Gaussian_process_regression_pymc

November 7, 2019

1 Gaussian Process Regression

At times you don't care about the underlying model for your data points and just want a model that describes the data. One such fitting technique is know as Gaussian process regression (also know as kriging). This kind of regression assumes all the data points are drawn from a common covariance function. This function is used to generate an (infinite) set of functions and only keeps the ones that pass through the observed data.

1.1 Packages being used

- `pymc3`: has a Gaussian process regression function

1.2 Relevant documentation

- `pymc3`: https://docs.pymc.io/notebooks/GP-MeansAndCovs.html, https://docs.pymc.io/notebooks/GP-Marginal.html

[9]:
```
import numpy as np
import pymc3 as pm
import theano.tensor as tt
from scipy import interpolate
from matplotlib import pyplot as plt
import mpl_style
%matplotlib inline
plt.style.use(mpl_style.style1)
```

[2]:
```
import warnings
warnings.simplefilter('ignore')
```

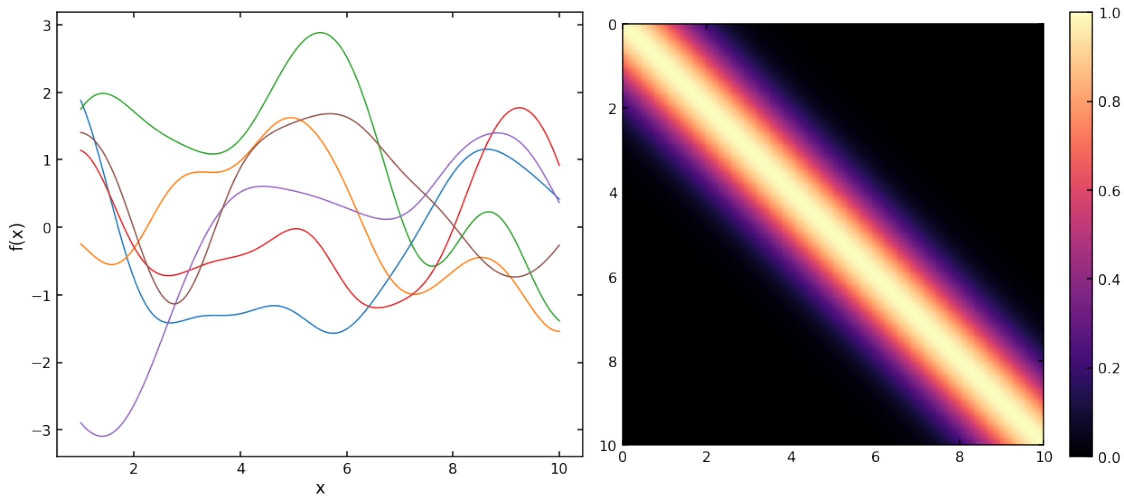
1.3 The squared exponential covariance (or Radial-basis function or Exponential Quadratic)

As an example we will use the squared exponential covariance function:

$$\text{Cov}\left(x_1, x_2; h\right) = \exp\left(\frac{-(x_1 - x_2)^2}{2h^2}\right)$$

Lets using this function to draw some *unconstrained* functions:

[10]:
```
h = 1
cov = pm.gp.cov.ExpQuad(1, h)

x = np.linspace(1, 10, 500)[:, None]
K = cov(x).eval()

plt.figure(1, figsize=(18, 8))

plt.subplot(121)
plt.plot(x, pm.MvNormal.dist(mu=np.zeros(K.shape[0]), cov=K).random(size=6).T)
plt.xlabel('x')
plt.ylabel('f(x)')

plt.subplot(122)
plt.imshow(K, interpolation='none', origin='upper', extent=[0, 10, 10, 0])
plt.colorbar()
plt.tight_layout();
```



1.4 Constrain the model

Assume we have some data points, we can use Gaussian process regression to only pick the models that pass through those points:

