Maximum a Posteriori Estimate - Avoiding Overfitting

Contents

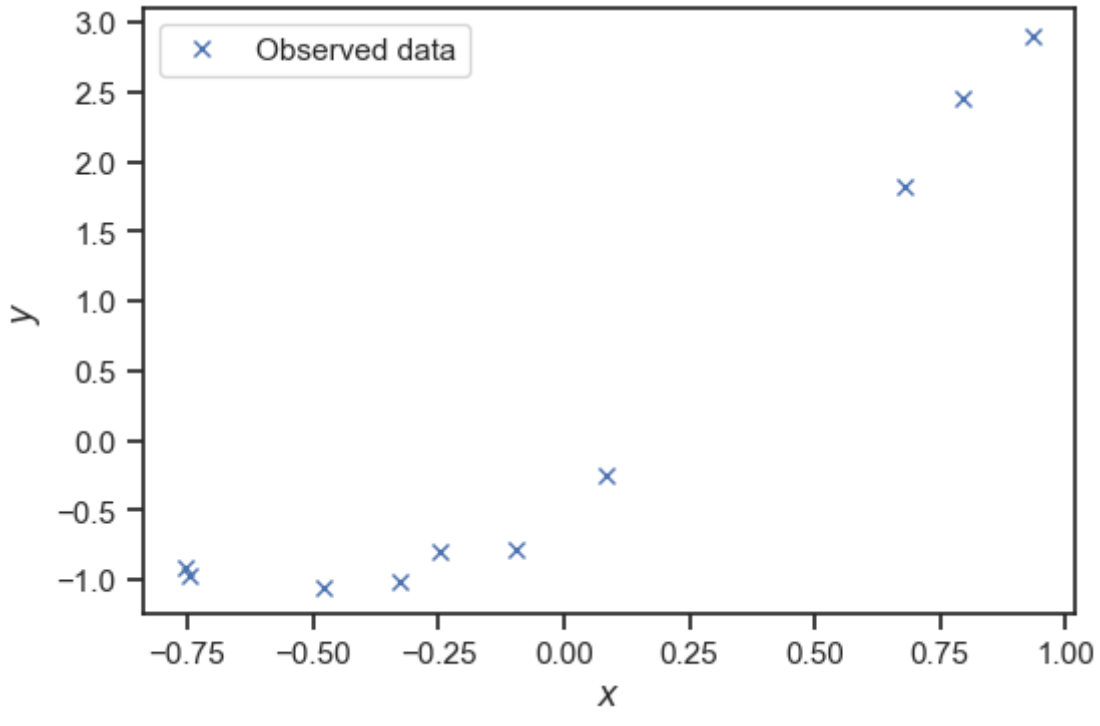

```
import numpy as np
np.set_printoptions(precision=3)
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
sns.set(rc={"figure.dpi":100, "savefig.dpi":300})
sns.set_context("notebook")
sns.set_style("ticks")
```

Objectives

- To demonstrate how regularization arises naturally from a Bayesian perspective and how it can be used to avoid overfitting.

Let's load the motorcyle data to play with:

```
np.random.seed(123456)

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');
```



Let's also copy-paste the code for creating design matrices for the three generalized linear models we have considered so far:

```
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.
    """
    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.'
    cols = []
    for i in range(degree+1):
        cols.append(x ** i)
    return np.hstack(cols)

def get_fourier_design_matrix(x, L, num_terms):
    """Fourier expansion with ``num_terms`` cosines and sines.

    Arguments:
    x          --  A 2D array with only one column.
    L          --  The "length" of the domain.
    num_terms  --  How many Fourier terms do you want.
                   This is not the number of basis
                   functions you get. The number of basis functions
                   is 1 + num_terms / 2. The first one is a constant.
    """
    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.'
    N = x.shape[0]
    cols = [np.ones((N, 1))]
    for i in range(int(num_terms / 2)):
        cols.append(np.cos(2 * (i+1) * np.pi / L * x))
        cols.append(np.sin(2 * (i+1) * np.pi / L * x))
    return np.hstack(cols)

def get_rbf_design_matrix(x, x_centers, ell):
    """Radial basis functions design matrix.

    Arguments:
    x         --  The input points on which you want to evaluate the
                  design matrix.
    x_center  --  The centers of the radial basis functions.
    ell       --  The lengthscale of the radial basis function.
    """
    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.'
    N = x.shape[0]
    cols = [np.ones((N, 1))]
    for i in range(x_centers.shape[0]):
        cols.append(np.exp(-(x - x_centers[i]) ** 2 / ell))
    return np.hstack(cols)
```

