Bayesian Linear 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")
```

Objectives

- To demonstrate how epistemic uncertainty can be quantitied.

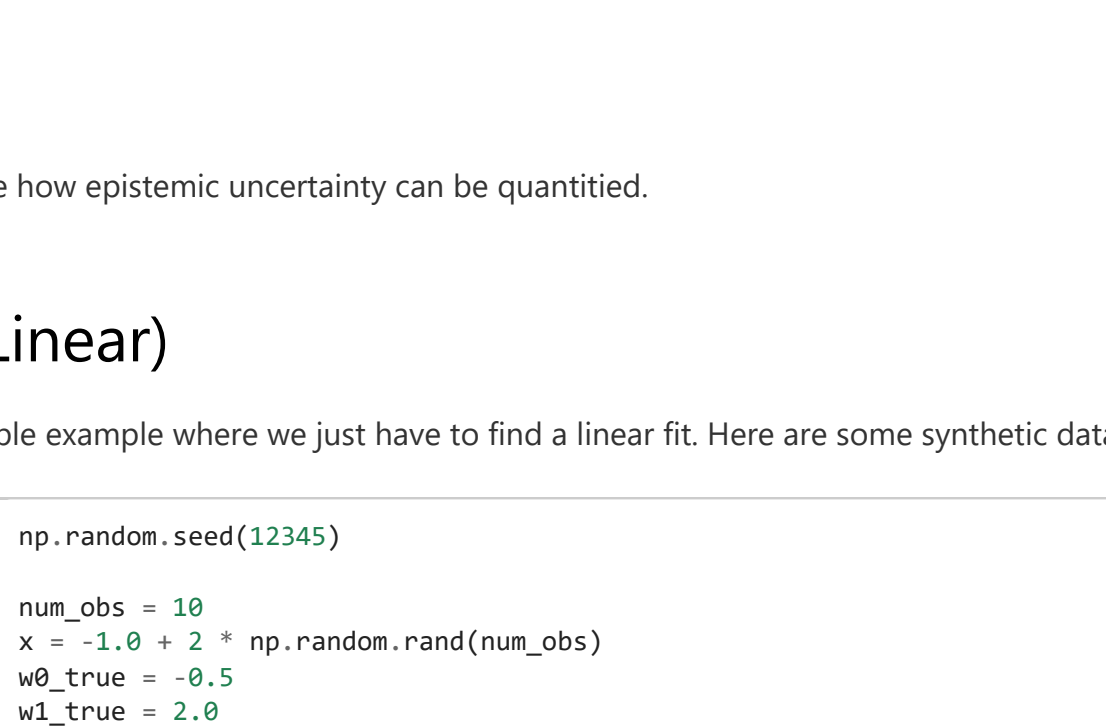
Example (Linear)

Let's start with a simple example where we just have to find a linear fit. Here are some synthetic data:

```
np.random.seed(12345)

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

We saw that for Gaussian likelihood and weightp prior, the posterior of the weights is Gaussian: $$p(\mathbf{w}|\mathbf{x}_{1:n}, \mathbf{y}_{1:n}, \sigma, \alpha) = \mathcal{N}\left(\mathbf{w}|\mathbf{m}, \mathbf{S}\right), where \mathbf{S} = \left(\sigma^{-2}\mathbf{\Phi}^T\mathbf{\Phi} + \alpha\mathbf{I}\right)^{-1}, and \mathbf{m} = \sigma^{-2}\mathbf{S}\Phi^T\mathbf{y}_{1:n}. Let's write some code that finds the posterior weight mean vector \mathbf{m} and the posterior weight covariance matrix \mathbf{S}$$:

```
import scipy

def find_m_and_S(Phi, y, sigma2, alpha):
    """Return the posterior mean and covariance of the weights
    of a Bayesian linear regression problem.

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

Let's apply this to the synthetic dataset.

