Gaussian Process Regression with Noise

Contents

- Objectives
- References
- Example: Gaussian process regression in 1D with fixed hyper-parameters and noise
- Diagnostics: How do you know if the fit is good?

```
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

- Perform Gaussian process regression with measurement noise

References

- Chapter 3 from C.E. Rasmussen's textbook on Gaussian processes

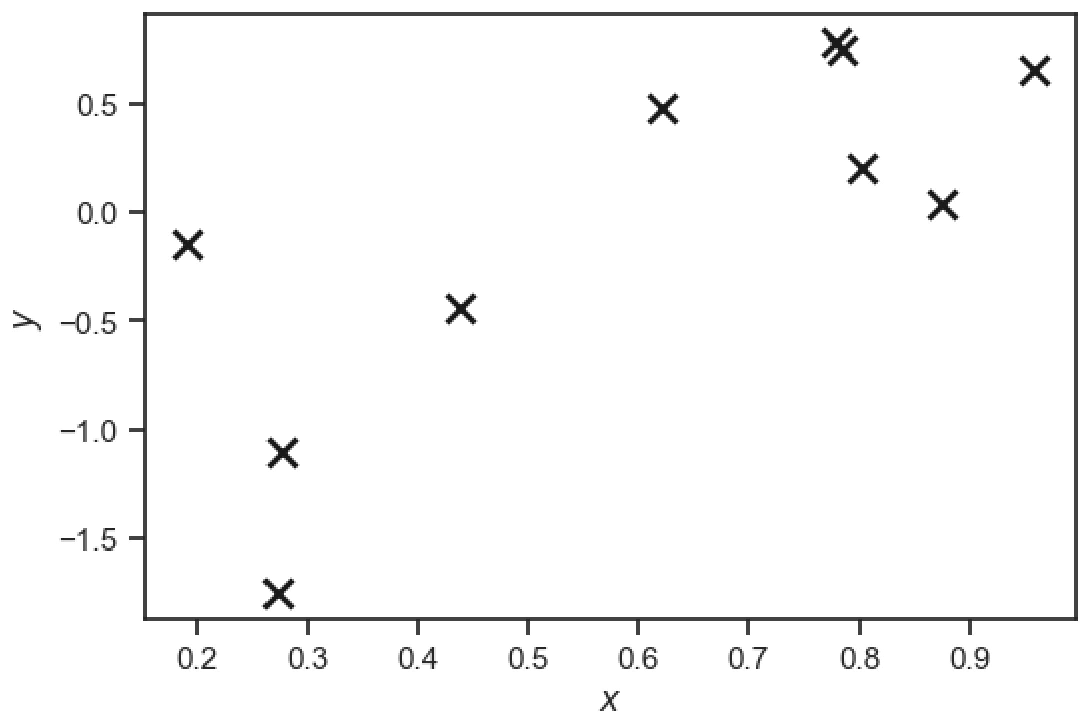
Example: Gaussian process regression in 1D with fixed hyper-parameters and noise

Let's generate some synthetic 1D data to work with:

```
np.random.seed(1234)

n = 10
X = np.random.rand(n, 1)
sigma = 0.4
f_true = lambda x: -np.cos(np.pi * x) + np.sin(4. * np.pi * x)
Y = f_true(X) + sigma * np.random.randn(X.shape[0], 1)

fig, ax = plt.subplots()
ax.plot(X, Y, 'kx', markersize=10, markeredgewidth=2)
ax.set_xlabel('$x$')
ax.set_ylabel('$y$');
```



Now, we will get started with the regression. First, import GPy:

```
import GPy

k = GPy.kern.RBF(1)
print(k)
```

```
  rbf.         |  value  |  constraints  |  priors
  variance     |    1.0  |      +ve      |
  lengthscale  |    1.0  |      +ve      |
```

The `variance` of the kernel is one. This seems reasonable. Let's leave it like that. The `lengthscale` seems to big. Let's change it to something reasonable (based on our expectations):

```
k.lengthscale = 0.1
print(k)
```

```
  rbf.         |  value  |  constraints  |  priors
  variance     |    1.0  |      +ve      |
  lengthscale  |    0.1  |      +ve      |
```

There is a possibility to choose a mean function, but for simplicity we are going to pick a zero mean function: $$m(x) = 0.$$ Now we put together the GP regression model as follows:

```
gpm = GPy.models.GPRegression(X, Y, k)
```

This model is automatically assuming that the likelihood is Gaussian (you can modify it if you wish). Where do can you find the σ^2 parameter specifying the likelihood noise? Here it is:

```
print(gpm)
```

```
Name : GP regression
Objective : 13.15046970174311
Number of Parameters : 3
Number of Optimization Parameters : 3
Updates : True
Parameters:
  GP_regression.           |  value  |  constraints  |  priors
  rbf.variance             |    1.0  |      +ve      |
  rbf.lengthscale          |    0.1  |      +ve      |
  Gaussian_noise.variance  |    1.0  |      +ve      |
```

We will talk about the meaning of all that later. For now, let's just fix the noise variance to something reasonable (actually the correct value):

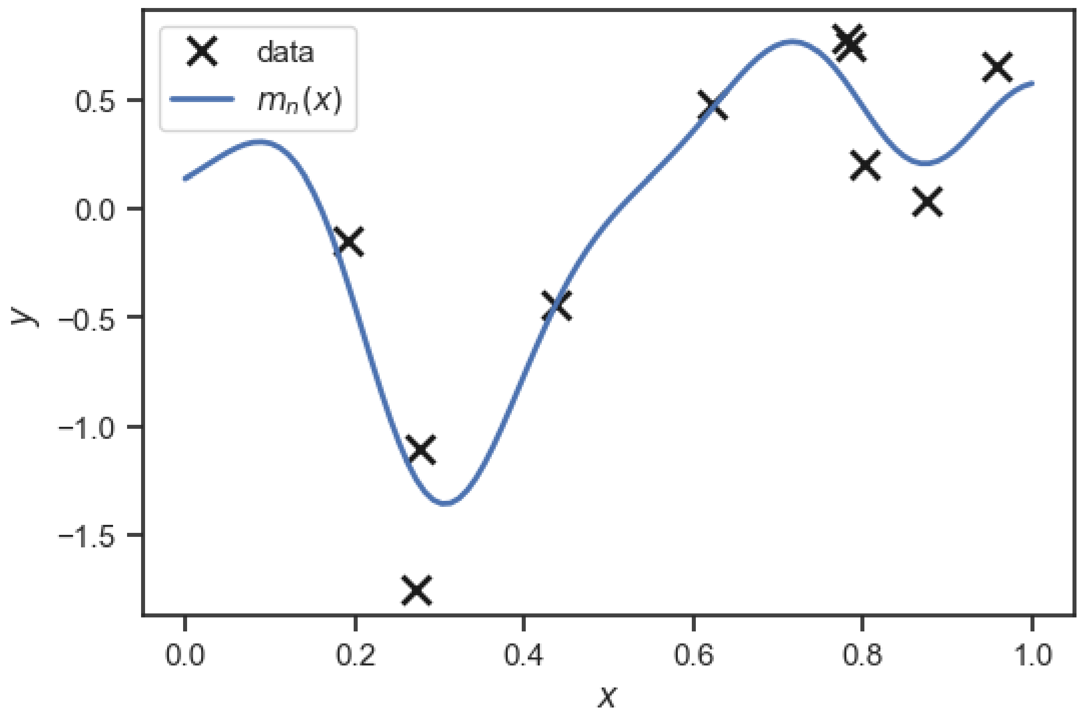
```
gpm.likelihood.variance = sigma ** 2
print(gpm)
```

```
Name : GP regression
Objective : 10.178171187043077
Number of Parameters : 3
Number of Optimization Parameters : 3
Updates : True
Parameters:
  GP_regression.           |                value  |  constraints  |  priors
  rbf.variance             |                  1.0  |      +ve      |
  rbf.lengthscale          |                  0.1  |      +ve      |
  Gaussian_noise.variance  |  0.16000000000000003  |      +ve      |
```

That's it. We have now specified the model completely. The posterior GP is completely defined. Where is the posterior mean $m_n(x)$ and variance $\sigma_n^2(x)$? You can get them like this:

```python
x_star = np.linspace(0, 1, 100)[:, None]
m_star, v_star = gpm.predict(x_star)

fig, ax = plt.subplots()
ax.plot(X, Y, 'kx', markersize=10, markeredgewidth=2, label='data')
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
ax.plot(x_star, m_star, lw=2, label='$m_n(x)$')
plt.legend(loc='best');
```

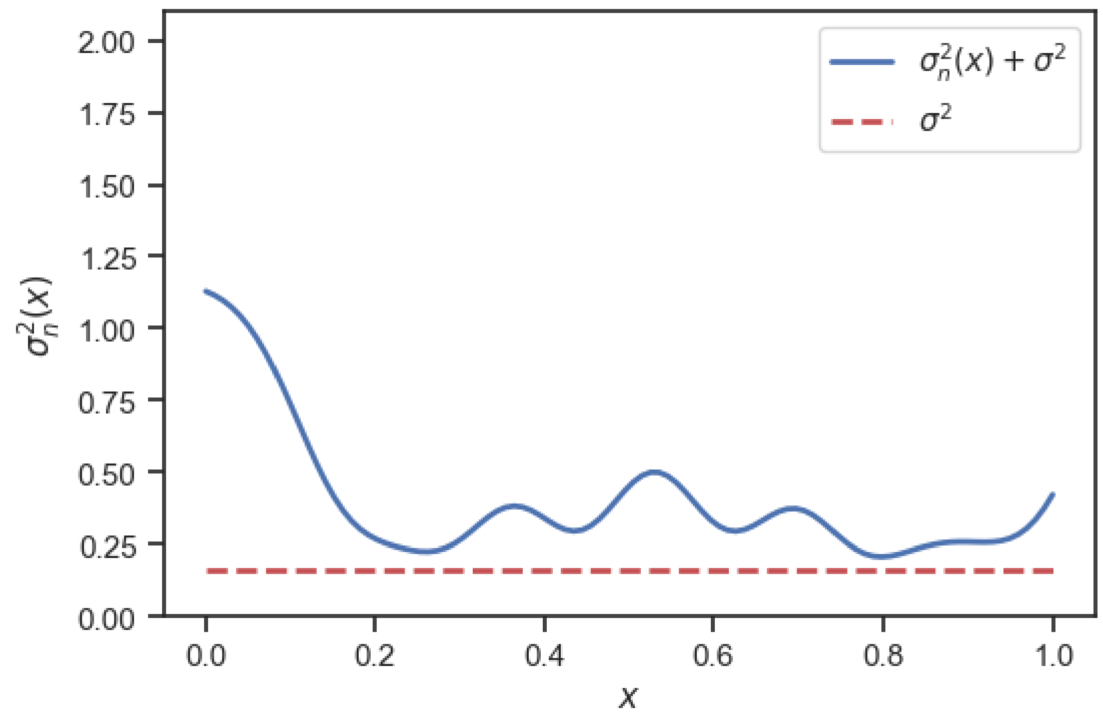


Extracting the variance is a bit more involved. Just a tiny bit though. This is because `v_star` returned by `gpm.predict` is not exactly $\sigma_n^2(x)$. It is actually $\sigma_n^2(x) + \sigma^2$ and not just $\sigma_n^2(x)$. Here, see it:

predictive variance (epistemic)

noise variance (aleatory)

