Linear regression with a single variable

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")
```

Objectives

- To introduce linear regression with a single variable

An example where things work as expected

Let's create a synthetic dataset to introduce the basic concepts. It has to be synthetic because we want to know what the ground truth is. Let's start with pairs of x and y which definitely have a linear relationship, albeit y may be contaminated with Gaussian noise. In particular, we generate the data from:

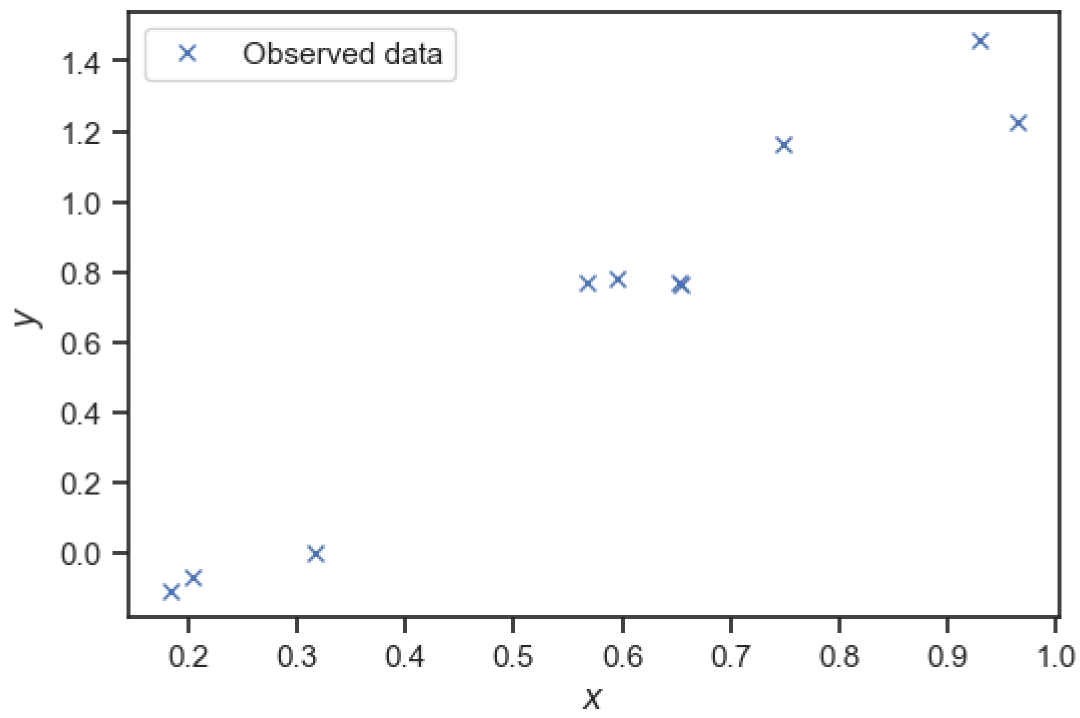
$$y_i = -0.5 + 2x_i + 0.1\epsilon_i,$$

where $\epsilon_i \sim N(0,1)$ and where we sample $x_i \sim U([0,1])$. Here is how to generate this synthetic dataset and how it looks like.

```
np.random.seed(12345)

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

Let's plot the data:

```
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 will now use least squares to fit the data to this linear model:

$$y = w_0 + w_1 x.$$

As we discussed in the previous section, least squares minimize the square loss:

$$L(\mathbf{w}) = \sum_{i=1}^{N}(y_i - w_0 - w_1 x_i)^2 = \parallel \mathbf{y} - \mathbf{X}\mathbf{w} \parallel^2,$$

where $\mathbf{y} = (y_1, \dots, y_N)$ is the vector of observations, $\mathbf{w} = (w_0, w_1)$ is the weight vector, and the $N \times 2$ *design matrix* $\mathbf{X}$ is:

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

We definitely need to make the design matrix $\mathbf{X}$:

```
# Put together a column of ones next to the observed x's
X = np.hstack(
    [np.ones((num_obs, 1)), x.reshape((num_obs, 1))]
)
X
```

