# Gaussian processes

*March 19, 2018*

Open in Colab
(https://colab.research.google.com/github/krasserm/bayesian-machine-learning/blob/dev/gaussian-processes/gaussian_processes.ipynb)

*Sources:*

- *Notebook (https://nbviewer.jupyter.org/github/krasserm/bayesian-machine-learning/blob/dev/gaussian-processes/gaussian_processes.ipynb)*
- *Repository (https://github.com/krasserm/bayesian-machine-learning)*

*Series:*

- *Gaussian processes (this article)*
- *Gaussian processes for classification (/2020/11/04/gaussian-processes-classification/)*
- *Sparse Gaussian processes (/2020/12/12/gaussian-processes-sparse/)*

## Introduction

In supervised learning, we often use parametric models $p(\mathbf{y}|\mathbf{X}, \boldsymbol{\theta})$ to explain data and infer optimal values of parameter $\boldsymbol{\theta}$ via maximum likelihood (https://en.wikipedia.org/wiki/Maximum_likelihood_estimation) or maximum a posteriori (https://de.wikipedia.org/wiki/Maximum_a_posteriori) estimation. If needed we can also infer a full posterior distribution (https://en.wikipedia.org/wiki/Posterior_probability) $p(\boldsymbol{\theta}|\mathbf{X}, \mathbf{y})$ instead of a point estimate $\hat{\boldsymbol{\theta}}$. With increasing data complexity, models with a higher number of parameters are usually needed to explain data reasonably well. Methods that use models with a fixed number of parameters are called parametric methods.

In non-parametric methods, on the other hand, the number of parameters depend on the dataset size. For example, in Nadaraya-Watson kernel regression (https://en.wikipedia.org/wiki/Kernel_regression), a weight $w_i$ is assigned to each observed target $y_i$ and for predicting the target value at a new point $\mathbf{x}$ a weighted average is computed:

$$f(\mathbf{x}) = \sum_{i=1}^{N} w_i(\mathbf{x}) y_i$$

$$w_i(\mathbf{x}) = \frac{\kappa(\mathbf{x}, \mathbf{x}_i)}{\sum_{i'=1}^{N} \kappa(\mathbf{x}, \mathbf{x}_{i'})}$$

Observations that are closer to $\mathbf{x}$ have a higher weight than observations that are further away. Weights are computed from $\mathbf{x}$ and observed $\mathbf{x}_i$ with a kernel $\kappa$. A special case is k-nearest neighbors (KNN) where the $k$ closest observations have a weight $1/k$, and all others have weight $0$. Non-parametric methods often need to process all training data for prediction and are therefore slower at inference time than parametric methods. On the other hand, training is usually faster as non-parametric models only need to remember training data.

Another example of non-parametric methods are Gaussian processes (https://en.wikipedia.org/wiki/Gaussian_process) (GPs). Instead of inferring a distribution over the parameters of a parametric function Gaussian processes can be used to infer a distribution over functions directly. A Gaussian process defines a prior over functions. After having observed some function values it can be converted into a posterior over functions. Inference of continuous function values in this context is known as GP regression but GPs can also be used for classification (/2020/11/04/gaussian-processes-classification/).

A Gaussian process is a random process (https://en.wikipedia.org/wiki/Stochastic_process) where any point $\mathbf{x} \in \mathbb{R}^d$ is assigned a random variable $f(\mathbf{x})$ and where the joint distribution of a finite number of these variables $p(f(\mathbf{x}_1), \ldots, f(\mathbf{x}_N))$ is itself Gaussian:

$$p(\mathbf{f}|\mathbf{X}) = \mathcal{N}(\mathbf{f}|\boldsymbol{\mu}, \mathbf{K}) \tag{1}$$

In Equation $(1)$, $\mathbf{f} = (f(\mathbf{x}_1), \ldots, f(\mathbf{x}_N))$, $\boldsymbol{\mu} = (m(\mathbf{x}_1), \ldots, m(\mathbf{x}_N))$ and $K_{ij} = \kappa(\mathbf{x}_i, \mathbf{x}_j)$. $m$ is the mean function and it is common to use $m(\mathbf{x}) = 0$ as GPs are flexible enough to model the mean arbitrarily well. $\kappa$ is a positive definite *kernel function* or *covariance function*. Thus, a Gaussian process is a distribution over functions whose shape (smoothness, ...) is defined by $\mathbf{K}$. If points $\mathbf{x}_i$ and $\mathbf{x}_j$ are considered to be similar by the kernel the function values at these points, $f(\mathbf{x}_i)$ and $f(\mathbf{x}_j)$, can be expected to be similar too.