```python
fig, ax = plt.subplots()
ax.plot(
    x_star,
    v_star,
    lw=2,
    label='$\sigma_n^2(x) + \sigma^2$'
)
ax.plot(
    x_star,
    gpm.likelihood.variance * np.ones(x_star.shape[0]),
    'r--',
    lw=2,
    label='$\sigma^2$'
)
ax.set_xlabel('$x$')
ax.set_ylabel('$\sigma_n^2(x)$')
ax.set_ylim(0, 2.1)
plt.legend(loc='best');
```

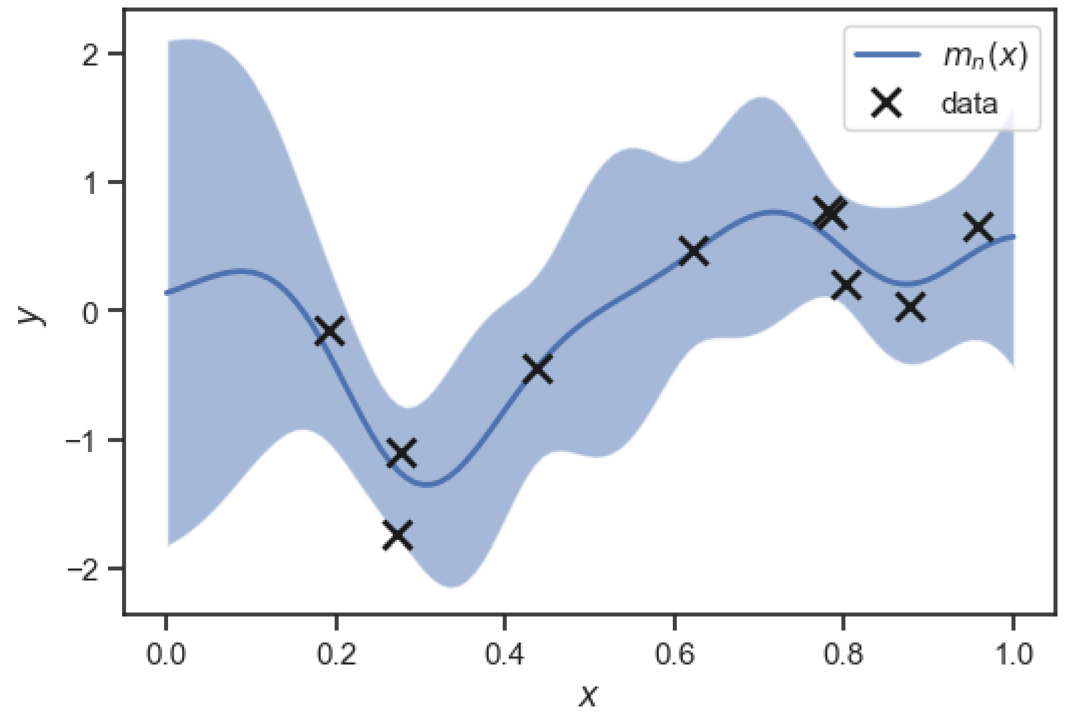


Notice that the variance is small wherever we have an observation. It is not, however, exactly, σ^2. It will become exactly σ^2 in the limit of many observations.

Having the posterior mean and variance, we can derive 95% predictive intervals for $f(x^*)$ and y^*. For $f(x^*)$ these are: $ $m_n(\mathbf{x}^*)) - 2\sigma_n(\mathbf{x}^*) \le f(\mathbf{x}^*) \le m_n(\mathbf{x}^*)) + 2\sigma_n(\mathbf{x}^*)$.$ Let's plot this:

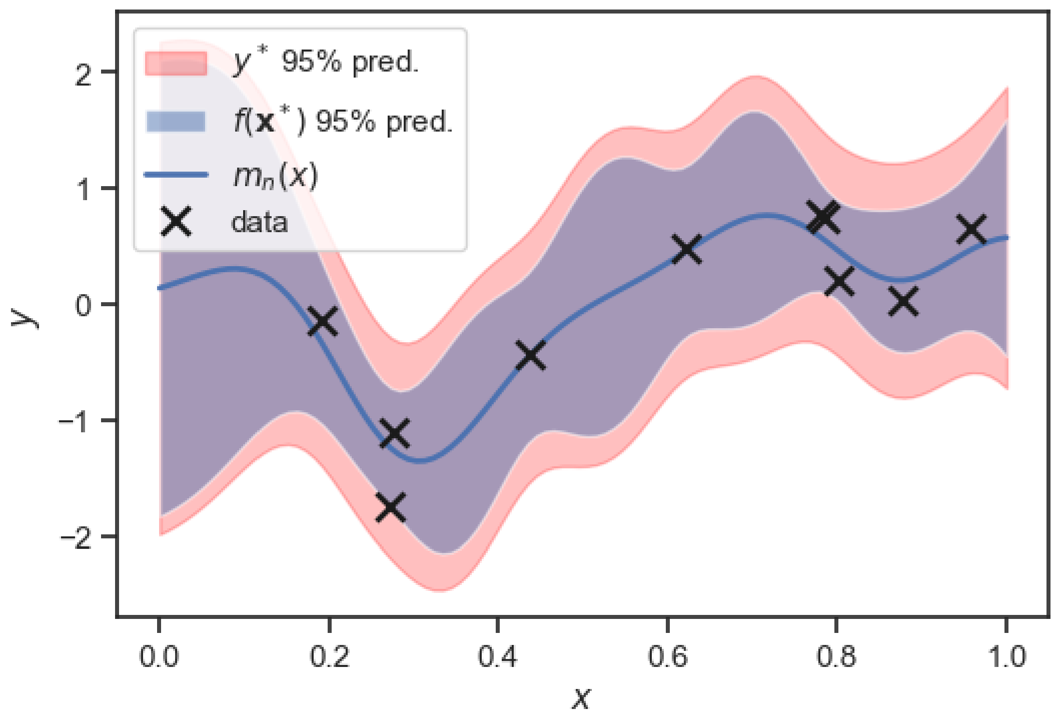
```
fig, ax = plt.subplots(dpi=100)
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
f_lower = m_star - 2.0 * np.sqrt(v_star - gpm.likelihood.variance)
f_upper = m_star + 2.0 * np.sqrt(v_star - gpm.likelihood.variance)
ax.fill_between(
    x_star.flatten(),
    f_lower.flatten(),
    f_upper.flatten(),
    alpha=0.5
)
ax.plot(x_star, m_star, lw=2, label='$m_n(x)$')
ax.plot(
    X,
    Y,