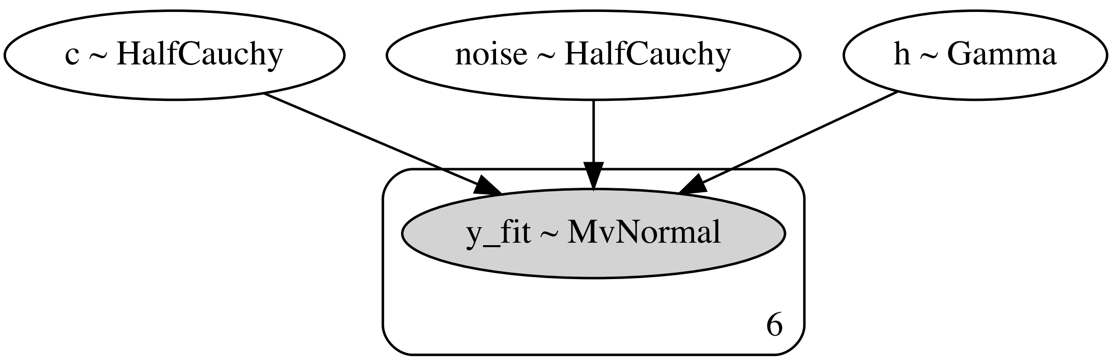
[4]:
```
x1 = np.array([1, 3, 5, 6, 7, 8])
y1 = x1 * np.sin(x1)
```

1.4.1 Build the pymc model

We will define priors for the lenght scale `h` and the leading scaling coefficient `c`. We will assume there is a small level of unknown noise associated with each data point.

[5]:
```python
X = x1[:, None]
with pm.Model() as model:
    h = pm.Gamma("h", alpha=2, beta=1)
    c = pm.HalfCauchy("c", beta=5)
    cov = c**2 * pm.gp.cov.ExpQuad(1, ls=h)
    gp = pm.gp.Marginal(cov_func=cov)
    noise = pm.HalfCauchy("noise", beta=5)
    y_fit = gp.marginal_likelihood("y_fit", X=X, y=y1, noise=noise)

display(model)
display(pm.model_to_graphviz(model))
```

$$
\begin{array}{rcl}
\text{h} & \sim & \text{Gamma}(alpha = 2.0,\, beta = 1.0) \\
\text{c} & \sim & \text{HalfCauchy}(beta = 5.0) \\
\text{noise} & \sim & \text{HalfCauchy}(beta = 5.0) \\
y_f it & \sim & \text{MvNormal}(mu = f(), cov = f(f(f(f(f(),\, f(f(f(f(),\, f(f(array,\, f(f(\text{h}))),\, f(f(array,\, f(f(\text{h}))))))),
\end{array}
$$



Find the maximum of the likelihood using the `find_MAP` function.

[6]:
```python
with model:
    mp = pm.find_MAP()

print()
print('Best fit kernel: {0:.2f}**2 * ExpQuad(ls={1:.2f})'.format(mp['c'],␣
↪mp['h']))
```

```
logp = -20.36, ||grad|| = 3.2706e-05: 100%|| 32/32 [00:00<00:00,

Best fit kernel: 4.16**2 * ExpQuad(ls=1.58)
```

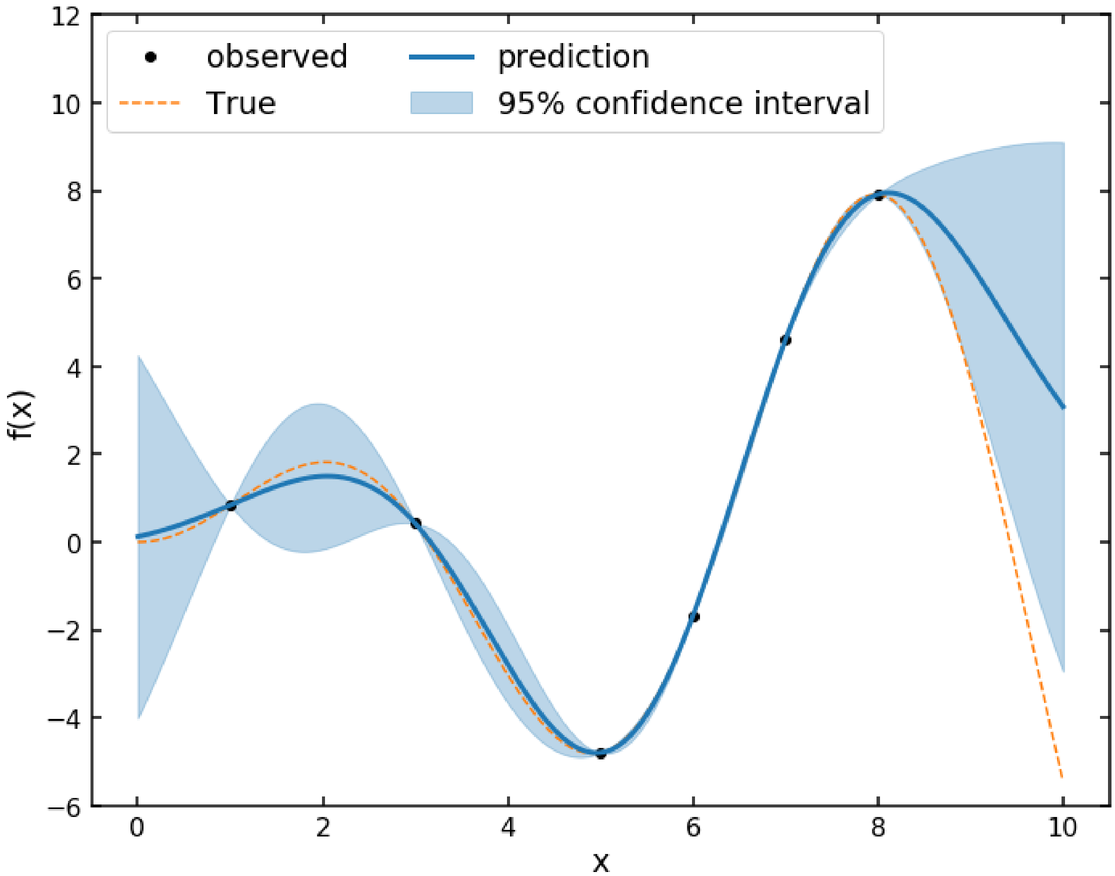
1.4.2 Use the fit to interpolate to new X values

