression model to data generated from:

$$y_i = -0.5 + 2x_i + 2x_i^2 + \epsilon_i,$$

where $\epsilon_i \sim N(0, 1)$ and where we sample $x_i \sim U([-1, 1])$:

```
np.random.seed(12345)

num_obs = 10
x = -1.0 + 2 * np.random.rand(num_obs)
w0_true = -0.5
w1_true = 2.0
w2_true = 2.0
sigma_true = 0.1
y = (
    w0_true
    + w1_true * x
    + w2_true * x ** 2
    + sigma_true * np.random.randn(num_obs)
)

fig, ax = plt.subplots()
ax.plot(x, y, 'x', label='Observed data')
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
plt.legend(loc='best');
```




We already saw that the linear model does not work here. We need to try to fit a quadratic model:

$$y = w_0 + w_1 x + w_2 x^2$$

$$y = w_0 + w_1 x + w_2 x^2.$$

How can we do this? Of course, by minimizing the square loss:

$$L(w_0, w_1, w_2) = \sum_{i=1}^{N} (y_i - w_0 - w_1 x_i - w_2 x_i^2)^2.$$

Fortunately, we do not have to do things from scratch. The notation we developed previously comes to our resque. Recall, that $\mathbf{y} = (y_1, \ldots, y_N)$ is the vector of observations. Use

$$\mathbf{w} = (w_0, w_1, w_2),$$

to denote the weight vector. What about the design matrix? Before it was an $N \times 2$ matrix with the first column being one and the second column being the vector of observed inputs. Well, now it is the $N \times 3$ matrix. The first two columns are exactly like before, but now the third column is the observed inputs squared. So, it is:

$$\mathbf{X} = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \\ 1 & x_N & x_N^2 \end{bmatrix}.$$

As before, if you multiply the design matrix $\mathbf{X}$ with the weight vector $\mathbf{w}$ you get the predictions of our model. So, again, the square loss can be written as:

$$L(w_0, w_1, w_2) = L(\mathbf{w}) = \| \mathbf{y} - \mathbf{X}\mathbf{w} \|^2.$$

Well, this is mathematically the same equation as before. The only difference is that we have 3-dimensional weight vector (instead of a 2-dimensional) and that the design matrix is $N \times 3$ instead of $N \times 2$. If you take the gradien of this with respect to $\mathbf{w}$ and set it equal to zero you will get that you need to solve exactly the same linear system of equations as before (but now it is 3 equations for 3 unknowns instead of 2 equations for 2 unknowns).

Let's solve it numerically. First, the design matrix:

```
X = np.hstack(
    [
        np.ones((num_obs, 1)),
        x.reshape((num_obs, 1)),
        x.reshape((num_obs, 1)) ** 2
    ]
)
X
```

```
array([[ 1.   ,  0.859,  0.738],
       [ 1.   , -0.367,  0.135],
       [ 1.   , -0.632,  0.4  ],
       [ 1.   , -0.591,  0.349],
       [ 1.   ,  0.135,  0.018],
       [ 1.   ,  0.191,  0.037],
       [ 1.   ,  0.929,  0.863],
       [ 1.   ,  0.306,  0.094],
       [ 1.   ,  0.498,  0.248],
       [ 1.   ,  0.307,  0.094]])
```

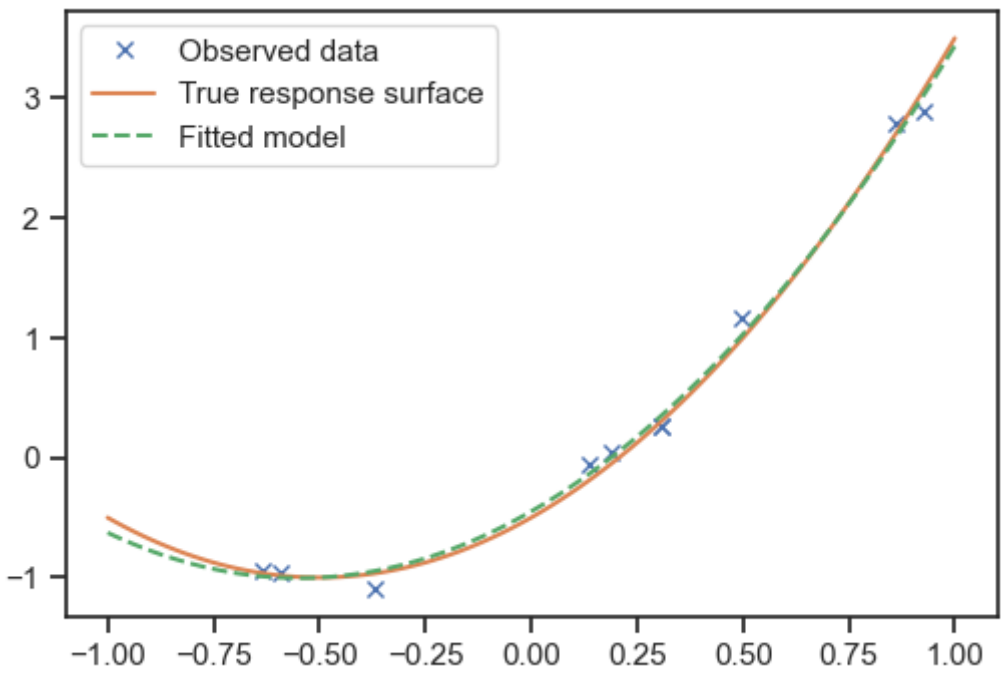
and then:

```
w, _, _, _ = np.linalg.lstsq(X, y, rcond=None)

print(f"w = {w}")
```

```
w = [-0.448  2.03   1.853]
```

Let's visualize the model predictions:

```
xx = np.linspace(-1, 1, 100)
yy_true = w0_true + w1_true * xx + w2_true * xx ** 2
yy = w[0] + w[1] * xx + w[2] * xx ** 2

fig, ax = plt.subplots()
ax.plot(x, y, 'x', label='Observed data')
ax.plot(xx, yy_true, label='True response surface')
ax.plot(xx, yy, '--', label='Fitted model')
plt.legend(loc='best');
```




## Questions

- Repeat with very small `num_obs` and very large `num_obs` and observe the behavior of the fit.

