

Automatic Relevance Determination

Contents

- Objectives
- Example (Quadratic)

```
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 automatic relevance determination can be used to select which features to keep.

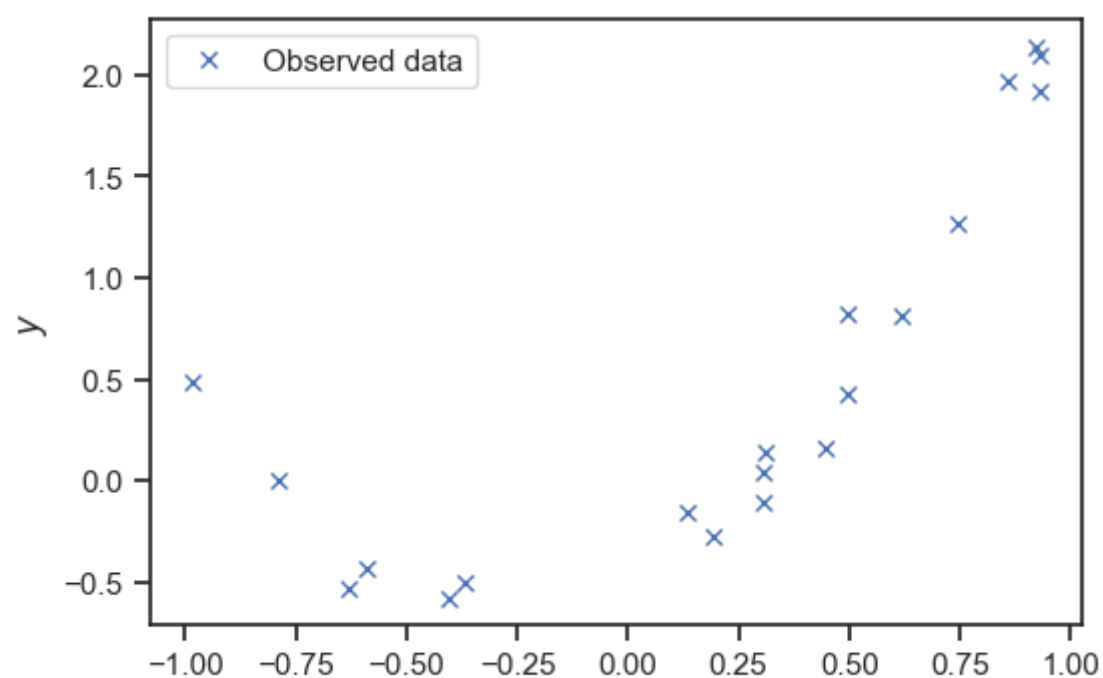
Example (Quadratic)

As before, let's start with our familiar synthetic example:

```
np.random.seed(12345)

num_obs = 20
x = -1.0 + 2 * np.random.rand(num_obs)
w0_true = -0.5
w1_true = 1.0
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 we have developped 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)

def plot_posterior_predictive(
    model,
    xx,
    phi_func,
    phi_func_args=(),
    y_true=None
):
    """Plot the posterior predictive separating
    aleatory and epistemic uncertainty.

    Arguments:
    model      -- A trained model.
    xx        -- The points on which to evaluate
                 the posterior predictive.
    phi_func   -- The function to use to compute
                 the design matrix.

    Keyword Arguments:
    phi_func_args -- Any arguments passed to the
                     function that calculates the
                     design matrix.
    y_true       -- The true response for plotting.
    """
    Phi_xx = phi_func(
        xx[:, None],
        *phi_func_args
    )
    yy_mean, yy_measured_std = model.predict(
        Phi_xx,
        return_std=True
    )

```