Given a training dataset with noise-free function values $\mathbf{f}$ at inputs $\mathbf{X}$, a GP prior can be converted into a GP posterior $p(\mathbf{f}_*|\mathbf{X}_*, \mathbf{X}, \mathbf{f})$ which can then be used to make predictions $\mathbf{f}_*$ at new inputs $\mathbf{X}_*$. By definition of a GP, the joint distribution of observed values $\mathbf{f}$ and predictions $\mathbf{f}_*$ is again a Gaussian which can be partitioned into

$$\begin{pmatrix} \mathbf{f} \\ \mathbf{f}_* \end{pmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{pmatrix} \mathbf{K} & \mathbf{K}_* \\ \mathbf{K}_*^T & \mathbf{K}_{**} \end{pmatrix}\right) \tag{2}$$

where $\mathbf{K}_* = \kappa(\mathbf{X}, \mathbf{X}_*)$ and $\mathbf{K}_{**} = \kappa(\mathbf{X}_*, \mathbf{X}_*)$. With $N$ training data and $N_*$ new input data $\mathbf{K}$ is a $N \times N$ matrix , $\mathbf{K}_*$ a $N \times N_*$ matrix and $\mathbf{K}_{**}$ a $N_* \times N_*$ matrix. Using standard rules for conditioning Gaussians, the predictive distribution is given by

$$p(\mathbf{f}_*|\mathbf{X}_*, \mathbf{X}, \mathbf{f}) = \mathcal{N}(\mathbf{f}_*|\boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*) \tag{3}$$
$$\boldsymbol{\mu}_* = \mathbf{K}_*^T\mathbf{K}^{-1}\mathbf{f} \tag{4}$$
$$\boldsymbol{\Sigma}_* = \mathbf{K}_{**} - \mathbf{K}_*^T\mathbf{K}^{-1}\mathbf{K}_* \tag{5}$$

If we have a training dataset with noisy function values $\mathbf{y} = \mathbf{f} + \boldsymbol{\epsilon}$ where noise $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \sigma_y^2\mathbf{I})$ is independently added to each observation then the predictive distribution is given by

$$p(\mathbf{f}_*|\mathbf{X}_*, \mathbf{X}, \mathbf{y}) = \mathcal{N}(\mathbf{f}_*|\boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*) \tag{6}$$
$$\boldsymbol{\mu}_* = \mathbf{K}_*^T\mathbf{K}_y^{-1}\mathbf{y} \tag{7}$$
$$\boldsymbol{\Sigma}_* = \mathbf{K}_{**} - \mathbf{K}_*^T\mathbf{K}_y^{-1}\mathbf{K}_* \tag{8}$$

where $\mathbf{K}_y = \mathbf{K} + \sigma_y^2\mathbf{I}$. Although Equation $(6)$ covers noise in training data, it is still a distribution over noise-free predictions $\mathbf{f}_*$. To additionally include noise $\boldsymbol{\epsilon}$ into predictions $\mathbf{y}_*$ we have to add $\sigma_y^2$ to the diagonal of $\boldsymbol{\Sigma}_*$

$$p(\mathbf{y}_*|\mathbf{X}_*, \mathbf{X}, \mathbf{y}) = \mathcal{N}(\mathbf{y}_*|\boldsymbol{\mu}_*, \boldsymbol{\Sigma}_* + \sigma_y^2\mathbf{I}) \tag{9}$$

using the definitions of $\boldsymbol{\mu}_*$ and $\boldsymbol{\Sigma}_*$ from Equations $(7)$ and $(8)$, respectively. This is the minimum we need to know for implementing Gaussian processes and applying them to regression problems. For further details, please consult the literature in the References section. The next section shows how to implement GPs with plain NumPy from scratch, later sections demonstrate how to use GP implementations from scikit-learn (http://scikit-learn.org/stable/) and GPy (http://sheffieldml.github.io/GPy/).

# Implementation with NumPy

Here, we will use the squared exponential kernel, also known as Gaussian kernel or RBF kernel:

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \sigma_f^2 \exp\left(-\frac{1}{2l^2}(\mathbf{x}_i - \mathbf{x}_j)^T(\mathbf{x}_i - \mathbf{x}_j)\right) \tag{10}$$

The length parameter $l$ controls the smoothness of the function and $\sigma_f$ the vertical variation. For simplicity, we use the same length parameter $l$ for all input dimensions (isotropic kernel).