# Regression with high-degree polynomials and overfitting

What would have happened if we tried to use a higher degree polynomial. To achieve this, we need to be able to evaluate a design matrix of the form:

$$\mathbf{X} = \begin{bmatrix} 1 & x_1 & x_1^2 \dots & x_1^\rho \\ 1 & x_2 & x_2^2 \dots & x_2^\rho \\ \vdots & \vdots \dots & \vdots & \\ 1 & x_N & x_N^2 \dots & x_N^\rho \end{bmatrix},$$

where $\rho$ is the degree of the polynomial. The linear system we need to solve is the same as before. Only the weight vector is now $\rho + 1$ dimensional and the design matrix $N \times (\rho + 1)$.

Let's write some code to find the design matrix:

```
def get_polynomial_design_matrix(x, degree):
    """Return the polynomial design matrix of ``degree`` evaluated at ``x``.

    Arguments:
    x      -- A 2D array with only one column.
    degree -- An integer greater than zero.
    """
    # Make sure this is a 2D numpy array with only one column
    assert isinstance(x, np.ndarray), 'x is not a numpy array.'
    assert x.ndim == 2, 'You must make x a 2D array.'
    assert x.shape[1] == 1, 'x must be a column.'
    # Start with an empty list where we are going to put the columns of the matrix
    cols = []
    # Loop over columns and add the polynomial
    for i in range(degree+1):
        cols.append(x ** i)
    return np.hstack(cols)
```

raise column vector x to power that increases by the loop, each column is appended to the design matrix

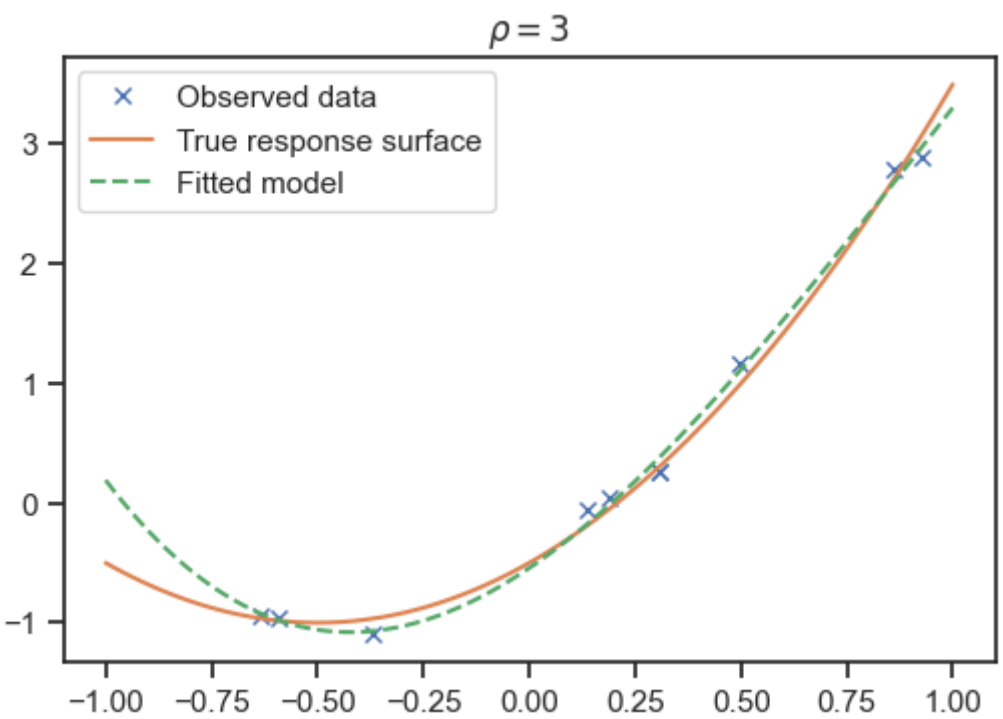
Let's try fitting a degree 3 polynomial and see what we get:

```
# The polynomial degree
degree = 3

# The design matrix is:
X = get_polynomial_design_matrix(x[:, None], degree)

# And we fit just like previously:
w, _, _, _ = np.linalg.lstsq(X, y, rcond=None)
print(f"w = {w}")
```

```
w = [-0.543  2.383  2.29  -0.827]
```

Let's visualize the fit. Notice, that for making predictions I am evaluating the design matrix on the points I want to make predictions at.

```
# Make predictions like this:
xx = np.linspace(-1, 1, 100)
XX = get_polynomial_design_matrix(xx[:, None], degree)
yy = XX @ w

# The true response
yy_true = w0_true + w1_true * xx + w2_true * xx ** 2

# Plot everything
fig, ax = plt.subplots()
ax.plot(x, y, 'x', label='Observed data')
ax.plot(xx, yy_true, label='True response surface')
ax.plot(xx, yy, '--', label='Fitted model')
ax.set_title(f'$\\rho = {degree}$')
plt.legend(loc='best');
```




# Questions

- Start increasing the polynomial degree from 3, to 4, to a number where things get bad... You will soon start *overfitting*.

---

By Ilias Bilionis (ibilion[at]purdue.edu)