```
import scipy.stats as st

degree = 1
Phi = get_polynomial_design_matrix(x[:, None], degree)

# We need to pick variance by hand for now (here I am using the true one)
sigma2 = 0.1 ** 2
# We also need to pick the regularization parameter by hand:
alpha = 0.1

# Here is the prior for the weights as a distribution:
w_prior = st.multivariate_normal(
    mean=np.zeros(degree+1),
    cov=np.eye(degree+1) / alpha
)

# Get the posterior mean and covariance for the weights:
m, S = find_m_and_S(Phi, y, sigma2, alpha)
# The posterior of the weights as a distribution:
w_post = st.multivariate_normal(mean=m, cov=S)

print(f"Posterior mean w: {m}")
print("Posterior covariance w:")
print(S)
```

Plotting these below

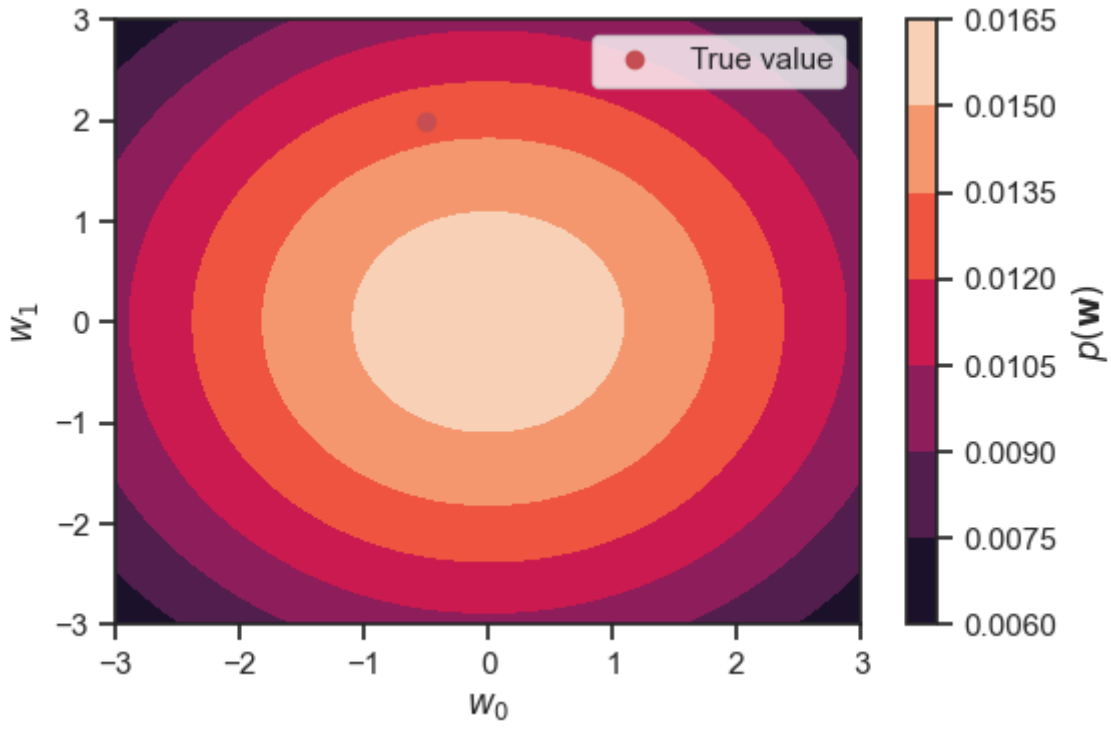
```
Posterior mean w: [-0.431  1.98 ]
Posterior covariance w:
[[ 0.001 -0.001]
 [-0.001  0.004]]
```

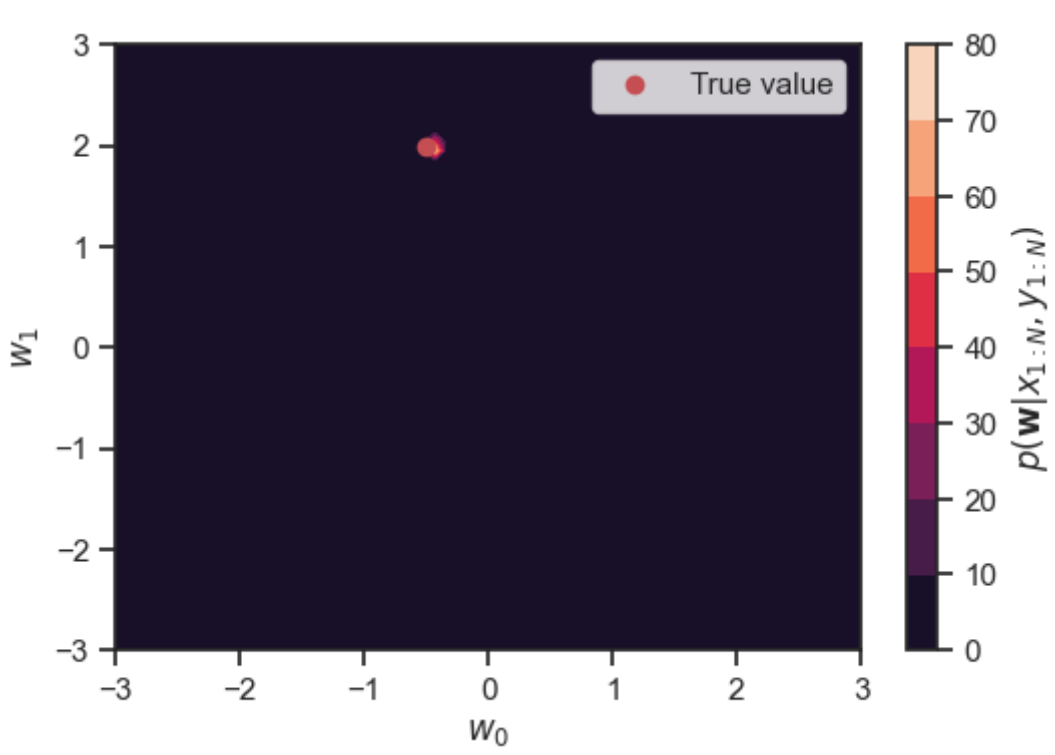
Now let's plot contours of the prior and the posterior.