This MAP fit can be used to interpolate and extrapolate to a new grid of points. PYMC offers the predict method to make this easier.

[7]:
```
n_new = 500
X_new = np.linspace(0, 10, n_new)

mu, var = gp.predict(X_new[:,None], point=mp, diag=True)
sd = np.sqrt(var)
```

Let's plot the result:

[11]:
```
plt.figure(2, figsize=(10, 8))
plt.plot(x1, y1, 'ok', label='observed')

plt.plot(
    X_new,
    X_new * np.sin(X_new),
    '--',
    color='C1',
    label='True'
)

plt.plot(
    X_new.flatten(),
    mu,
    color='C0',
    lw=3,
    zorder=3,
    label='prediction'
)

# plot 95% best fit region
plt.fill_between(
    X_new.flatten(),
    mu - 1.96*sd,
    mu + 1.96*sd,
    color='C0',
    alpha=0.3,
    zorder=1,
    label='95% confidence interval'
)

# labels and legend
plt.xlabel('x')
plt.ylabel('f(x)')
plt.ylim(-6, 12)
plt.legend(loc='upper left', ncol=2)
plt.tight_layout();
```



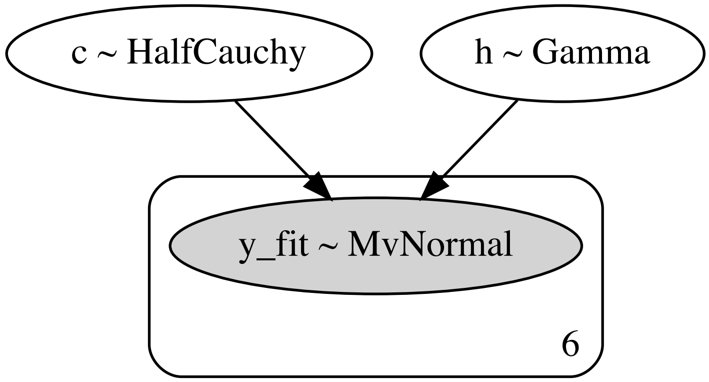
1.5 Noisy data

Let's add some noise to the data. We will assume each data point has indipendend errorbars. These values can be passed directly into the `marginal_lieklihood` function instead of the prior we were using before.

[12]:
```
dy = 0.5 + np.random.random(y1.shape)
y_noise = np.random.normal(0, dy)
y2 = y1 + y_noise
```

[13]:
```
with pm.Model() as model_noise:
    h = pm.Gamma("h", alpha=2, beta=1)
    c = pm.HalfCauchy("c", beta=5)
    cov = c**2 * pm.gp.cov.ExpQuad(1, ls=h)
    gp = pm.gp.Marginal(cov_func=cov)
    y_fit = gp.marginal_likelihood("y_fit", X=X, y=y2, noise=dy)

display(model_noise)
display(pm.model_to_graphviz(model_noise))
```

$$
\begin{array}{rcl}
\text{h} & \sim & \text{Gamma}(\mathit{alpha} = 2.0,\ \mathit{beta} = 1.0) \\
\text{c} & \sim & \text{HalfCauchy}(\mathit{beta} = 5.0) \\
y_f it & \sim & \text{MvNormal}(\mathit{mu} = f(), \mathit{cov} = f(f(f(f(f(), f(f(f(f(), f(f(\mathit{array}, f(f(\text{h}))), f(f(\mathit{array}, f(f(\text{h})))))), f
\end{array}
$$