```

)

sigma = np.sqrt(1.0 / model.alpha_)
yy_std = np.sqrt(yy_measured_std ** 2 - sigma**2)
yy_le = yy_mean - 2.0 * yy_std
yy_ue = yy_mean + 2.0 * yy_std
yy_lae = yy_mean - 2.0 * yy_measured_std
yy_uae = yy_mean + 2.0 * yy_measured_std

fig, ax = plt.subplots()
ax.plot(xx, yy_mean, 'r', label="Posterior mean")
ax.fill_between(
    xx,
    yy_le,
    yy_ue,
    color='red',
    alpha=0.25,
    label="95% epistemic credible interval"
)
ax.fill_between(
    xx,
    yy_lae,
    yy_ue,
    color='green',
    alpha=0.25
)
ax.fill_between(
    xx,
    yy_ue,
    yy_uae,
    color='green',
    alpha=0.25,
    label="95% epistemic + aleatory credible interval"
)
ax.plot(x, y, 'kx', label='Observed data')
if y_true is not None:
    ax.plot(xx, y_true, "--", label="True response")
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
plt.legend(loc="best");

def plot_posterior_samples(
    model,
    xx,
    phi_func,
    phi_func_args=(),
    num_samples=10,
    y_true=None,
    nugget=1e-6
):
    """Plot posterior samples from the model.

    Arguments:
    model      -- A trained model.
    xx         -- The points on which to evaluate
                  the posterior predictive.
    phi_func   -- The function to use to compute
                  the design matrix.

    Keyword Arguments:
    phi_func_args -- Any arguments passed to the
                     function that calculates the
                     design matrix.
    num_samples  -- The number of samples to take.
    y_true       -- The true response for plotting.
    nugget       -- A small number to add the covariance
                     if it is not positive definite
                     (numerically).

    """
    Phi_xx = phi_func(
        xx[:, None],
        *phi_func_args
    )
    m = model.coef_
    S = model.sigma_
    w_post = st.multivariate_normal(
        mean=m,
        cov=S + nugget * np.eye(S.shape[0])
    )
    fig, ax = plt.subplots()
    for _ in range(num_samples):
        w_sample = w_post.rvs()
        yy_sample = 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");
```

We are not going to implement the automatic relevance determination from scratch. Instead we are going to use the implementation in [scikit-learn](#). Here is how you do it:

```
from sklearn.linear_model import ARDRegression

# Parameters
degree = 6

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

print(Phi.shape)

# Fit
model = ARDRegression(
    fit_intercept=False,
    threshold_lambda=np.inf
).fit(Phi, y)
```

```
(20, 7)
```

Here is how you can get the noise variance σ^2 :

```
sigma = np.sqrt(1.0 / model.alpha_)
print('sigma = {0:1.2f}'.format(sigma))
```

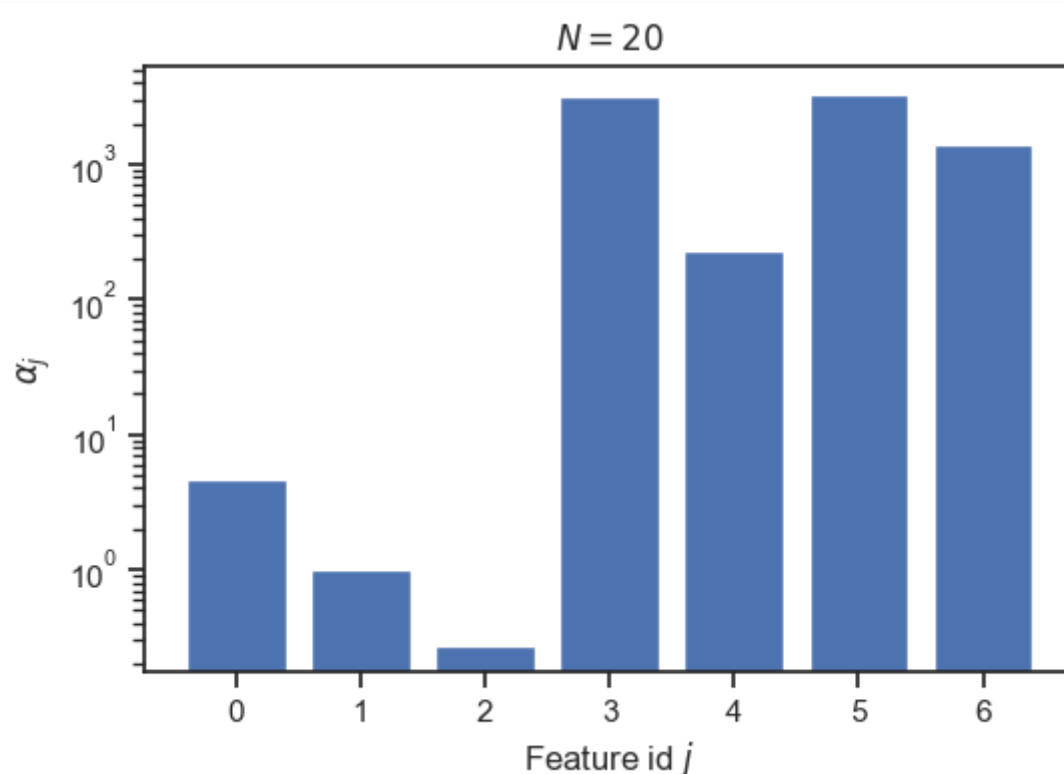
```
sigma = 0.14
```

Here is how you can get the precision corresponding to each weight. Remember, that very high precision means that the corresponding basis function can be safely removed from your model.