```
# Grid on w
ws = np.linspace(-3.0, 3.0, 64)
W1, W2 = np.meshgrid(ws, ws)
w_all = np.hstack(
    [
        W1.flatten()[:, None],
        W2.flatten()[:, None]
    ]
)

# Prior contour
W_prior_pdf = w_prior.pdf(w_all).reshape(W1.shape)
fig, ax = plt.subplots()
c = ax.contourf(W1, W2, W_prior_pdf)
ax.plot(w0_true, w1_true, 'ro', label='True value');
plt.legend(loc='best')
ax.set_xlabel('$w_0$')
ax.set_ylabel('$w_1$')
plt.colorbar(c, label='$p(\mathbf{w})$');

# Postrior contour
W_post_pdf = w_post.pdf(w_all).reshape(W1.shape)
fig, ax = plt.subplots()
c = ax.contourf(W1, W2, W_post_pdf)
ax.plot(w0_true, w1_true, 'ro', label='True value');
plt.legend(loc='best')
ax.set_xlabel('$w_0$')
ax.set_ylabel('$w_1$')
plt.colorbar(c, label='$p(\mathbf{w}|x_{1:N}, y_{1:N})$');
```





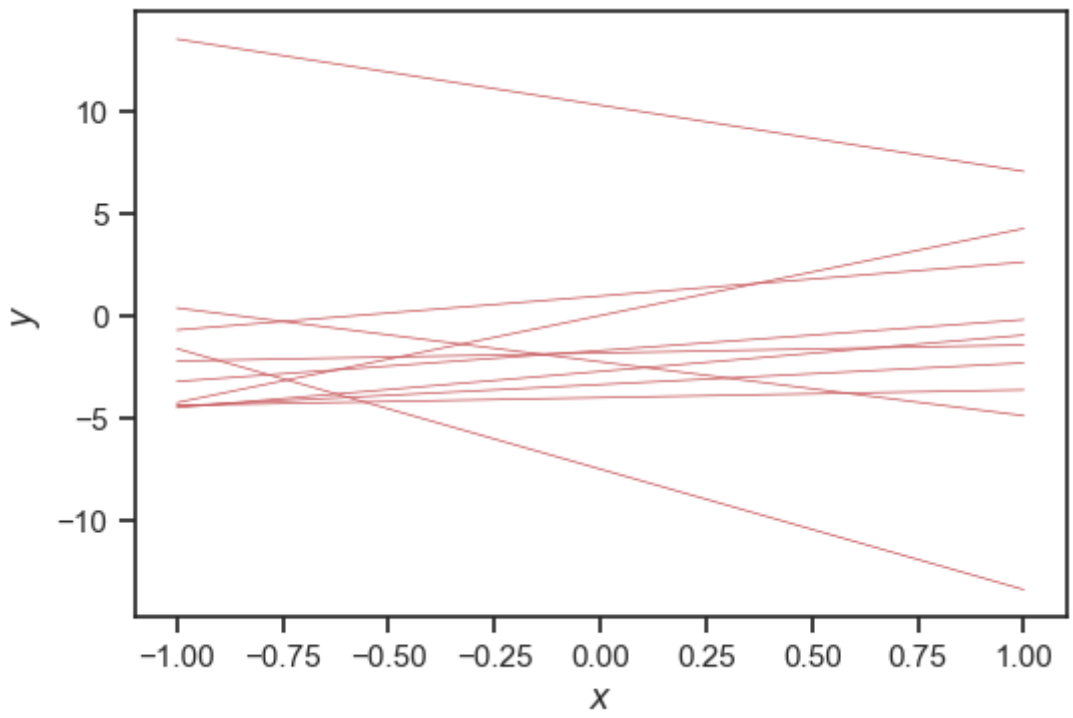
We see that in this case the posterior collapses to the true value of the weights.

