

Tuning the Hyperparameters

Contents

- [Objectives](#)
- [References](#)
- [Example: Gaussian process regression with fitted hyperparameters](#)
- [Questions](#)

```
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

- Tune the hyper-parameters of a Gaussian process using maximum marginal likelihood.

References

- [Chapter 5 of GP for ML textbook](#).

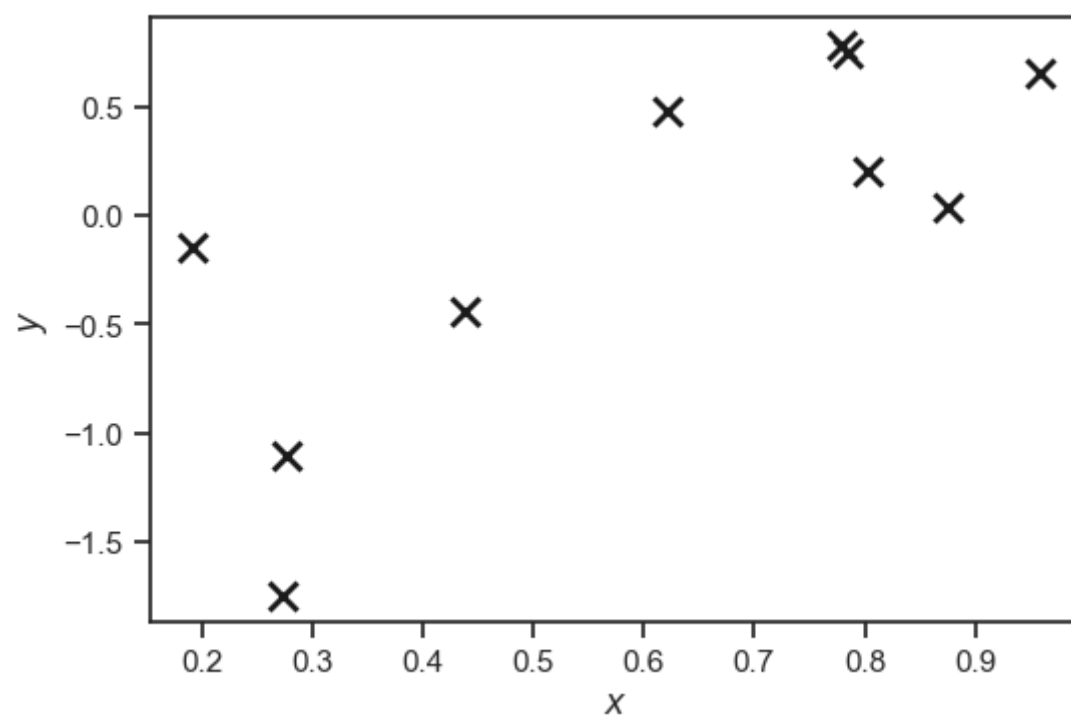
Example: Gaussian process regression with fitted hyperparameters

Let's generate synthetic data:

```
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('$y$');
```



Some of our code for plotting posterior predictives:

```

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

    Arguments
    x_star -- 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.'
    )

    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")

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

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

```

```
plt.legend(loc='best')

plt.tight_layout()
```

Let's pick a squared exponential kernel and make a model with Gaussian likelihood (the default choice):

```
import GPy

k = GPy.kern.RBF(1)
gpm = GPy.models.GPRegression(X, Y, k)