[14]:
```
with model_noise:
    mp_noise = pm.find_MAP()

print()
print('Best fit kernel: {0:.2f}**2 * ExpQuad(ls={1:.2f})'.format(mp_noise['c'],␣
 ↪mp_noise['h']))
```

```
logp = -24.089, ||grad|| = 6.1563: 100%|| 10/10 [00:00<00:00,
```

```
Best fit kernel: 4.14**2 * ExpQuad(ls=0.91)
```

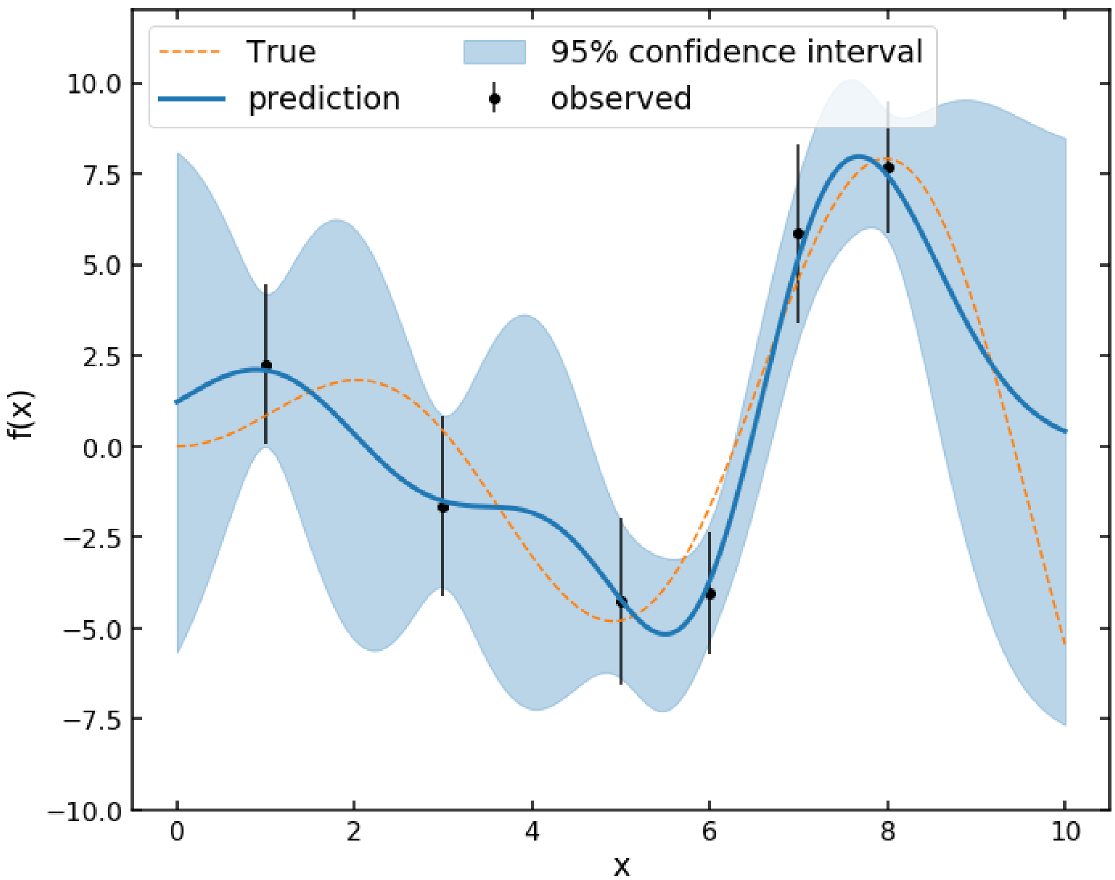
1.5.1 Plot the results

As before we can interpolate and extrapolate to new points.