As we saw in the video lecture, the problem that we need to solve to find the maximum a posteriori estimate is when the prior of the weights is a zero mean Gaussian with precision α is:

$$\left(\sigma^{-2}\mathbf{\Phi}^T\mathbf{\Phi} + \alpha\mathbf{I}\right)\mathbf{w} = \sigma^{-2}\mathbf{\Phi}^T\mathbf{y}.$$

This problem cannot be solved by `numpy.linalg.lstsq`. The stable way to solve this problem is this:

- Construct the positive-definite matrix:

$$\mathbf{A} = \left(\sigma^{-2}\mathbf{\Phi}^T\mathbf{\Phi} + \alpha\mathbf{I}\right)$$

- Compute the Cholesky decomposition of $\mathbf{A}$:

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T,$$

where $\mathbf{L}$ is lower triangular.

- Then, solve the system:

$$\mathbf{L}\mathbf{L}^T\mathbf{w} = \sigma^{-2}\mathbf{\Phi}^T\mathbf{y}_{1:n},$$

doing a forward and a backward substitution. The methods scipy.linalg.cho_factor and scipy.linalg.cho_solve can be used for this. We

implement this process below:

```
# I need this to take the Cholesky decomposition
import scipy

def find_w_map(Phi, y, sigma2, alpha):
    """Return the MAP weights of a Bayesian linear regression problem.

    Arguments
    Phi    -- The design matrix.
    y      -- The observed targets.
    sigma2 -- The noise variance.
    aplha  -- The prior weight precision.
    """
    A = (
        Phi.T @ Phi / sigma2
        + alpha * np.eye(Phi.shape[1])
    )
    L = scipy.linalg.cho_factor(A)
    w = scipy.linalg.cho_solve(
        L,
        Phi.T @ y / sigma2
    )
    return w
```

Let's apply this to the synthetic dataset. Select polynomial degree and get design matrix:

```
degree = 6
Phi = get_polynomial_design_matrix(x[:, None], degree)
```

Pick variance (here I am using the true one):

```
sigma2 = 0.1 ** 2
```

Pick the regularization parameter:

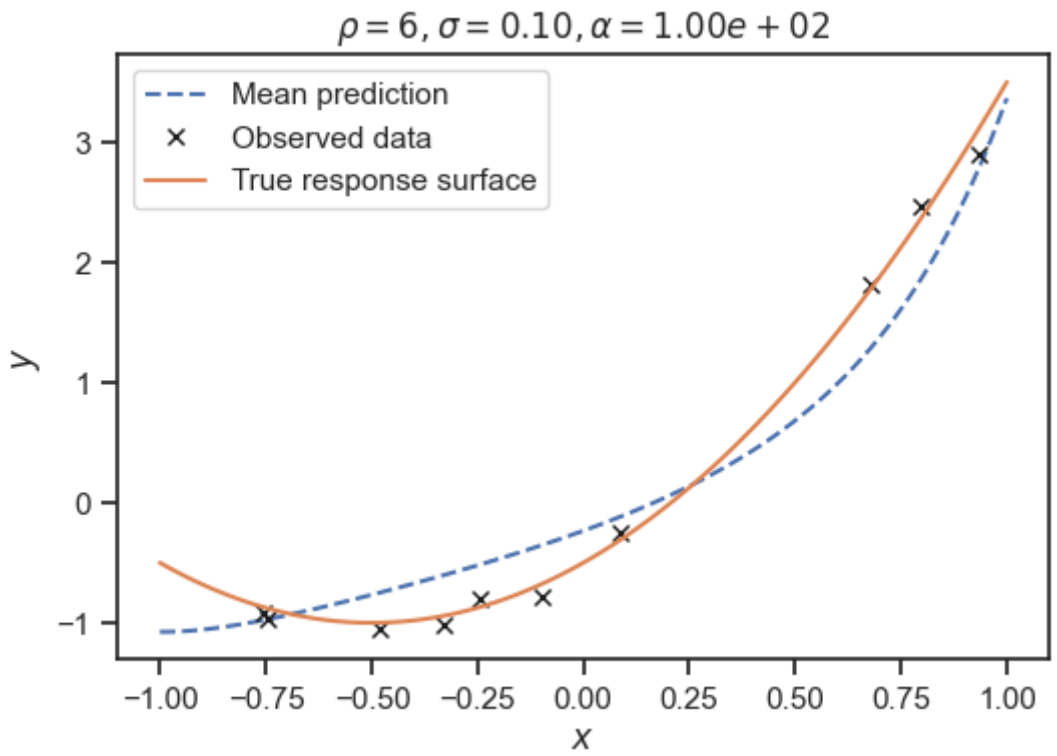
```
alpha = 100.0
```

Solve for the MAP of the weights:

```
w = find_w_map(Phi, y, sigma2, alpha)
```

Make predictions and plot the results:

```
xx = np.linspace(-1, 1, 100)
yy_true = w0_true + w1_true * xx + w2_true * xx ** 2
Phi_xx = get_polynomial_design_matrix(
    xx[:, None],
    degree
)
yy = Phi_xx @ w

fig, ax = plt.subplots()
ax.plot(xx, yy, '--', label='Mean prediction')
ax.plot(x, y, 'kx', label='Observed data')
ax.plot(xx, yy_true, label='True response surface')
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
ax.set_title(
    f'$\\rho={degree}, '
    + f'\\sigma = {np.sqrt(sigma2):1.2f}, '
    + f'\\alpha={alpha:1.2e}$'
)
plt.legend(loc='best');
```



Questions

- Experiment with different α's. Notice that for very small α's we are overfitting. Notice that with very large α's, we are underfitting.

Selecting α through a validation set

Let's plot the mean square error of a validation dataset as a function of α.

First, here is our validation dataset:

```
x_valid = -1.0 + 2 * np.random.rand(20)
y_valid = (
    w0_true
    + w1_true * x_valid
    + w2_true * x_valid ** 2
    + sigma_true * np.random.randn(x_valid.shape[0])
)
```