```
array([[1.   , 0.93 ],
       [1.   , 0.316],
       [1.   , 0.184],
       [1.   , 0.205],
       [1.   , 0.568],
       [1.   , 0.596],
       [1.   , 0.965],
       [1.   , 0.653],
       [1.   , 0.749],
       [1.   , 0.654]])
```

Once we have this, we can use [numpy.linalg.lstsq](https://numpy.org) to solve the least squares problem. This function solves in a smart way the linear system we derived in the previous section, i.e.,

See this page for a lot more information on linear regression: https://en.wikipedia.org/wiki/Least_squares

$$\mathbf{X}^T\mathbf{X}\mathbf{w} = \mathbf{X}^T\mathbf{y}.$$

Start point: y = Xw, isolate for w

It works as follows:

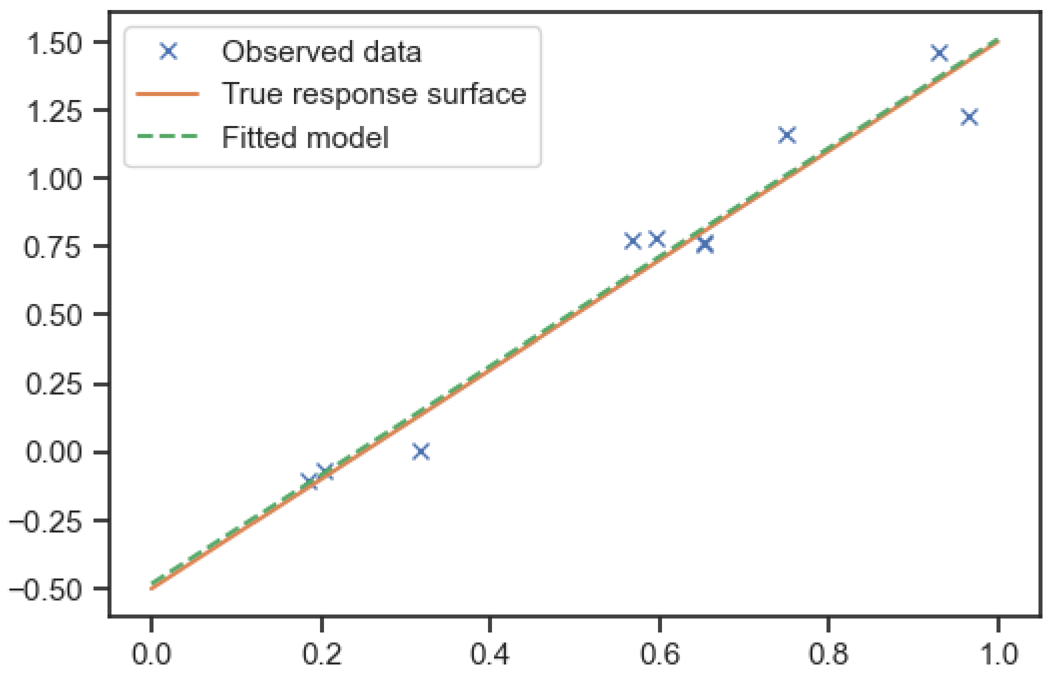
```
w, _, _, _ = np.linalg.lstsq(X, y, rcond=None)
print(f'w_0 = {w[0]:1.2f}')
print(f'w_1 = {w[1]:1.2f}')
```

```
w_0 = -0.48
w_1 = 1.99
```

So, you see that the values we found for w_0 and w_1 are close to the correct values, but not exactly the same. That is fine. There is noise in the data and we have only used ten observations. The more noise there is, the more observations it would take to identify the regression coefficients correctly.

Let's now plot the regression function against the data:

```
# Make predictions
# Some points on which to evaluate the regression function
xx = np.linspace(0, 1, 100)
# The true connection between x and y
yy_true = w0_true + w1_true * xx
# The model we just fitted
yy = w[0] + w[1] * xx