[15]:
```
mu_noise, var_noise = gp.predict(X_new[:,None], point=mp_noise, diag=True)
sd_noise = np.sqrt(var_noise)
```

[16]:
```
plt.figure(3, figsize=(10, 8))
plt.errorbar(x1, y2, yerr=1.96*dy, fmt='ok', label='observed')

plt.plot(
    X_new,
    X_new * np.sin(X_new),
    '--',
    color='C1',
```

```
    label='True'
)

plt.plot(
    X_new.flatten(),
    mu_noise,
    color='C0',
    lw=3,
    zorder=3,
    label='prediction'
)

# plot 95% best fit region
plt.fill_between(
    X_new.flatten(),
    mu_noise - 1.96*sd_noise,
    mu_noise + 1.96*sd_noise,
    color='C0',
    alpha=0.3,
    zorder=1,
    label='95% confidence interval'
)

# labels and legend
plt.xlabel('x')
plt.ylabel('f(x)')
plt.ylim(-10, 12)
plt.legend(loc='upper left', ncol=2)
plt.tight_layout();
```



1.6 A Cubic Spline

So far we have been using the ExpQuad kernel, but it there are others that can be used. You may have noticed that this method of fitting provides smooth curves that pass thorugh the data points very similar to how a spline fit does. As it turns out, a spline fit is just a special case of a gaussian process fit. To recreate a cubic spline we can use the following kernel:

$$\mathrm{Cov}(x_1, x_2) = 1 + |x_1 - x_2| \frac{\min(x_1, x_2)^2}{2} + \frac{\min(x_1, x_2)^3}{3}$$

Under the condition that all values of x_1 and x_2 are between the values of 0 and 1. This normalization ensure that the covariance matrix is positive definite. So unlike other kernels we will needed to know the range we plan to extrapolate onto before doing our fit.

This kernel is not built into pymc3 so we will have to write a custom kernel for it:

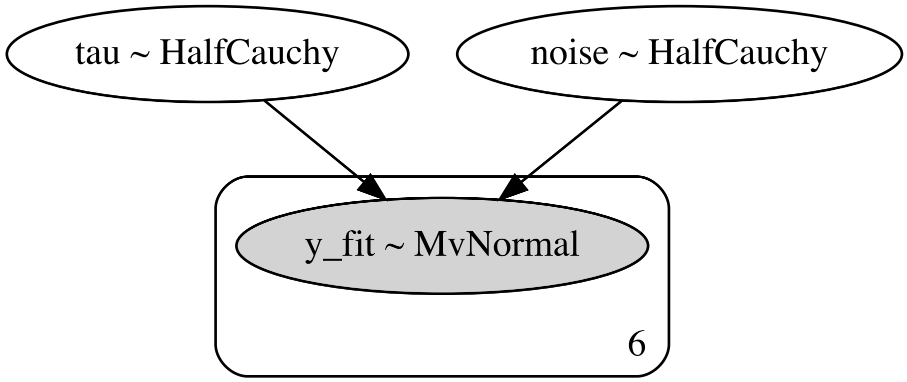
[17]:
```
class CubicSpline(pm.gp.cov.Covariance):
    def __init__(self, dim, x_min=-100, x_max=100):
        super(CubicSpline, self).__init__(1, None)
        self.x_min = x_min
        self.x_max = x_max
```

```python
    def norm(self, X):
        d = self.x_max - self.x_min
        return (X - self.x_min) / d

    def diag(self, X):
        X, _ = self._slice(X, Xs=None)
        X = self.norm(X)
        Xt = tt.flatten(X)
        return 1 + (Xt**3) / 3

    def full(self, X, Xs=None):
        X, Xs = self._slice(X, Xs)
        if Xs is None:
            Xs = X
        X = self.norm(X)
        Xs = self.norm(Xs)
        d = tt.abs_(X - tt.transpose(Xs))
        v = tt.minimum(X, tt.transpose(Xs))
        k = 1 + (0.5 * d * v**2) + ((v**3) / 3)
        return k
```

Now we can fit for this kernel's coefficient. The prior for the noise is pushed to lower values to ensrue the fit does not treat the data as "noise only."

[18]:
```python
with pm.Model() as model_poly:
    tau = pm.HalfCauchy('tau', beta=5)
    cov = CubicSpline(1, x_min=0, x_max=10) * tau**2
    gp = pm.gp.Marginal(cov_func=cov)
    noise = pm.HalfCauchy('noise', beta=0.1)
    y_fit = gp.marginal_likelihood("y_fit", X=x1[:, None], y=y1, noise=noise)

display(model_poly)
display(pm.model_to_graphviz(model_poly))
```

$$\begin{array}{rcl} \text{tau} & \sim & \text{HalfCauchy}(\mathit{beta}=5.0) \\ \text{noise} & \sim & \text{HalfCauchy}(\mathit{beta}=0.1) \\ y_fit & \sim & \text{MvNormal}(\mathit{mu}=f(), \mathit{cov}=f(f(f(f(f(), f(f(f(), f(f(f(f(\mathit{array}, f()), f()), f(f(f(\mathit{array}, f()), f \end{array}$$