```
alpha = model.lambda_
print("Prior w precision: {alpha}")

fig, ax = plt.subplots()
ax.bar(np.arange(degree+1), alpha)
ax.set_xlabel('Feature id $j$')
ax.set_ylabel(r'$\alpha_j$')
ax.set_yscale("log")
ax.set_title(f'$N={num\_obs}$');
```

```
Prior w precision: {alpha}
```



Remember that the higher the prior precision α_j of a weight, the more its prior (and consequently the posterior) concentrates about zero. For extremely high values of prior precision, the corresponding weight is essentially zero. We see that ARD tells us that we don't have to include the 3rd, 4th, 5th, and 6th degree monomials.

Now, let's get the postserior mean of the weights. Notice that the weights corresponding to 3rd, 4th, 5th, and 6th degree polynomials are very close to zero:

```
# The posterior mean of the weights is here (this is for the normalized features, however)
m = model.coef_
print(f"Posterior mean w: {m}")
```

```
Posterior mean w: [-4.603e-01  9.906e-01  1.893e+00 -7.907e-04  9.780e-03 -1.195e-03
 1.786e-03]
```

And here is the posterior covariance of the weights:

```
S = model.sigma_
print("Posterior covariance w:")
print(S)
```

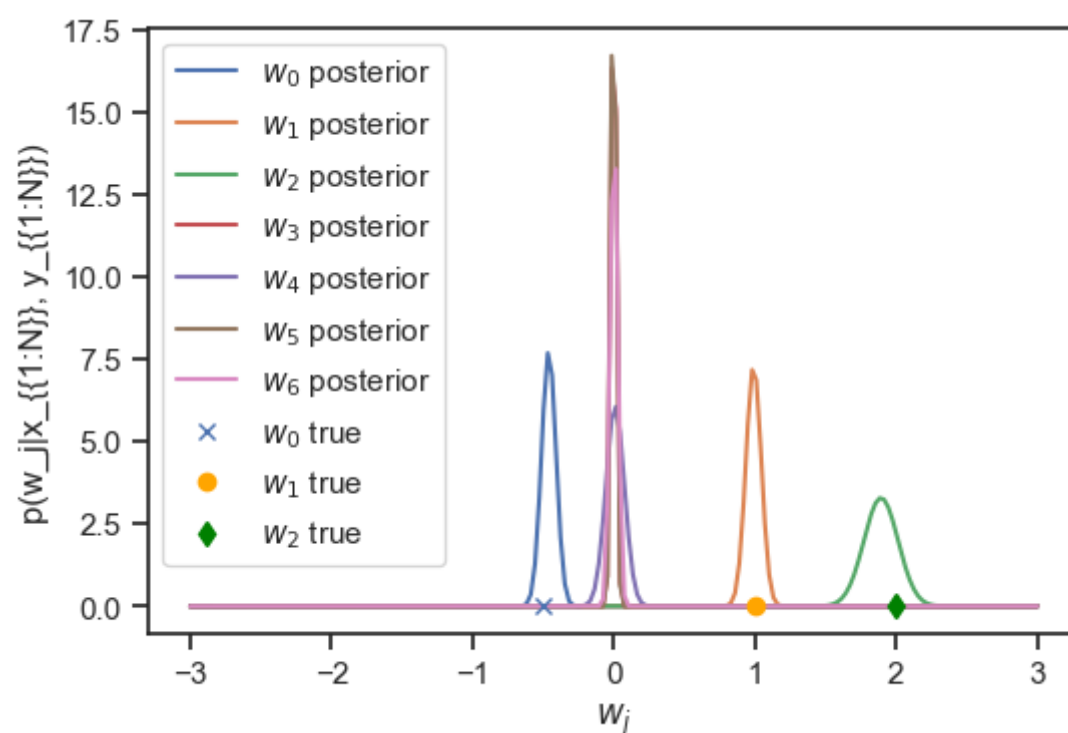
```
Posterior covariance w:
[[ 2.645e-03 -2.656e-04 -4.507e-03  2.806e-05  5.301e-04  2.347e-05
   9.399e-05]
 [-2.656e-04  2.974e-03 -7.205e-04 -1.930e-04 -1.085e-05 -1.423e-04
   4.510e-06]
 [-4.507e-03 -7.205e-04  1.469e-02 -5.819e-05 -4.080e-03 -4.802e-05
  -5.663e-04]
 [ 2.806e-05 -1.930e-04 -5.819e-05  3.069e-04  2.892e-06 -2.948e-06
   8.687e-07]
 [ 5.301e-04 -1.085e-05 -4.080e-03  2.892e-06  4.314e-03  3.217e-06
  -2.259e-05]
 [ 2.347e-05 -1.423e-04 -4.802e-05 -2.948e-06  3.217e-06  2.996e-04
   1.003e-06]
 [ 9.399e-05  4.510e-06 -5.663e-04  8.687e-07 -2.259e-05  1.003e-06
   7.027e-04]]
```

Now, let's plot the marginal posterior for each weight.