Let's visualize some prior samples of the weights. What we do below is just sampling from the prior of the weights and drawing the line that corresponds to each sample. Notice that the lines are all over the place. Some have positive slope. Some are negative slope. Some cut the x-axis and some don't. That's fine. We have so much uncertainty because we haven't seen the data yet.

```
xx = np.linspace(-1, 1, 100)
Phi_xx = get_polynomial_design_matrix(xx[:, None], degree)

fig, ax = plt.subplots()
for _ in range(10):
    # Here is how you can sample from the prior
    w_sample = w_prior.rvs()
    yy_sample = Phi_xx @ w_sample
    ax.plot(xx, yy_sample, 'r', lw=0.5)

ax.set_xlabel('$x$')
ax.set_ylabel('$y$');
```



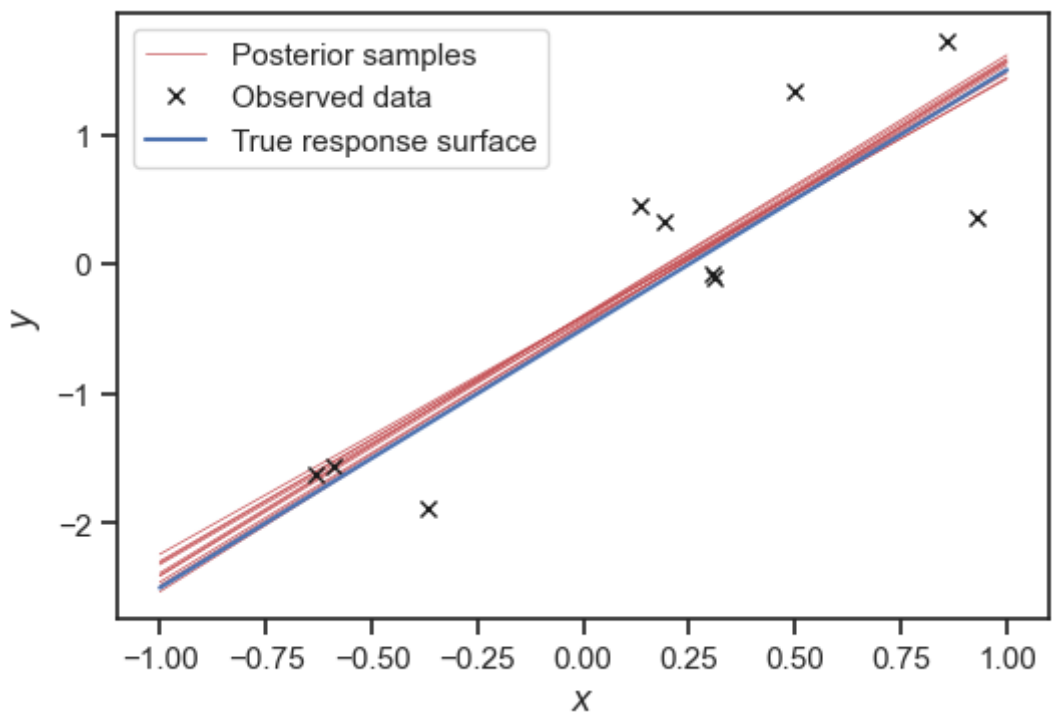
Now let's do the same thing, but instead of sampling the weights from the prior, let's sample them from the posterior. Notice that we there is much less (epistemic) uncertainty now that we are taking the data into account.