[19]:
```
with model_poly:
    mp_poly = pm.find_MAP()

print()
print('{0}**2 * CubicSpline'.format(mp_poly['tau']))
```

```
logp = -27.037, ||grad|| = 0.045779: 100%|| 31/31 [00:00<00:00,
```

```
96.42041087387564**2 * CubicSpline
```

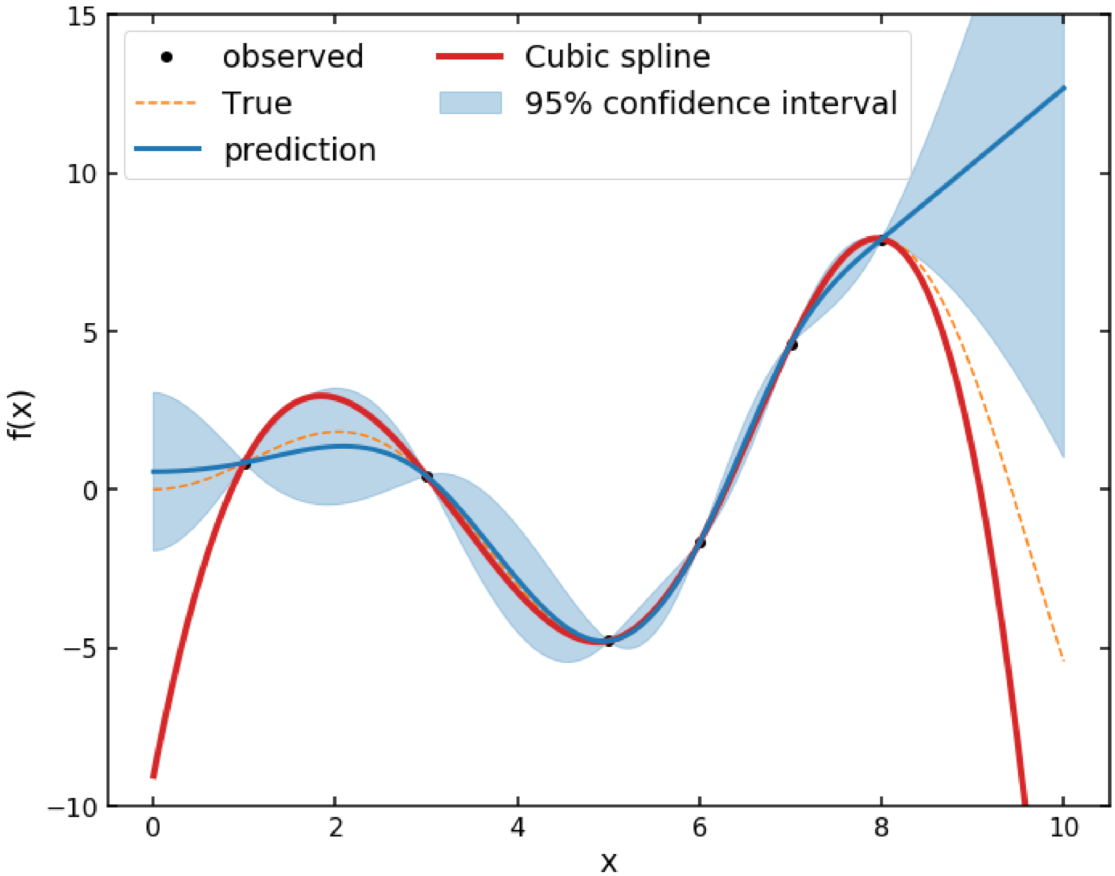
[20]:
```
mu_poly, var_poly = gp.predict(X_new[:,None], point=mp_poly, diag=True)
sd_poly = np.sqrt(var_poly)
```

We will also fit a cubic spline to the data and compare it to the gaussian process fit.

[21]:
```
tck = interpolate.splrep(x1, y1, k=3)
y_new = interpolate.splev(X_new, tck)
```

[22]:
```
plt.figure(4, figsize=(10, 8))
plt.plot(x1, y1, 'ok', label='observed')

plt.plot(
    X_new,
    X_new * np.sin(X_new),
    '--',
    color='C1',
    label='True'
)

plt.plot(
    X_new.flatten(),
    mu_poly,
```

```
    color='C0',
    lw=3,
    zorder=3,
    label='prediction'
)

# plot 95% best fit region
plt.fill_between(
    X_new.flatten(),
    mu_poly - 1.96*sd_poly,
    mu_poly + 1.96*sd_poly,
    color='C0',
    alpha=0.3,
    zorder=1,
    label='95% confidence interval'
)

plt.plot(X_new, y_new, color='C3', label='Cubic spline', lw=4)

# labels and legend
plt.xlabel('x')
plt.ylabel('f(x)')
plt.ylim(-10, 15)
plt.legend(loc='upper left', ncol=2)
plt.tight_layout();
```