    'kx',
    markersize=10,
    markeredgewidth=2,
    label='data'
)
plt.legend(loc='best');
```

just the epistemic uncertainty



Now, on the same plot, let's superimpose our predictive error bar about y^*. This is: $ $m_n(\mathbf{x}^*)) - 2\sqrt{\sigma_n^2(\mathbf{x}^*) + \sigma^2} \le f(\mathbf{x}^*) \le m_n(\mathbf{x}^*)) + 2\sqrt{\sigma_n(\mathbf{x}^*) + \sigma^2}$.$ Let's use red color for this:

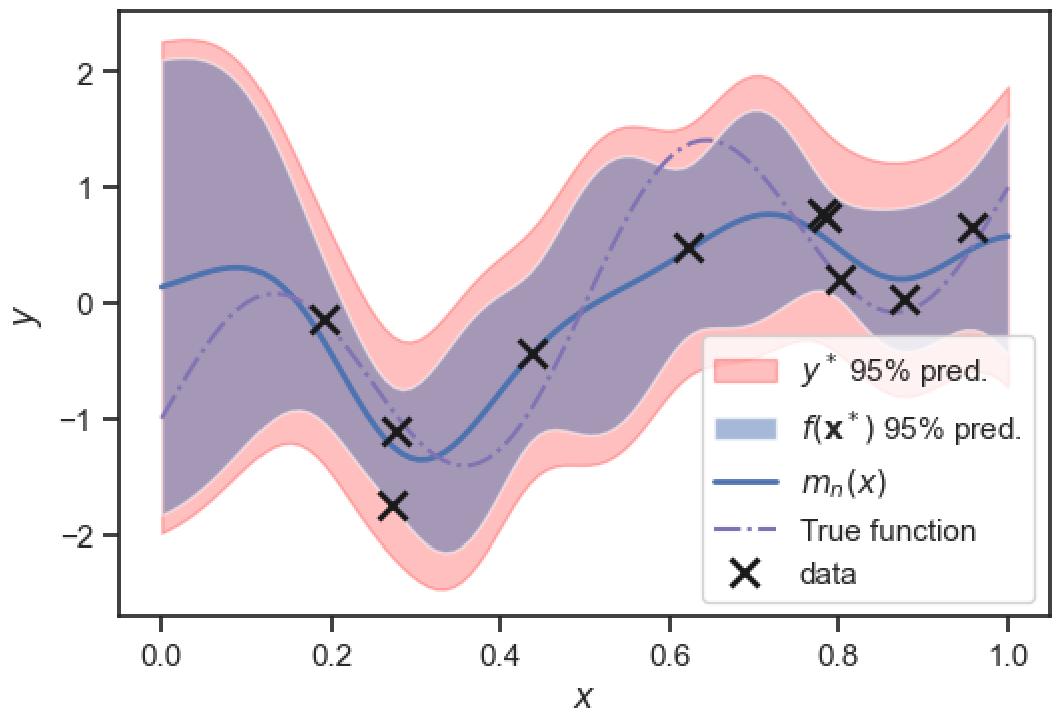
```
fig, ax = plt.subplots(dpi=100)
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
f_lower = m_star - 2.0 * np.sqrt(v_star - gpm.likelihood.variance)
f_upper = m_star + 2.0 * np.sqrt(v_star - gpm.likelihood.variance)
y_lower = m_star - 2.0 * np.sqrt(v_star)
y_upper = m_star + 2.0 * np.sqrt(v_star)
ax.fill_between(
    x_star.flatten(),
    y_lower.flatten(),
    y_upper.flatten(),
    color='red',
    alpha=0.25,
    label='$y^*$ 95% pred.'
)
ax.fill_between(
    x_star.flatten(),
    f_lower.flatten(),
    f_upper.flatten(),
    alpha=0.5,
    label='$f(\mathbf{x}^*)$ 95% pred.'
)
ax.plot(x_star, m_star, lw=2, label='$m_n(x)$')
ax.plot(X, Y, 'kx', markersize=10, markeredgewidth=2, label='data')
plt.legend(loc='best');
```



Let’s also put the correct function there for comparison:

```
fig, ax = plt.subplots(dpi=100)
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
f_lower = m_star - 2.0 * np.sqrt(v_star - gpm.likelihood.variance)
f_upper = m_star + 2.0 * np.sqrt(v_star - gpm.likelihood.variance)
y_lower = m_star - 2.0 * np.sqrt(v_star)
y_upper = m_star + 2.0 * np.sqrt(v_star)
ax.fill_between(
    x_star.flatten(),
    y_lower.flatten(),
    y_upper.flatten(),
    color='red',
    alpha=0.25,
    label='$y^*$ 95% pred.')
ax.fill_between(
    x_star.flatten(),
    f_lower.flatten(),
    f_upper.flatten(),
    alpha=0.5,
    label='$f(\mathbf{x}^*)$ 95% pred.'
)
ax.plot(x_star, m_star, lw=2, label='$m_n(x)$')
ax.plot(x_star, f_true(x_star), 'm-.', label='True function')
ax.plot(X, Y, 'kx', markersize=10, markeredgewidth=2, label='data');
plt.legend(loc='best')
```

```
<matplotlib.legend.Legend at 0x28ec404c0>
```



You see that the true function is almost entirely within the blue bounds. It is ok that it is a little bit off, becuase these are 95% prediction intervals. About 5% of the function can be off.

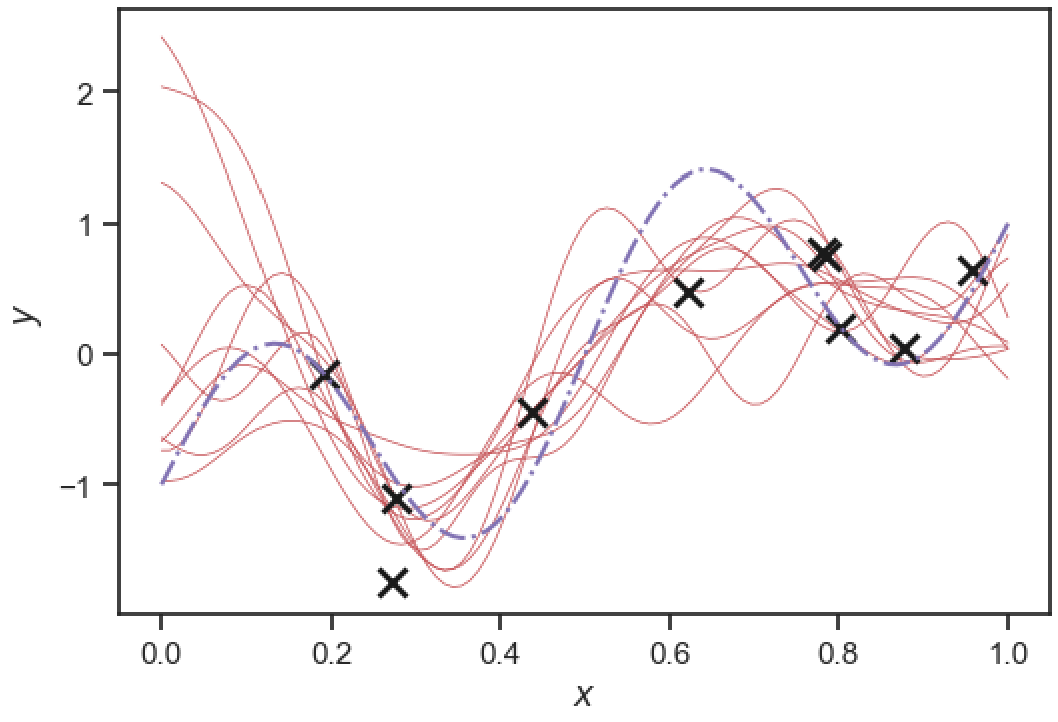
Let's now take some samples from the posterior:

```
f_post_samples = gpm.posterior_samples_f(x_star, 10)
print(f_post_samples.shape)
```

```
(100, 1, 10)
```

This is `test points x number of outputs (1 here) x number of samples`. Let's plot them along with the data and the truth:

```
fig, ax = plt.subplots()
ax.plot(x_star, f_post_samples[:, 0, :], 'r', lw=0.5)
ax.plot(X, Y, 'kx', markersize=10, markeredgewidth=2, label='data');
ax.plot(x_star, f_true(x_star), 'm-.', label='True function')
ax.set_xlabel('$x$')
ax.set_ylabel('$y$');
```



The following interactive function regenerates the figures above allowing you to experiment with various choices of the hyperparameters.

```python
from ipywidgets import interact_manual

def plot_1d_regression(
    x_star,
    gpm,
    ax=None,
    f_true=None,
    num_samples=10
):
    """Plot the posterior predictive.

    Arguments
    x_start  --  The test points on which to evaluate.
    gpm      --  The trained model.

    Keyword Arguments
    ax          --  An axes object to write on.
    f_true      --  The true function.
    num_samples --  The number of samples.
    """
    m_star, v_star = gpm.predict(x_star)

    f_lower = (
        m_star - 2.0 * np.sqrt(v_star - gpm.likelihood.variance)
    )
    f_upper = (
        m_star + 2.0 * np.sqrt(v_star - gpm.likelihood.variance)
    )
    y_lower = m_star - 2.0 * np.sqrt(v_star)
    y_upper = m_star + 2.0 * np.sqrt(v_star)

    if ax is None:
        fig, ax = plt.subplots()

    ax.plot(x_star, m_star, lw=2, label='$m_n(x)$')

    ax.fill_between(
        x_star.flatten(),
        f_upper.flatten(),
        y_upper.flatten(),
        color="red",
        alpha=0.25
    )
    ax.fill_between(
        x_star.flatten(),
        f_lower.flatten(),
        f_upper.flatten(),
        color="blue",
        alpha=0.25,
        label='$f(\mathbf{x}^*)$ 95% pred.'
    )
    ax.fill_between(
        x_star.flatten(),
        y_lower.flatten(),
        f_lower.flatten(),
        color="red",
        alpha=0.25,
        label='$y^*$ 95% pred.'
    )

    if f_true is not None:
        ax.plot(
            x_star,
            f_true(x_star),
            'm-.',
            label='True function'
        )

    ax.plot(gpm.X,
            gpm.Y,
            'kx',
            markersize=10,
            markeredgewidth=2,
            label='Observations'
    )

    if num_samples > 0:
        f_post_samples = gpm.posterior_samples_f(
            x_star,
            num_samples
        )
        ax.plot(x_star, f_post_samples[:, 0, :], 'b--', lw=1)
        ax.plot([], [], 'b--', lw=0.5, label="Posterior samples")
    ax.set_xlabel('$x$')

    ax.set_ylabel('$y$')

    plt.legend(loc='best')

    plt.tight_layout()