```
yy_true = w0_true + w1_true * xx

fig, ax = plt.subplots()

for _ in range(10):
    # Here is how you can sample from the posterior
    w_sample = w_post.rvs()
    yy_sample = np.dot(Phi_xx, w_sample)
    ax.plot(xx, yy_sample, 'r', lw=0.5)

ax.plot([], [], 'r', lw=0.5, label="Posterior samples")
ax.plot(x, y, 'kx', label='Observed data')
ax.plot(xx, yy_true, label='True response surface')
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
plt.legend(loc="best");
```



Questions

- Rerun the code cells above with a very small α. What happens?
- Rerun he code cells above with a very big α. What happens?
- Fix α to 5 and rerun the code cells above with a very small and the very big value for σ. What happens in each case?

Example (Quadratic)

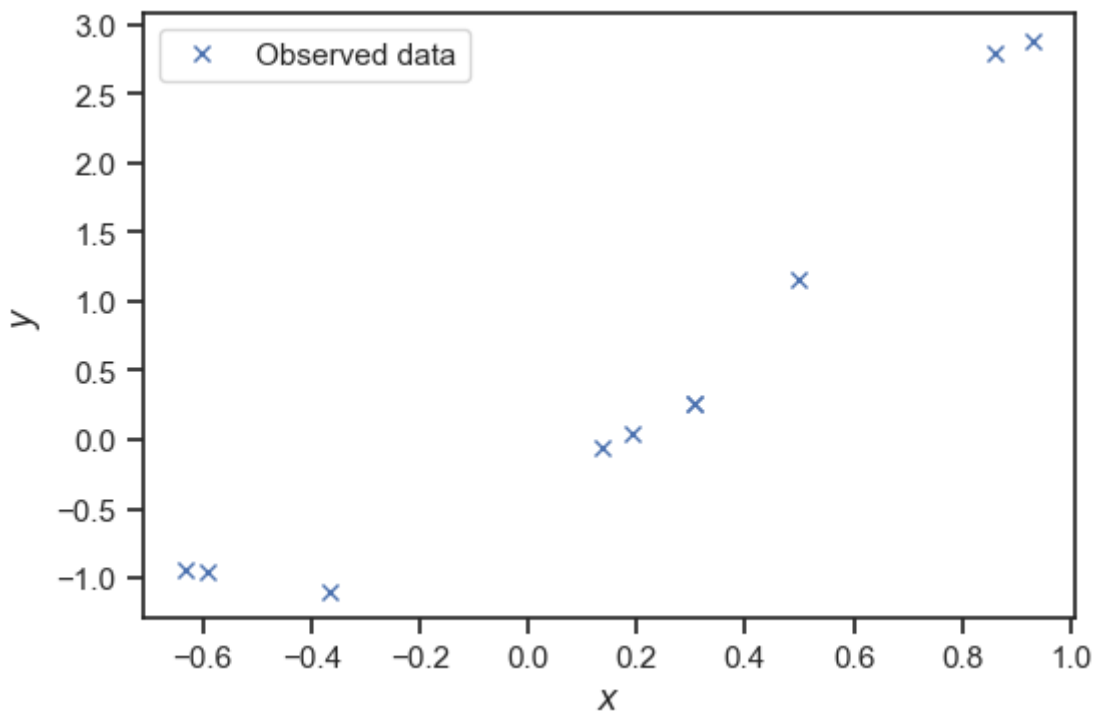
Let's repeat what we did above with a quadratic example. Here are some synthetic data:

```
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');
```



Here is how we fit a 7 degree polynomial:

```
# Parameters
degree = 7
sigma2 = 0.1 ** 2
alpha = 5.0

# Weight prior
w_prior = st.multivariate_normal(
    mean=np.zeros(degree+1),
    cov=np.eye(degree+1) / alpha
)

# Design matrix
Phi = get_polynomial_design_matrix(
    x[:, None],
    degree
)

# Fit
m, S = find_m_and_S(Phi, y, sigma2, alpha)

# Weight posterior
w_post = st.multivariate_normal(mean=m, cov=S)

print(f"Posterior mean w: {m}")
print("Posterior covariance w:")
print(S)
```

```
print(S)
```

```
Posterior mean w: [-0.364  1.873  1.329  0.361  0.538 -0.086  0.02  -0.271]
Posterior covariance w:
[[ 0.003 -0.003 -0.01   0.005  0.001  0.003  0.002  0.001]
 [-0.003  0.02   0.01  -0.036  0.007 -0.007  0.005  0.001]
 [-0.01   0.01   0.074 -0.007 -0.056 -0.004 -0.023 -0.001]
 [ 0.005 -0.036 -0.007  0.131 -0.017 -0.045 -0.018 -0.028]
 [ 0.001  0.007 -0.056 -0.017  0.151 -0.025 -0.037 -0.025]
 [ 0.003 -0.007 -0.004 -0.045 -0.025  0.157 -0.031 -0.037]
 [ 0.002  0.005 -0.023 -0.018 -0.037 -0.031  0.161 -0.035]
 [ 0.001  0.001 -0.001 -0.028 -0.025 -0.037 -0.035  0.16 ]]
```

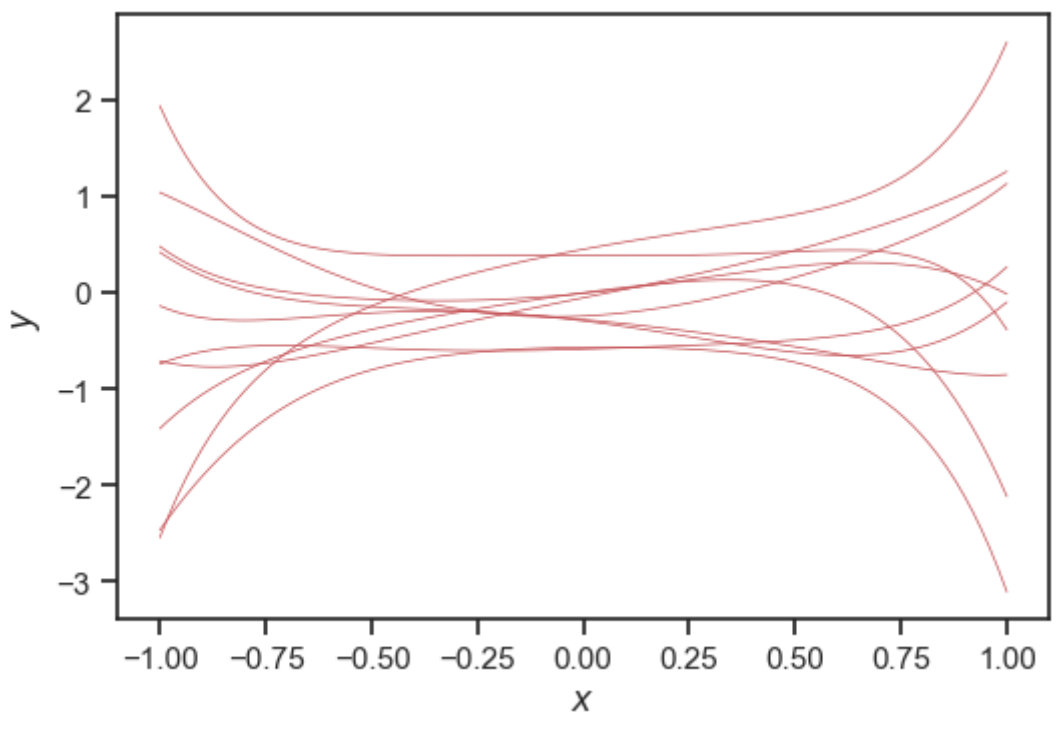
Let's visualize some samples from the prior:

```
xx = np.linspace(-1, 1, 100)
Phi_xx = get_polynomial_design_matrix(xx[:, None], degree)

fig, ax = plt.subplots()

for _ in range(10):
    w_sample = w_prior.rvs()
    yy_sample = Phi_xx @ w_sample
    ax.plot(xx, yy_sample, 'r', lw=0.5)

ax.set_xlabel('$x$')
ax.set_ylabel('$y$');
```



Let's visualize some samples from the posterior:

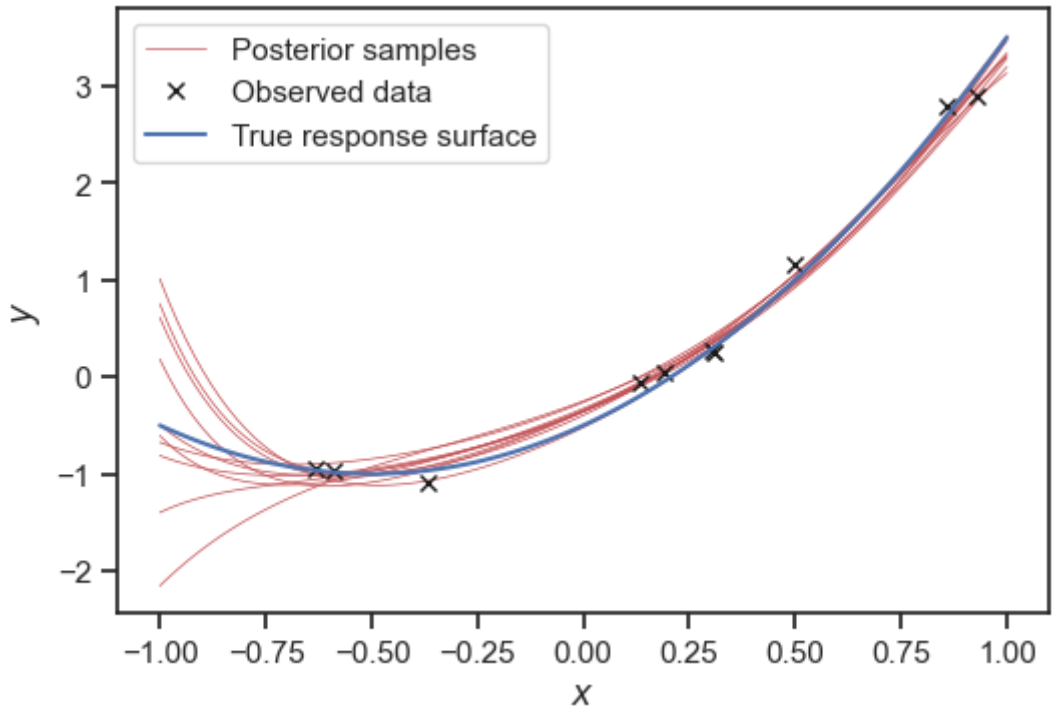
```
yy_true = w0_true + w1_true * xx + w2_true * xx ** 2

fig, ax = plt.subplots()

for _ in range(10):
    w_sample = w_post.rvs()
    yy_sample = np.dot(Phi_xx, w_sample)
    ax.plot(xx, yy_sample, 'r', lw=0.5)

ax.plot([], [], 'r', lw=0.5, label="Posterior samples")
ax.plot(x, y, 'kx', label='Observed data')
ax.plot(xx, yy_true, label='True response surface')
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
plt.legend(loc="best")
```

```
<matplotlib.legend.Legend at 0x287d4a1f0>
```



Questions

- Rerun the code cells above with a very small α. What happens?
- Rerun he code cells above with a very big α. What happens?
- Fix α to 5 and rerun the code cells above with a very small and the very big value for σ. What happens in each case?

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