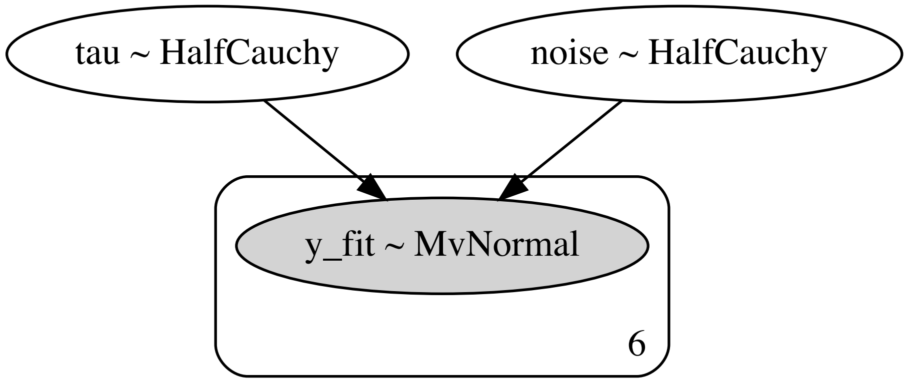


Notice that aside from the end points the gaussian process the the spline give the same result. Additionally we now have an errorbar estmate for a spline fit!

Now that we have this working we can include measurement errors on each of the data points like we did before.

[23]:
```
with pm.Model() as model_poly_err:
    tau = pm.HalfCauchy('tau', beta=5)
    cov = CubicSpline(1, x_min=0, x_max=10) * tau**2
    gp = pm.gp.Marginal(cov_func=cov)
    y_fit = gp.marginal_likelihood("y_fit", X=x1[:, None], y=y2, noise=dy)

display(model_poly)
display(pm.model_to_graphviz(model_poly))
```

$$\begin{array}{rcl} \text{tau} & \sim & \text{HalfCauchy}(\mathit{beta}=5.0) \\ \text{noise} & \sim & \text{HalfCauchy}(\mathit{beta}=0.1) \\ y_f it & \sim & \text{MvNormal}(\mathit{mu}=f(), \mathit{cov}=f(f(f(f(f(), f(f(f(), f(f(f(f(\mathit{array}, f()), f()), f(f(f(\mathit{array}, f()), f \end{array}$$



[24]:
```
with model_poly_err:
    mp_poly_err = pm.find_MAP()

print()
print('{0}**2 * CubicSpline'.format(mp_poly['tau']))
```

```
logp = -61.174, ||grad|| = 9.7592: 100%|| 10/10 [00:00<00:00,
```

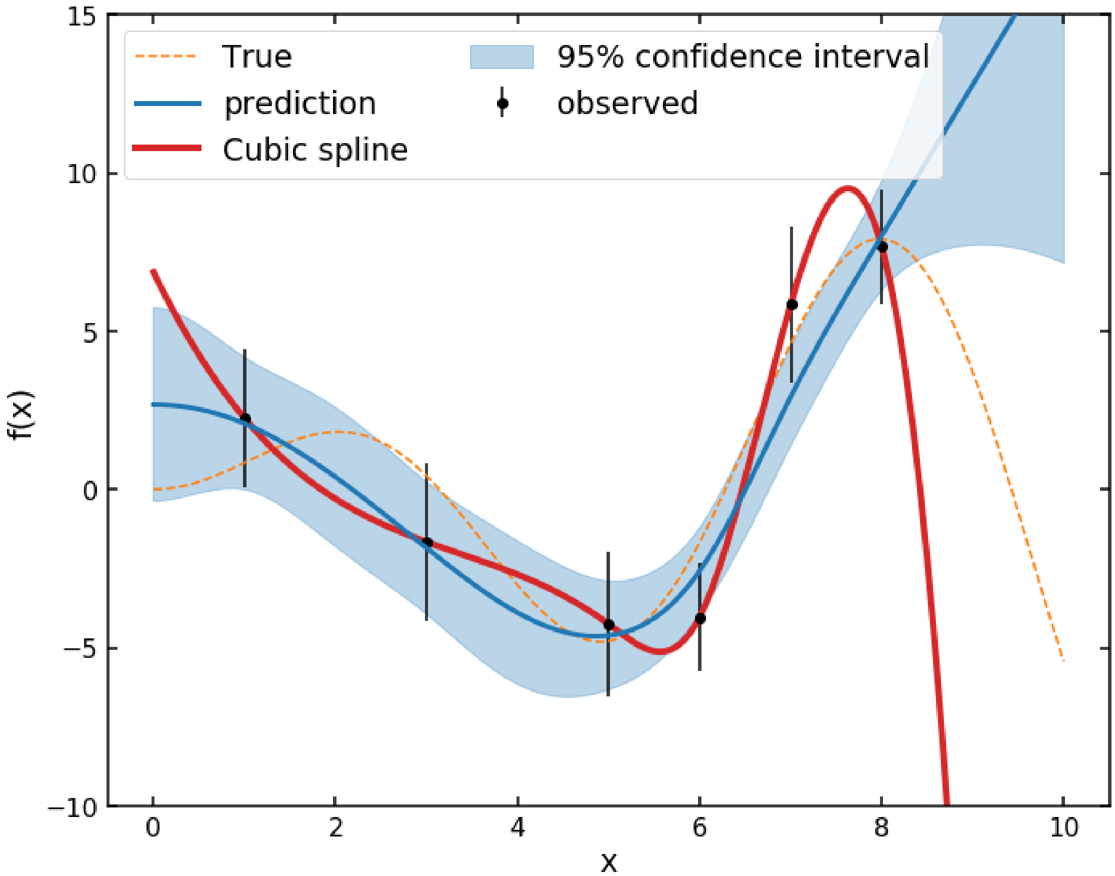
```
96.42041087387564**2 * CubicSpline
```