# Plot them
fig, ax = plt.subplots()
# plot the data again
ax.plot(x, y, 'x', label='Observed data')
# overlay the true
ax.plot(xx, yy_true, label='True response surface')
# overlay our prediction
ax.plot(xx, yy, '--', label='Fitted model')
plt.legend(loc='best');
```



Questions

- Try increasing `num_obs` to 100. Does the fit improve? Conclusion: When you training with least squares, the more data you have the better.
- Try decreasing `num_obs` to 2. What is happening here? This is an example of fitting the noise.

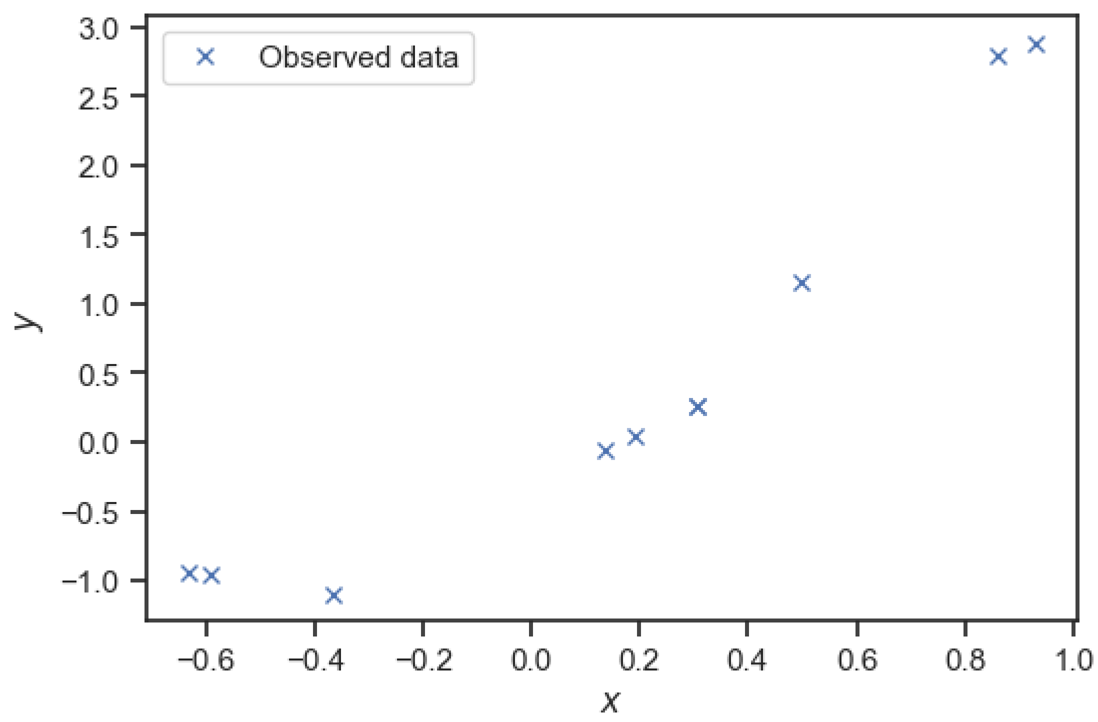
An example where things do not work as expected: underfitting

Let's try 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)  method to generate uniform random numbers between -1 and 1
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');
```

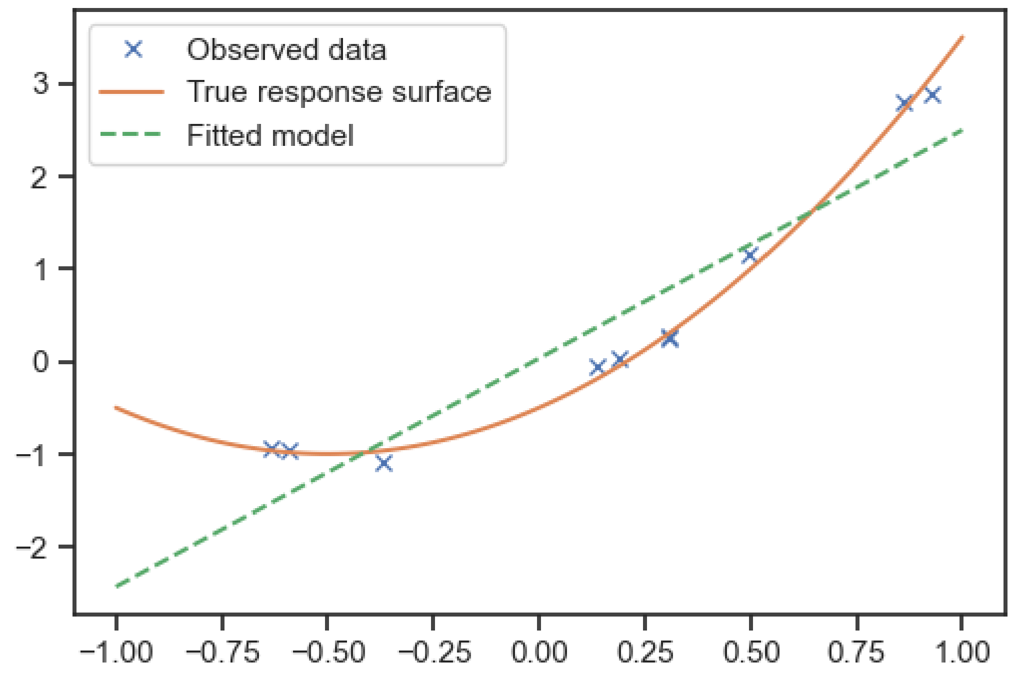


We will still try to fit a linear model to this dataset. We know that it is not going to work well, but let's try it anyway. First, create the design matrix just like before:

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

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

print(f'w_0 = {w[0]:1.2f}')
print(f'w_1 = {w[1]:1.2f}')
```

```
w_0 = 0.03
w_1 = 2.46
```

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

# Plot them
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

- Experiment with very small `num_obs`. If you did not know what the true response surface was, would you be able to say whether or not the fit is good?
- Experiment with a big `num_obs`. Does the fit improve? This is an example of *underfitting*. Your model does not have enough expressivity to capture the data.

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