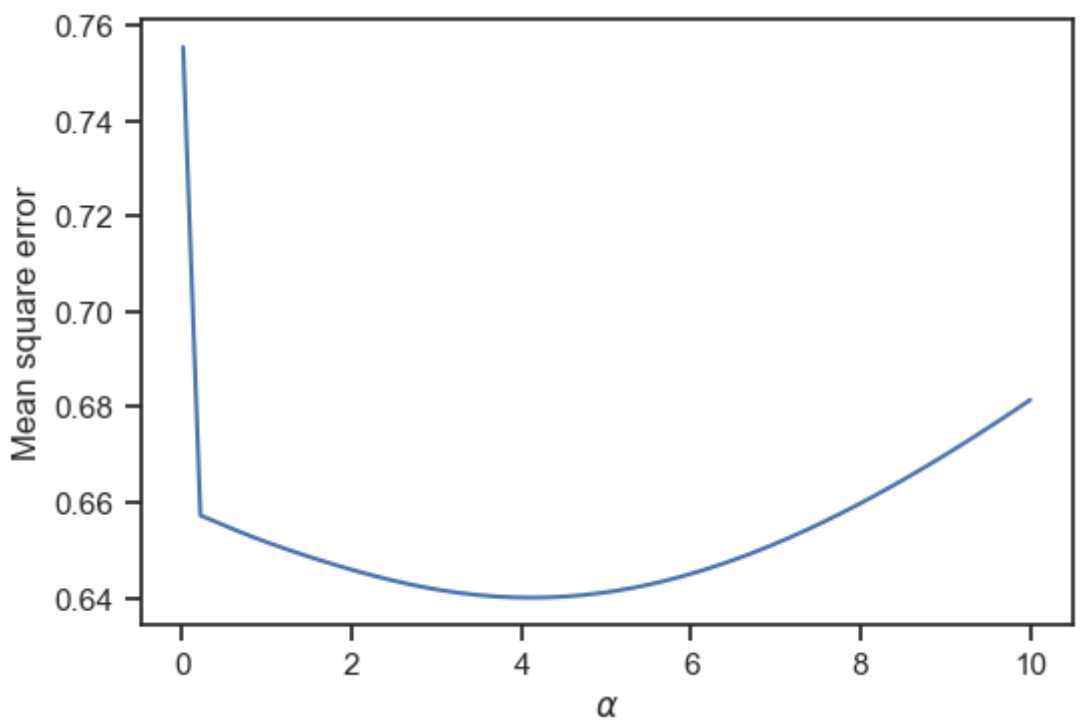
We are going to take 50 α's between 0.01 and 5 and try each one of them out.

```
num_alphas = 50
min_alpha = 0.01
max_alpha = 10

# The alphas we will try
alphas = np.linspace(
    min_alpha,
    max_alpha,
    num_alphas
)

# To store the MSE:
MSEs = np.ndarray((num_alphas,))

# To store the weights
weights = np.ndarray((num_alphas, degree + 1))

for i, alpha in enumerate(alphas):
    w = find_w_map(Phi, y, sigma2, alpha)
    Phi_valid = get_polynomial_design_matrix(
        x_valid[:, None],
        degree
    )
    y_valid_predict = Phi_valid @ w
    MSE_alpha = np.linalg.norm(
        y_valid_predict - y_valid
    )
    MSEs[i] = MSE_alpha
    weights[i] = w

print("alpha\t MSE\t Weights")
print("-" * 60)
for alpha, mse, w in zip(alphas, MSEs, weights):
    print(
        f"{alpha:.3f}\t {mse:.3f}\t"
        + "[" +
        " ".join(
            [f"{wi:.2f}" for wi in w]
        )
        + "]"
    )
```

```
alpha    MSE     Weights
-----------------------------------------------------------
0.010    0.755  [-0.49 1.99 1.41 0.25 2.49 -0.12 -2.56]
0.214    0.657  [-0.50 2.05 1.90 0.39 0.56 -0.59 -0.72]
0.418    0.656  [-0.49 2.07 1.90 0.34 0.46 -0.56 -0.59]
0.622    0.654  [-0.49 2.07 1.88 0.31 0.43 -0.54 -0.52]
0.826    0.653  [-0.48 2.08 1.86 0.29 0.42 -0.53 -0.47]
1.029    0.651  [-0.48 2.08 1.83 0.27 0.42 -0.51 -0.43]
1.233    0.650  [-0.48 2.07 1.81 0.26 0.42 -0.50 -0.40]
1.437    0.649  [-0.47 2.07 1.78 0.25 0.43 -0.49 -0.37]
1.641    0.648  [-0.47 2.07 1.76 0.25 0.43 -0.48 -0.35]
1.845    0.647  [-0.47 2.07 1.74 0.24 0.44 -0.47 -0.33]
2.049    0.646  [-0.46 2.06 1.72 0.24 0.44 -0.46 -0.31]
2.253    0.645  [-0.46 2.06 1.70 0.24 0.45 -0.45 -0.29]
2.457    0.644  [-0.46 2.05 1.68 0.23 0.45 -0.44 -0.27]
2.660    0.643  [-0.46 2.05 1.67 0.23 0.45 -0.43 -0.25]
2.864    0.642  [-0.45 2.04 1.65 0.23 0.46 -0.42 -0.24]
3.068    0.642  [-0.45 2.04 1.64 0.23 0.46 -0.41 -0.22]