print(gpm)
```

```
Name : GP regression
Objective : 13.029643687983668
Number of Parameters : 3
Number of Optimization Parameters : 3
Updates : True
Parameters:
```

GP_regression.	value	constraints	priors
rbf.variance	1.0	+ve	
rbf.lengthscale	1.0	+ve	
Gaussian_noise.variance	1.0	+ve	

Let's explain what all this means. Notice that there are some default values for the hyperparameters (they are all one). Also, notice that **GPy** is keeping track of how many parameters it needs to fit. Here we have three parameters (s , ℓ and σ). The second column are constraints for the parameters. The **+ve** term means that the corresponding hyperparameter has to be positive. Notice that there is nothing in the **priors** column. This is because we have set no priors right now. When this happens, **GPy** assumes that we assign a flat prior, i.e., here it assumes that we have assigned $p(\theta) \propto 1$ and $p(\sigma) \propto 1$. That's not the best choice, but it should be ok for now.

Now, pay attention to the **Objective**. This is the $-\log p(\theta, \sigma | \mathcal{D})$ for the current choice of parameters. Let's now find the MAP: [Maximum a Posteriori](#)

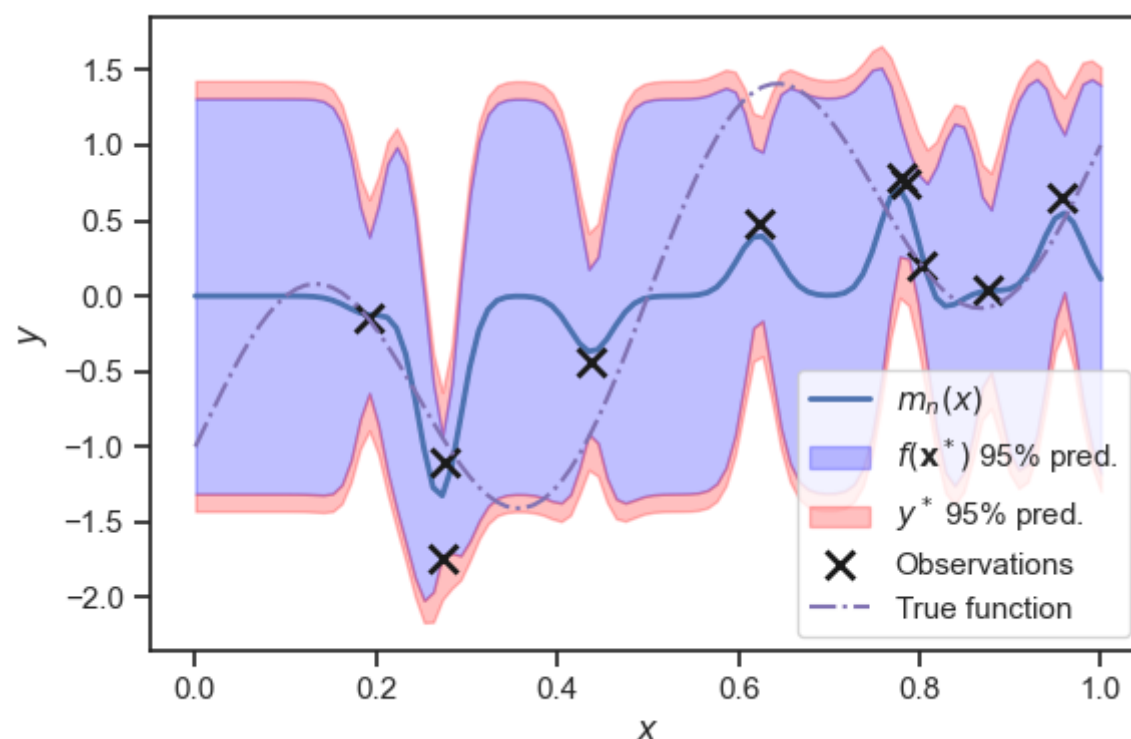
```
gpm.optimize(messages=True)
print(gpm)
```

```
Name : GP regression
Objective : 9.376317801621127
Number of Parameters : 3
Number of Optimization Parameters : 3
Updates : True
Parameters:
```

GP_regression.	value	constraints	priors
rbf.variance	0.4296554984401683	+ve	
rbf.lengthscale	0.023487942263585294	+ve	
Gaussian_noise.variance	0.07952049967422603	+ve	

Ok. We did find something with higher posterior value. Let's plot the prediction:

```
xstar = np.linspace(0, 1, 100)[: , None]
plot_1d_regression(xstar, gpm, f_true=f_true, num_samples=0)
```



Admititdly, this doesn't look very good. (Of course, we can tell this only because we know the truth). It seems that the assigned lengthscale is too small. Also, the likelihood variance seems smaller than it really is. **What do we do now? You have two choices:**

- You encode some prior knowledge and repeat.
- You add some more data and repeat.

Let's start with some prior knowledge and let the other item for the questions section. Let's say that we know that the noise variance. How do we encode this? Here you go:

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

Name : GP regression
Objective : 9.649326603819407
Number of Parameters : 3
Number of Optimization Parameters : 2
Updates : True
Parameters:

GP_regression.	value	constraints	priors
rbf.variance	0.4296554984401683	+ve	
rbf.lengthscale	0.023487942263585294	+ve	
Gaussian_noise.variance	0.16000000000000003	+ve fixed	

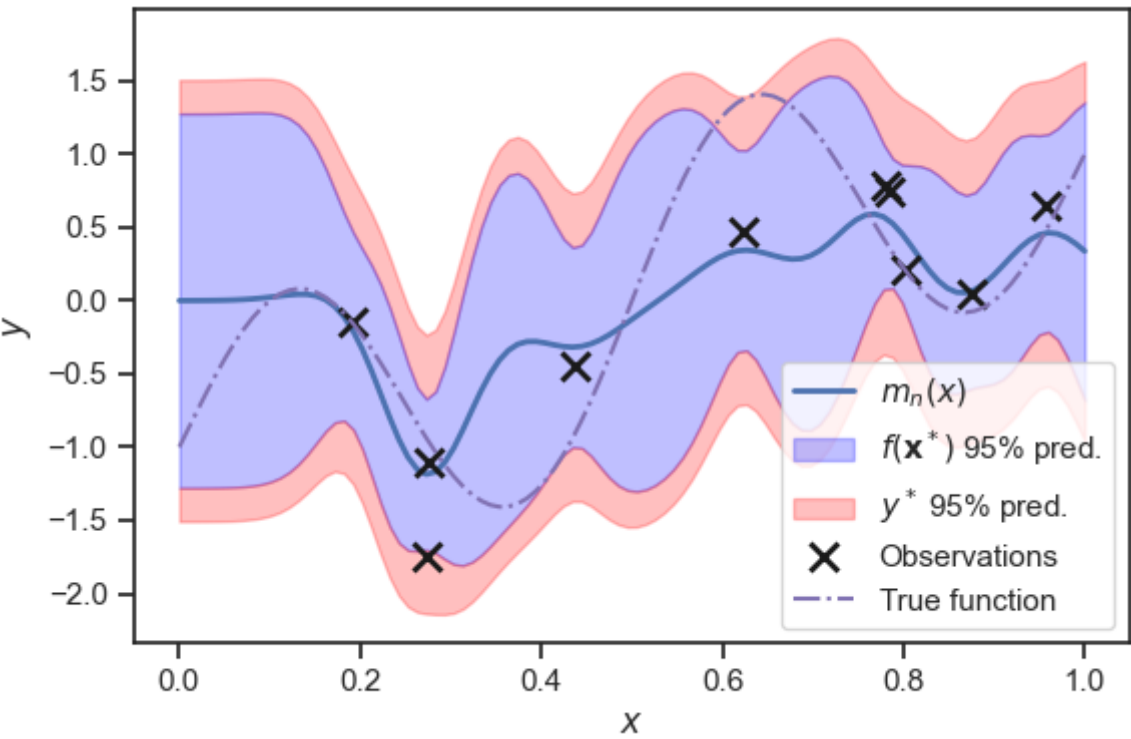
Notice that it now **GP**y reports that the likelihood variance is fixed. Let's repeat the optimization:

```
gpm.optimize(messages=True)
print(gpm)
```

Name : GP regression
Objective : 9.574736850990453
Number of Parameters : 3
Number of Optimization Parameters : 2
Updates : True
Parameters:

GP_regression.	value	constraints	priors
rbf.variance	0.4079238212592344	+ve	
rbf.lengthscale	0.0504172036832454	+ve	
Gaussian_noise.variance	0.16000000000000003	+ve fixed	

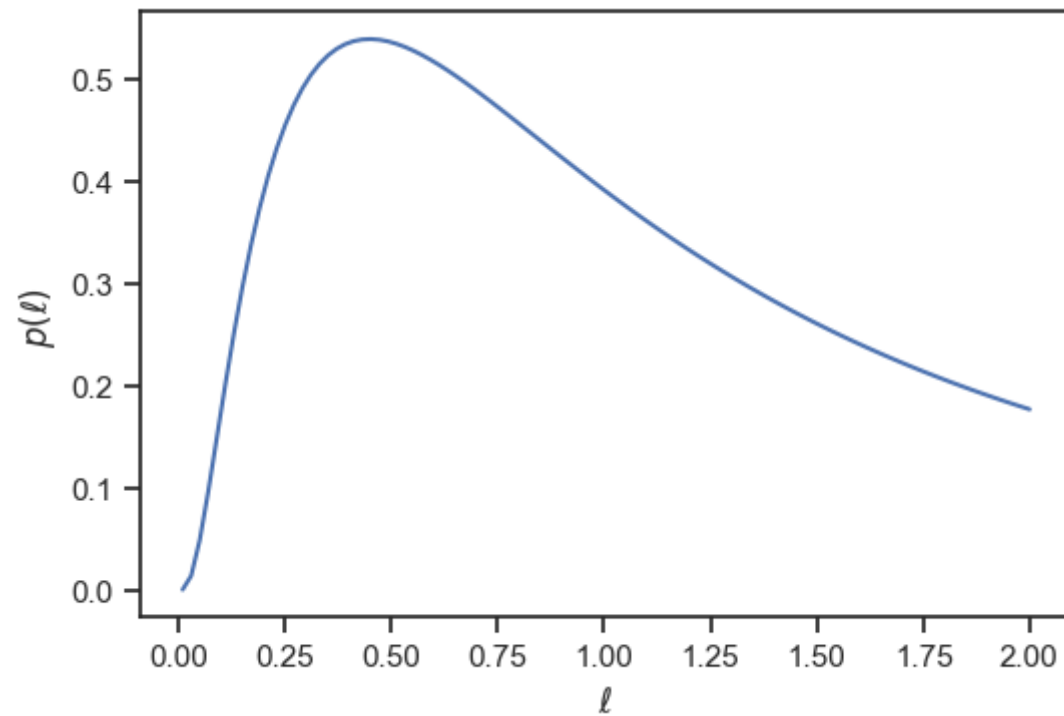
```
plot_1d_regression(xstar, gpm, f_true=f_true, num_samples=0)
```



This looks better. But it seems that the automatically selected lengthscale is smaller than the true one. (Of course, in reality, we don't really know what the true lengthscale is). Let's assign a prior probability density on the lengthscale which pushes it to be greater. **Since we are dealing with a positie parameter and we don't know much about it, let's assign an exponential prior with a rate of 2 (which will yield an expectation of 0.5): $\ell \sim \text{Log-N}(0.2, 1)$.**

```
ell_prior = GPy.priors.LogGaussian(.2, 1.0)

fig, ax = plt.subplots()
ells = np.linspace(0.01, 2.0, 100)
ax.plot(ells, ell_prior.pdf(ells))
ax.set_xlabel('$\ell$')
ax.set_ylabel('$p(\ell)$');
```



Now here is how you can set it:

```
gpm.kern.lengthscale.set_prior(ell_prior)
print(gpm)
```

reconstraining parameters GP_regression.rbf.lengthscale

Name : GP regression
 Objective : 15.598610190278492
 Number of Parameters : 3
 Number of Optimization Parameters : 2
 Updates : True
 Parameters:

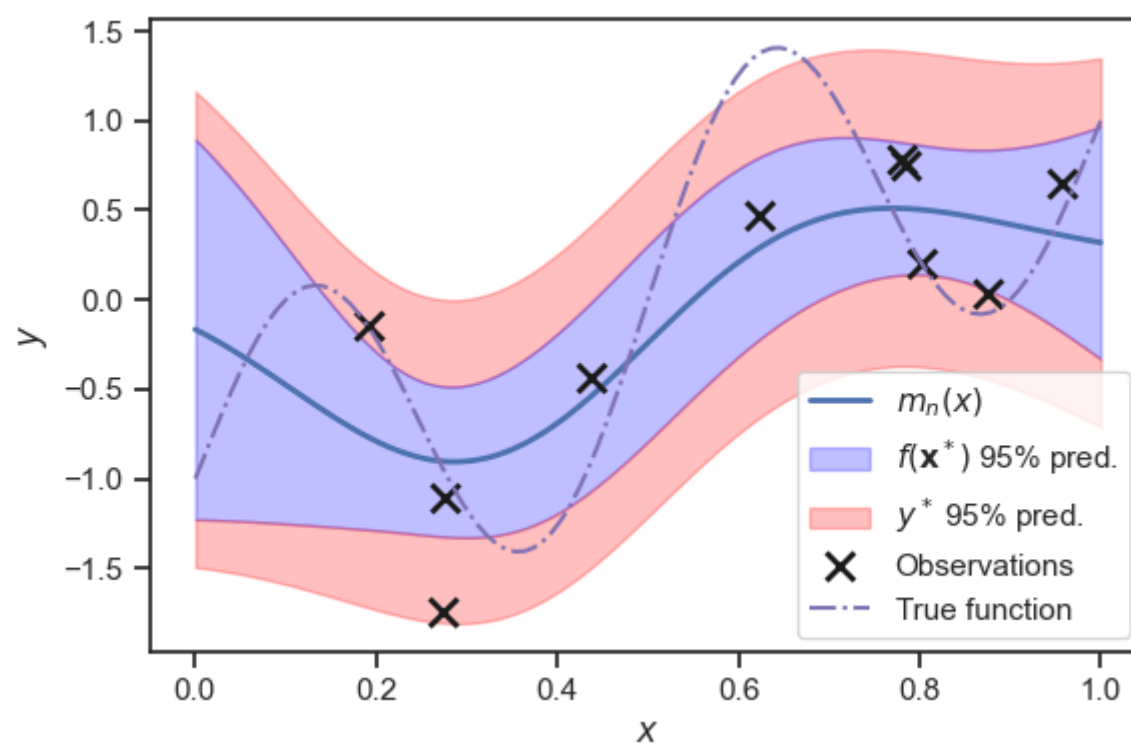
GP_regression.	value	constraints	priors
rbf.variance	0.4079238212592344	+ve	
rbf.lengthscale	0.0504172036832454	+ve	lnN(0.2, 1)
Gaussian_noise.variance	0.16000000000000003	+ve fixed	

```
gpm.optimize(messages=True)
print(gpm)
```

Name : GP regression
 Objective : 12.755925770647496
 Number of Parameters : 3
 Number of Optimization Parameters : 2
 Updates : True
 Parameters:

GP_regression.	value	constraints	priors
rbf.variance	0.46913636718256563	+ve	
rbf.lengthscale	0.23199119551940342	+ve	lnN(0.2, 1)
Gaussian_noise.variance	0.16000000000000003	+ve fixed	

```
plot_1d_regression(xstar, gpm, f_true=f_true, num_samples=0)
```



That's better, but not perfect. But remember: You don't know what the truth is...

Questions

Let's investigate what happens to the previous examples as you increase the number of observations.

- Rerun everything gradually increasing the number of samples from $n = 10$ to $n = 100$. Notice that as the number of samples increases it doesn't really matter what your prior knowledge is. As a matter of fact, for the largest number of samples you try, pick a very wrong prior probability for ℓ . See what happens.
- Rerun everything with $\sigma = 0.01$ (small noise) and gradually increasing the number of samples from $n = 10$ to $n = 100$. For small noise, the model is trying to interpolate. Is it capable of figuring out that the noise is small when the number of observations is limited? When does the method realize the noise is indeed small?

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

© Copyright 2021.