@interact_manual(
    kern_variance=(0.01, 10.0, 0.01),
    kern_lengthscale=(0.01, 1.0, 0.01),
    like_variance=(0.01, 1.0, 0.01)
)
def analyze_and_plot_gp_ex1(kern_variance=1.0, kern_lengthscale=0.1, like_variance=0.4):
    """
    Performs GP regression with given kernel variance, lengthcale and likelihood variance.
    """
    k = GPy.kern.Matern32(1)
    gp_model = GPy.models.GPRegression(X, Y, k)

    gp_model.kern.variance = kern_variance
    gp_model.kern.lengthscale = kern_lengthscale
    gp_model.likelihood.variance = like_variance

    print(gp_model)

    x_star = np.linspace(0, 1, 100)[:, None]

    plot_1d_regression(x_star, gp_model, f_true=f_true)
```

Diagnostics: How do you know if the fit is good?

To objective test the resulting model we need a *validation dataset* consisting of inputs:

$$\mathbf{x}_{1:n^v}^v = (\mathbf{x}_1^v, \dots, \mathbf{x}_{n^v}^v),$$

and corresponding, observed outputs:

$$\mathbf{y}_{1:n^v}^v = (y_1^v, \dots, y_{n^v}^v).$$

We will use this validation dataset to define some diagnostics. Let's do it directly through the 1D example above. First, we generate some validation data:

```
n_v = 100
X_v = np.random.rand(n_v)[:, None]
Y_v = f_true(X_v) + sigma * np.random.randn(n_v, 1)
```

Point-predictions

Point-predictions only use $m_n\left(\mathbf{x}_i^v\right)$. Of course, when there is a lot of noise, they are not very useful. But let's look at what we get anyway. (In the questions section I will ask you to reduce the noise and repeat).

The simplest thing we can do is to compare y_i^v to $m_n\left(\mathbf{x}_i^v\right)$. We start with the *mean square error*:

$$\text{MSE} := \frac{1}{n^v}\sum_{i=1}^{n^v}\left[y_i^v - m_n\left(\mathbf{x}_i^v\right)\right]^2.$$

```
m_v, v_v = gpm.predict(X_v)
mse = np.mean((Y_v - m_v) ** 2)
print(f'MSE = {mse:1.2f}')
```

```
MSE = 0.36
```

This is not very intuitive though. An somewhat intuitive measure is coefficient of determination also known as R^2, *R squared*. It is defined as:

$$R^2 = 1 - \frac{\sum_{i=1}^{n^v}\left[y_i^v - m_n(\mathbf{x}_i^v)\right]^2}{\sum_{i=1}^{n^v}\left[y_i^v - \bar{y}^v\right]^2},$$

where $\bar{y}^v$ is the mean of the observed data:

$$\bar{y}^v = \frac{1}{n^v}\sum_{i=1}^{n^v} y_i^v.$$

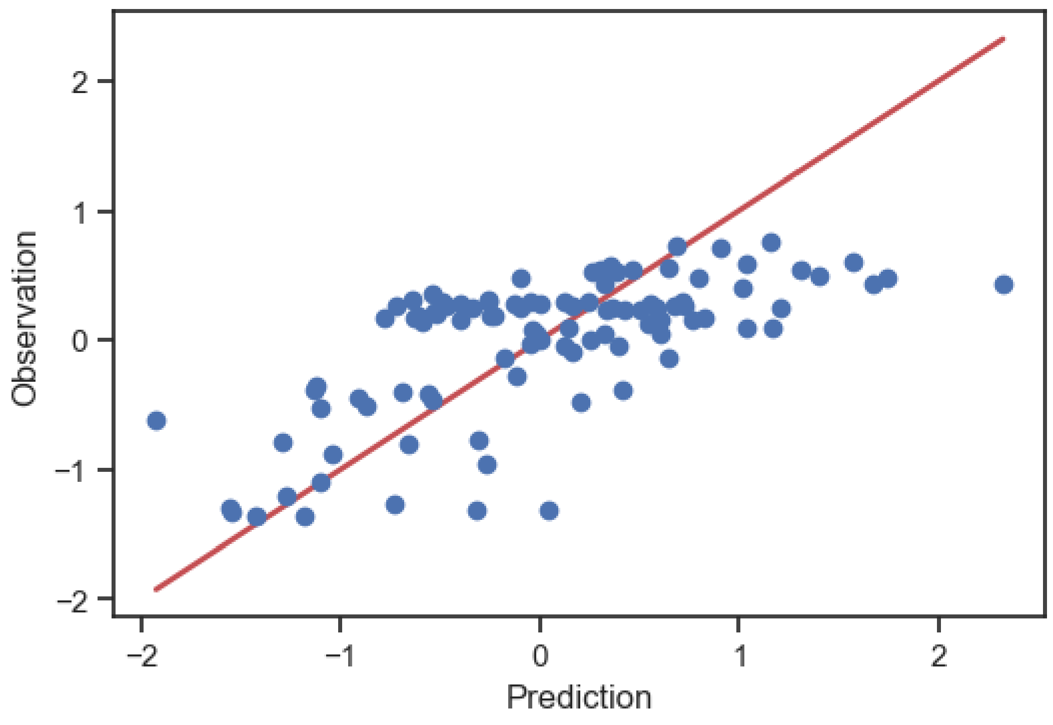
The interpretation of R^2, and take this with a grain of salt, is that it gives the percentage of variance of the data explained by the model. A score of $R^2 = 1$, is a perfect fit. In our data we get:

```
R2 = 1.0 - np.sum((Y_v - m_v) ** 2) / np.sum((Y_v - np.mean(Y_v)) ** 2)
print(f'R2 = {R2:1.2f}')
```

```
R2 = 0.43
```

Finally, on point-predictions, we can simply plot the predictions vs the observations:

```
fig, ax = plt.subplots()
y_range = np.linspace(Y_v.min(), Y_v.max(), 50)
ax.plot(y_range, y_range, 'r', lw=2)
ax.plot(Y_v, m_v, 'bo')
ax.set_xlabel('Prediction')
ax.set_ylabel('Observation');
```



Statistical diagnostics

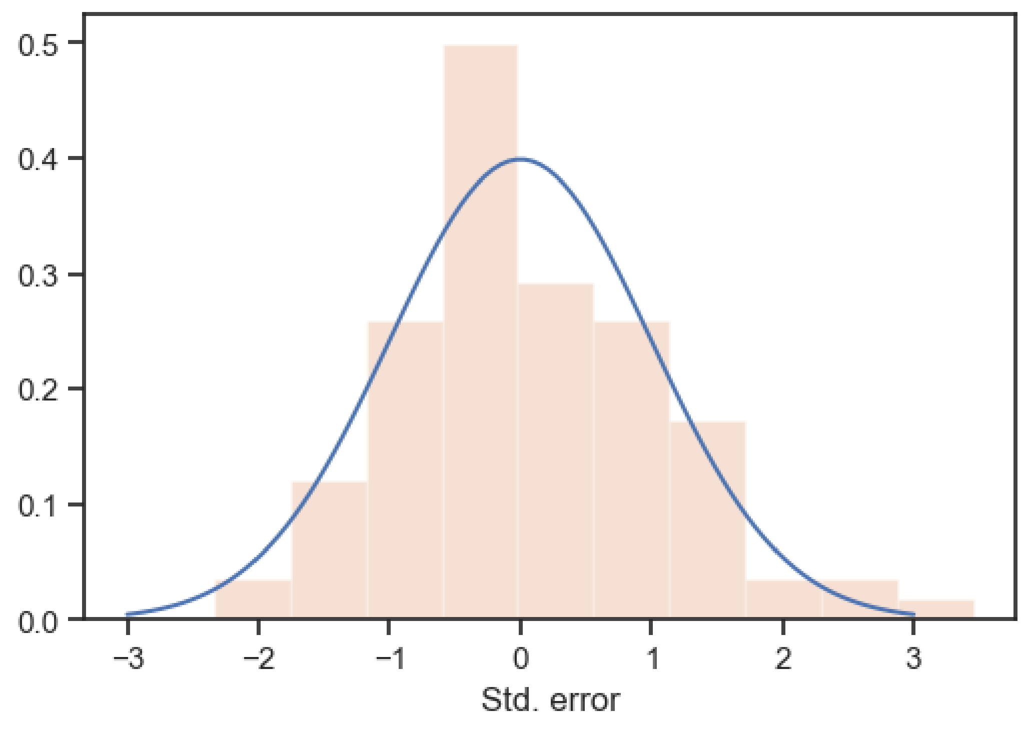
Statistical diagnostics compare the predictive distribution to the distribution of the validation dataset. The way to start, are the standarized errors defined by:

$$e_i = \frac{y_i^v - m_n\left(\mathbf{x}_i^v\right)}{\sigma_n\left(\mathbf{x}_i^v\right)}.$$

Now, if our model is correct, the standarized errors must be distributed as a standard normal $N(0,1)$ (why?). There are various plots that you can do to test that. First, the histogram of the standarized errors:

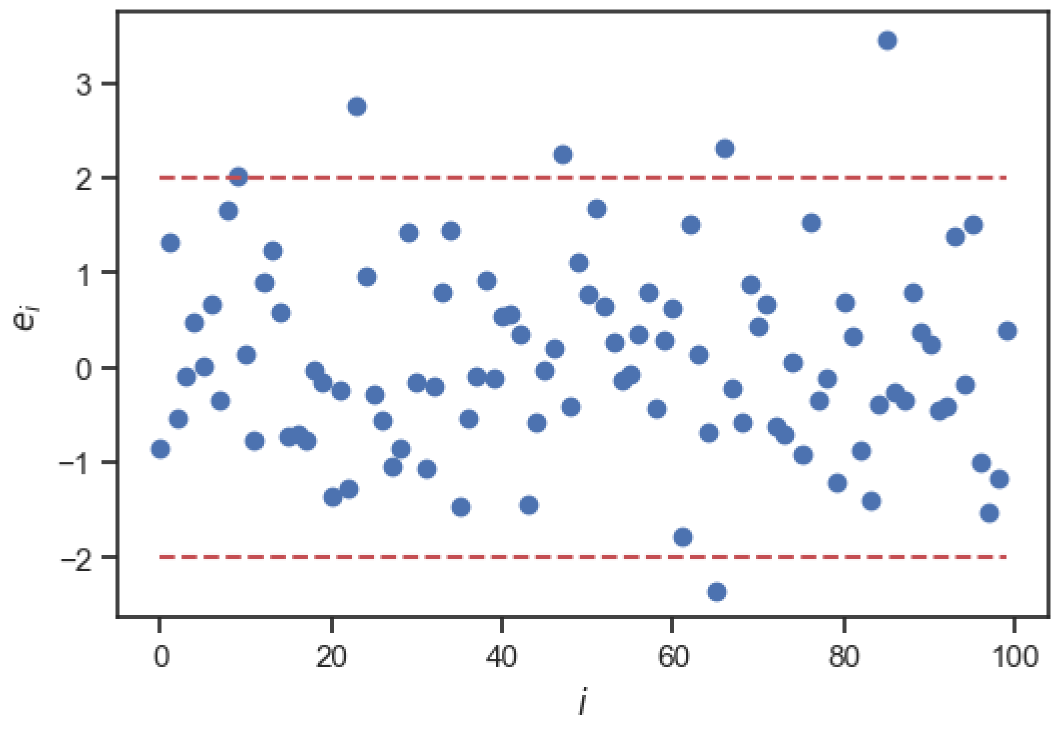
```
import scipy.stats as st