3.272    0.641  [-0.45 2.03 1.62 0.23 0.47 -0.40 -0.21]
3.476    0.641  [-0.45 2.03 1.61 0.23 0.47 -0.39 -0.20]
3.680    0.640  [-0.45 2.02 1.59 0.23 0.47 -0.38 -0.18]
3.884    0.640  [-0.45 2.02 1.58 0.24 0.48 -0.37 -0.17]
4.088    0.640  [-0.44 2.01 1.57 0.24 0.48 -0.37 -0.16]
4.291    0.640  [-0.44 2.00 1.56 0.24 0.48 -0.36 -0.15]
4.495    0.640  [-0.44 2.00 1.54 0.24 0.49 -0.35 -0.14]
4.699    0.641  [-0.44 1.99 1.53 0.24 0.49 -0.34 -0.13]
4.903    0.641  [-0.44 1.99 1.52 0.25 0.49 -0.33 -0.12]
5.107    0.641  [-0.44 1.98 1.51 0.25 0.49 -0.32 -0.11]
5.311    0.642  [-0.44 1.98 1.50 0.25 0.50 -0.32 -0.10]
5.515    0.643  [-0.43 1.97 1.49 0.25 0.50 -0.31 -0.10]
5.719    0.644  [-0.43 1.97 1.48 0.25 0.50 -0.30 -0.09]
5.922    0.645  [-0.43 1.96 1.47 0.26 0.50 -0.29 -0.08]
6.126    0.646  [-0.43 1.95 1.46 0.26 0.50 -0.29 -0.07]
6.330    0.647  [-0.43 1.95 1.45 0.26 0.51 -0.28 -0.07]
6.534    0.648  [-0.43 1.94 1.44 0.27 0.51 -0.27 -0.06]
6.738    0.649  [-0.43 1.94 1.43 0.27 0.51 -0.27 -0.05]
6.942    0.651  [-0.43 1.93 1.43 0.27 0.51 -0.26 -0.05]
7.146    0.652  [-0.43 1.93 1.42 0.27 0.51 -0.25 -0.04]
7.350    0.654  [-0.42 1.92 1.41 0.28 0.51 -0.24 -0.03]
7.553    0.656  [-0.42 1.92 1.40 0.28 0.51 -0.24 -0.03]
7.757    0.658  [-0.42 1.91 1.39 0.28 0.52 -0.23 -0.02]
7.961    0.659  [-0.42 1.91 1.39 0.28 0.52 -0.23 -0.02]
8.165    0.661  [-0.42 1.90 1.38 0.29 0.52 -0.22 -0.01]
8.369    0.663  [-0.42 1.90 1.37 0.29 0.52 -0.21 -0.01]
8.573    0.665  [-0.42 1.89 1.37 0.29 0.52 -0.21 -0.00]
8.777    0.668  [-0.42 1.89 1.36 0.30 0.52 -0.20 0.00]
8.981    0.670  [-0.42 1.88 1.35 0.30 0.52 -0.19 0.01]
9.184    0.672  [-0.42 1.88 1.34 0.30 0.52 -0.19 0.01]
9.388    0.674  [-0.41 1.87 1.34 0.30 0.52 -0.18 0.02]
9.592    0.677  [-0.41 1.87 1.33 0.31 0.52 -0.18 0.02]
9.796    0.679  [-0.41 1.87 1.33 0.31 0.53 -0.17 0.03]
10.000   0.681  [-0.41 1.86 1.32 0.31 0.53 -0.17 0.03]
```