[25]:
```
mu_poly_err, var_poly_err = gp.predict(X_new[:,None], point=mp_poly_err,␣
 ↪diag=True)
sd_poly_err = np.sqrt(var_poly_err)
```

[26]:
```
tck = interpolate.splrep(x1, y2, k=3)
y_new = interpolate.splev(X_new, tck)
```

[27]:
```
plt.figure(5, figsize=(10, 8))
plt.errorbar(x1, y2, yerr=1.96*dy, fmt='ok', label='observed')

plt.plot(
    X_new,
    X_new * np.sin(X_new),
    '--',
    color='C1',
    label='True'
)

plt.plot(
    X_new.flatten(),
    mu_poly_err,
```

```
    color='C0',
    lw=3,
    zorder=3,
    label='prediction'
)

# plot 95% best fit region
plt.fill_between(
    X_new.flatten(),
    mu_poly_err - 1.96*sd_poly_err,
    mu_poly_err + 1.96*sd_poly_err,
    color='C0',
    alpha=0.3,
    zorder=1,
    label='95% confidence interval'
)

plt.plot(X_new, y_new, color='C3', label='Cubic spline', lw=4)

# labels and legend
plt.xlabel('x')
plt.ylabel('f(x)')
plt.ylim(-10, 15)
plt.legend(loc='upper left', ncol=2)
plt.tight_layout();
```



1.7 Other notes

- There are many covariance kernels you can pick;
  - `Constant`: a constant value that can be multiplied or added to any of the other kernels
  - `WhiteNoise`: a white noise kernel
  - `ExpQud`: exponentiated quadratic, smooth kernel parameterized by a length-scale
  - `RatQuad`: rational quadratic, a (infinite sum) mixture of different `ExpQud`'s each with different length-scales
  - `Exponential`: Simlar to `ExpQud` but without the square in the exponent.
  - `Marten52`: Marten 5/2 non-smooth generalization of `RBF`, parameterized by length-scale and smoothness
  - `Marten32`: Marten 3/2 non-smooth generalization of `RBF`, parameterized by length-scale and smoothness
  - `Cosine`: periodic kernel built with `cos`
  - `Linear`: a non-stationary kernel that can be used to fit a line
  - `Polynomial`: a non-stationary kernel commonly polynomial like fit
  - `Periodic`: periodic function kernel, parameterized by a length-scale and periodicity
- There are also three mean functions to choose from:

- `Zero`: The mean is all zeros (this is the default)
  - `Constant`: The mean is a constant value (i.e. a global y-offset)
  - `Linear`: The mean is a linear function (i.e. a polynomial)
- See https://docs.pymc.io/notebooks/GP-MeansAndCovs.html for examples of each kernel and mean function
- All `X` positions must be unique
- The computational complexity is $O(N^3)$ where N is the number of data point. If you have a large number of data points you can use an MCMC sampler instead of `find_MAP` to fit the data in a faster amout of time (see https://docs.pymc.io/notebooks/GP-Latent.html).

[ ]:

```

```