s_v = np.sqrt(v_v)
e = (Y_v - m_v) / s_v
fig, ax = plt.subplots()
zs = np.linspace(-3.0, 3.0, 100)
ax.plot(zs, st.norm.pdf(zs))
ax.hist(e, density=True, alpha=0.25)
ax.set_xlabel('Std. error');
```



Close, but not perfect. Another common plot is this:

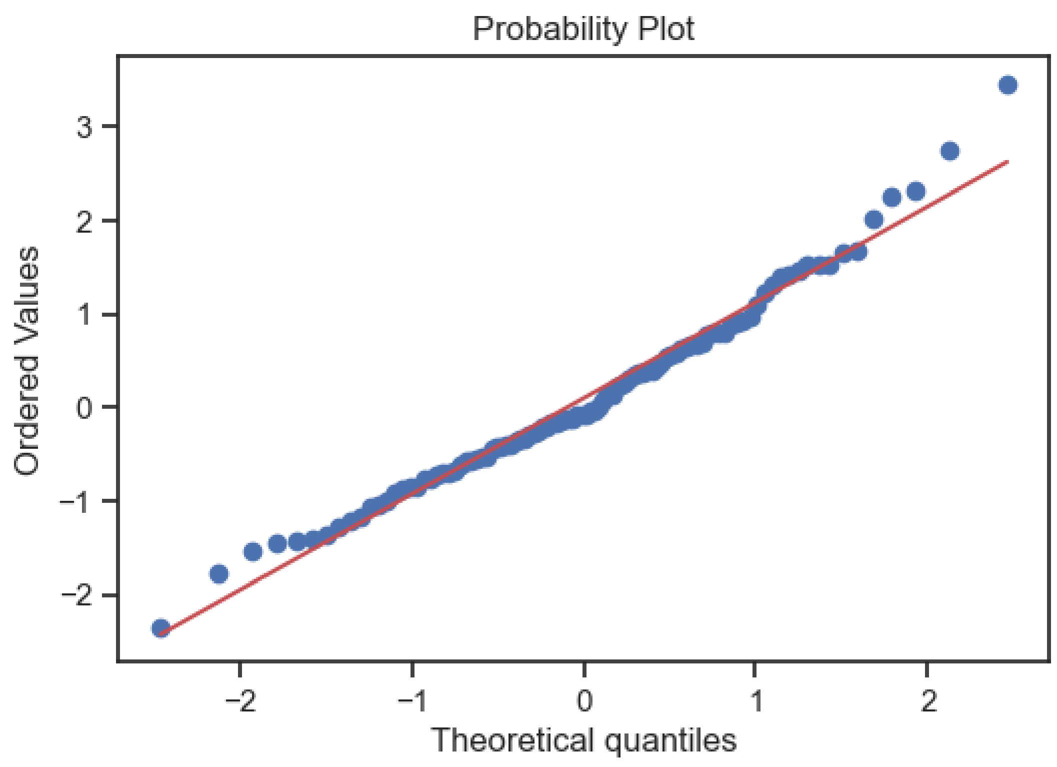
```
fig, ax = plt.subplots()
ax.plot(e, 'o')
ax.plot(np.arange(e.shape[0]), 2.0 * np.ones(e.shape[0]), 'r--')
ax.plot(np.arange(e.shape[0]), -2.0 * np.ones(e.shape[0]), 'r--')
ax.set_xlabel('$i$')
ax.set_ylabel('$e_i$');
```



Where the red lines indicate the 95% quantiles of the standard normal. This means that if 5% of the errors are inside, then we are good to go.

Yet another plot yielding the same information is the q-q plot comparing the empirical quantiles of the standarized errors to what they are supposed to be, i.e., to the quantiles of $N(0,1)$:

```
fig, ax = plt.subplots(dpi=100)
st.probplot(e.flatten(), dist=st.norm, plot=ax);
```



Note on Gaussian process diagnostics

For a more detailed description of GP regression diagnostics, please see this paper.

Questions

In the interactive tool above:

- Experiment with differnet lengthscales for the kernel. You need to click on `Run Interact` for the code to run. What happens to the posterior mean and the 95% predictive error bar as the lengthscale increases (decreases)?
- Experiment with difference likelihood variances. What happens for very big variances? What happens for very small variances?
- Experiment with different kernel variances. This the s^2 parameter of the squared exponential covariance function. It specifies our prior variance about the function values. What is its effect?
- Imagine that, as it would be the case in reality, you do not know the true function. How would you pick the correct values for the hyperparameters specifying the kernel?
- Try some other kernels. Edit the function `analyze_and_plot_gp_ex1` and change the line `k = GPy.kern.RBF(1)` to `k = GPy.kern.Matern52(1)`. This is a kernel that is less regular than the RBF. What do you observe? Then try `k = GPy.kern.Matern32(1)`. Then `k = GPy.kern.Exponential(1)`. The last one is continuous but nowhere differentiable. How can you pick the right kernel?
- Experiment with larger number of training points n. Are the models becoming better according to the metrics we defined above?
- Experiment with smaller measurement noises σ. What do you observe? Which diagnostics make sense for very small σ's?

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