```
import scipy.stats as st

ww = np.linspace(-3.0, 3.0, 200)

fig, ax = plt.subplots()
for j in range(S.shape[0]):
    wj_post = st.norm(
        loc=m[j],
        scale=np.sqrt(S[j, j])
    )
    ax.plot(
        ww,
        wj_post.pdf(ww),
        label=f'$w_{{{j}}}$ posterior')
ax.plot([w0_true], [0], 'bx', label="$w_0$ true")
ax.plot([w1_true], [0], 'o', color="orange", label="$w_1$ true")
ax.plot([w2_true], [0], 'd', color="green", label="$w_2$ true")
ax.set_xlabel("$w_j$")
ax.set_ylabel("p(w_j|x_{{1:N}}, y_{{1:N}})")
plt.legend(loc='best');
```

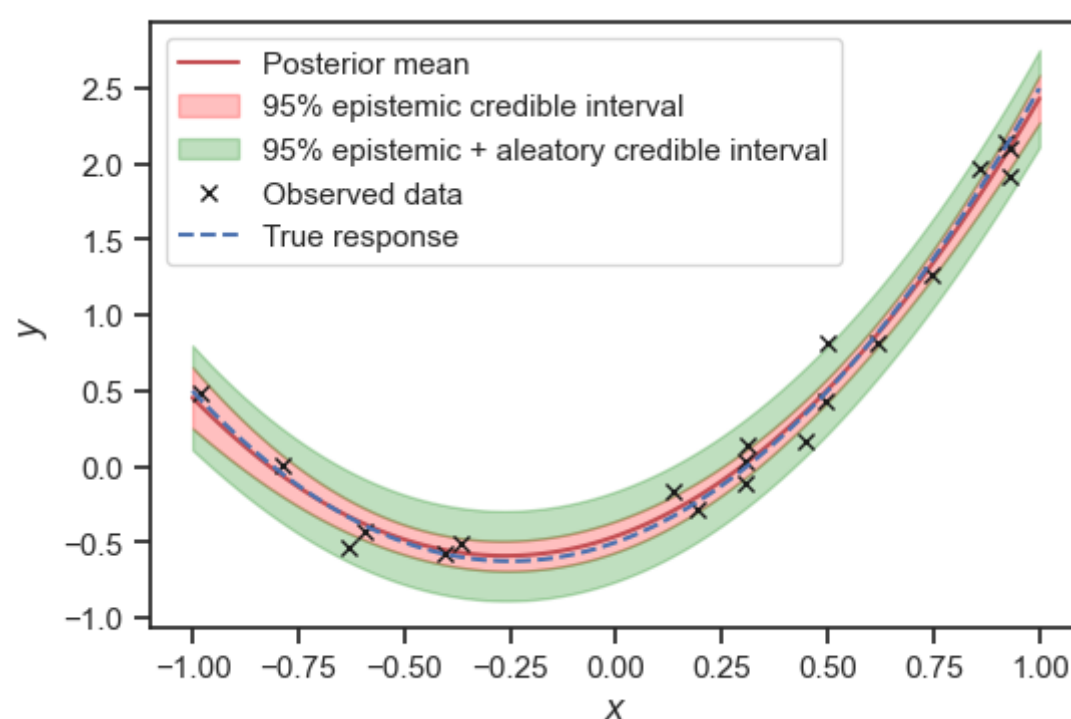


Notice how 3rd, 4th, 5th, and 6th are centered at zero. Also, notice how the true weights have been correctly identified.

Let's now plot the posterior predictive separating aleatory and epistemic uncertainty:

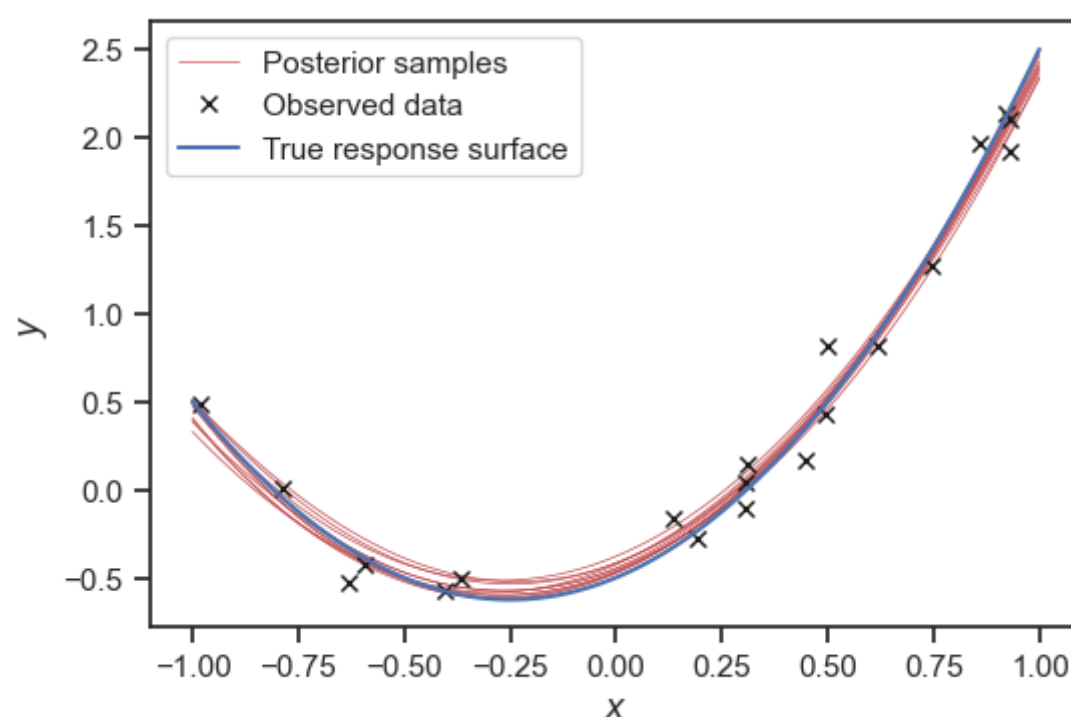
```
xx = np.linspace(-1, 1, 100)
yy_true = w0_true + w1_true * xx + w2_true * xx ** 2

plot_posterior_predictive(
    model,
    xx,
    get_polynomial_design_matrix,
    phi_func_args=(degree,),
    y_true=yy_true
)
```



Let's take samples from the posterior:

```
plot_posterior_samples(
    model,
    xx,
    get_polynomial_design_matrix,
    phi_func_args=(degree,),
    y_true=yy_true
)
```



Questions

- Rerun with a smaller number of observations, say $N = 5$. What happens to the epistemic uncertainty?
- Rerun with a very small number of observations, say $N = 2$. What happens then? (The step that samples from the posterior may not work as expected. Please look at the comment in the code to fix the problem.)
- Rerun everything with a higher degree polynomial. Try 4, 8, and 16. Notice that the fit remains good in between but the way you extrapolate changes. Why?

By Ilias Bilonis (ibilion[at]purdue.edu)
© Copyright 2021.