Let's now plot the MSE as a function of the alphas:

```
fig, ax = plt.subplots()
ax.plot(alphas, MSEs)
ax.set_xlabel(r'$\alpha$')
ax.set_ylabel('Mean square error');
```



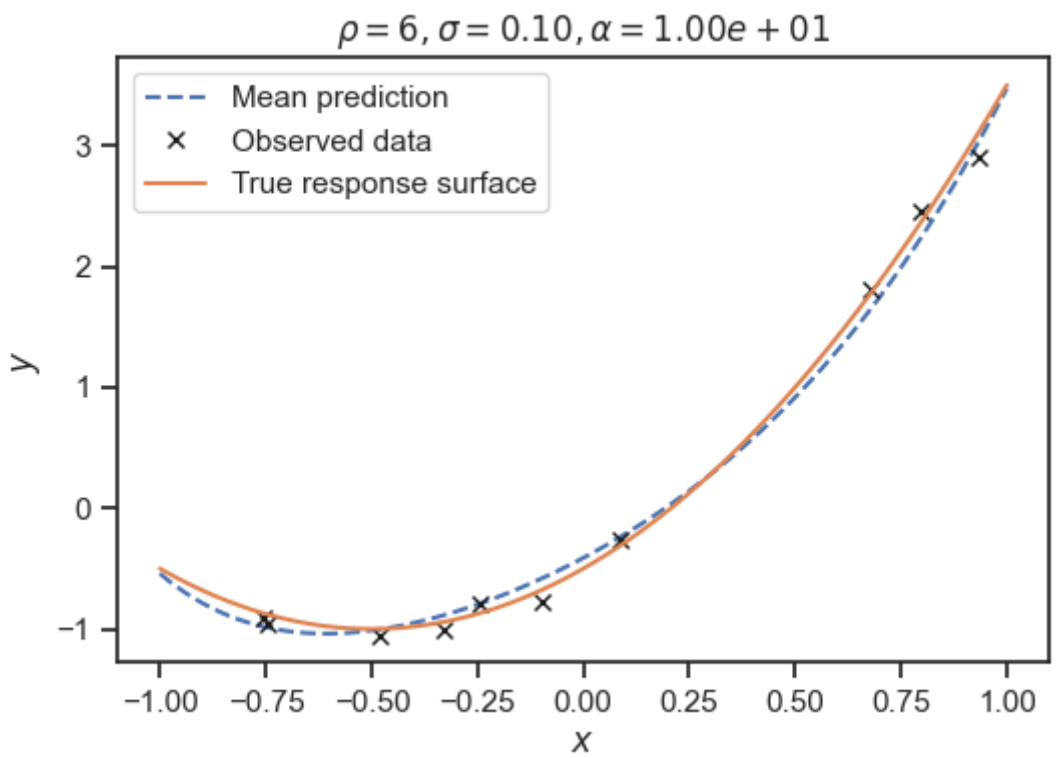
We should pick the α with the smallest MSE. Here is how:

```
min_mse_index = np.argmin(MSEs)
best_alpha = alphas[min_mse_index]
best_w = weights[min_mse_index]
print(f"Best alpha {best_alpha:.2f}")
```

```
Best alpha 4.09
```

Let's also plot the best fit:

```
xx = np.linspace(-1, 1, 100)
yy_true = w0_true + w1_true * xx + w2_true * xx ** 2
Phi_xx = get_polynomial_design_matrix(xx[:, None], degree)
yy = Phi_xx @ w

fig, ax = plt.subplots()
ax.plot(xx, yy, '--', label='Mean prediction')
ax.plot(x, y, 'kx', label='Observed data')
ax.plot(xx, yy_true, label='True response surface')
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
ax.set_title(r'$\rho={0:d}, \sigma = {1:1.2f}, \alpha={2:1.2e}$'.format(degree,
np.sqrt(sigma2), alpha))
plt.legend(loc='best');
```



Questions

- Rerun the code cell above with an α that is smaller than the optimal one. Observe how we overfit.
- Rerun the code cell above with an α that is greater than the optimal one. Observe how we underfit.

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