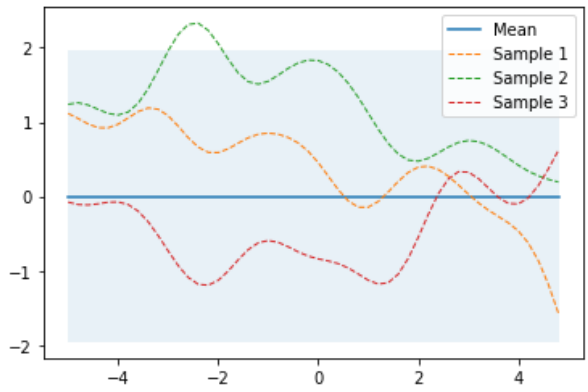
```
import numpy as np

def kernel(X1, X2, l=1.0, sigma_f=1.0):
    """
    Isotropic squared exponential kernel.

    Args:
        X1: Array of m points (m x d).
        X2: Array of n points (n x d).

    Returns:
        (m x n) matrix.
    """
    sqdist = np.sum(X1**2, 1).reshape(-1, 1) + np.sum(X2**2, 1) - 2 * np.dot(X1, X2.T)
    return sigma_f**2 * np.exp(-0.5 / l**2 * sqdist)
```

There are many other kernels that can be used for Gaussian processes. See [3] for a detailed reference or the scikit-learn documentation for some examples (http://scikit-learn.org/stable/modules/gaussian_process.html#gp-kernels).

## Prior

Let's first define a prior over functions with mean zero and a covariance matrix computed with kernel parameters $l = 1$ and $\sigma_f = 1$. To draw random functions from that GP we draw random samples from the corresponding multivariate normal. The following example draws three random samples and plots it together with the zero mean and the *uncertainty region* (the 95% highest density interval computed from the diagonal of the covariance matrix).

```
%matplotlib inline

from gaussian_processes_util import plot_gp

# Finite number of points
X = np.arange(-5, 5, 0.2).reshape(-1, 1)

# Mean and covariance of the prior
mu = np.zeros(X.shape)
cov = kernel(X, X)

# Draw three samples from the prior
samples = np.random.multivariate_normal(mu.ravel(), cov, 3)

# Plot GP mean, uncertainty region and samples
plot_gp(mu, cov, X, samples=samples)
```




The `plot_gp` function is defined here (https://github.com/krasserm/bayesian-machine-learning/blob/dev/gaussian-processes/gaussian_processes_util.py).

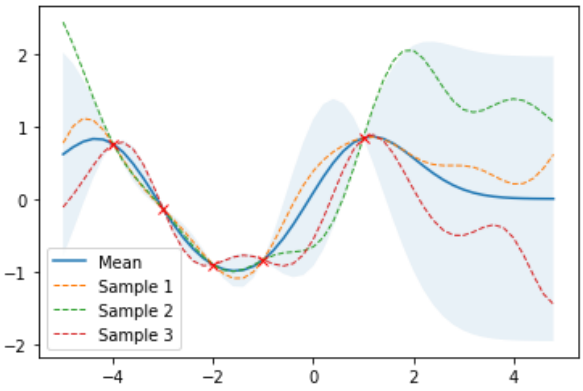
## Prediction from noise-free training data

To compute the sufficient statistics i.e. mean and covariance of the posterior we implement Equations $(7)$ and $(8)$.

```
from numpy.linalg import inv

def posterior(X_s, X_train, Y_train, l=1.0, sigma_f=1.0, sigma_y=1e-8):
    """
    Computes the suffifient statistics of the posterior distribution
    from m training data X_train and Y_train and n new inputs X_s.

    Args:
        X_s: New input locations (n x d).
        X_train: Training locations (m x d).
        Y_train: Training targets (m x 1).
        l: Kernel length parameter.
        sigma_f: Kernel vertical variation parameter.
        sigma_y: Noise parameter.

    Returns:
        Posterior mean vector (n x d) and covariance matrix (n x n).
    """
    K = kernel(X_train, X_train, l, sigma_f) + sigma_y**2 * np.eye(len(X_train))
    K_s = kernel(X_train, X_s, l, sigma_f)
    K_ss = kernel(X_s, X_s, l, sigma_f) + 1e-8 * np.eye(len(X_s))
    K_inv = inv(K)

    # Equation (7)
    mu_s = K_s.T.dot(K_inv).dot(Y_train)

    # Equation (8)
    cov_s = K_ss - K_s.T.dot(K_inv).dot(K_s)

    return mu_s, cov_s
```

and apply them to noise-free training data `X_train` and `Y_train`. The following example draws three samples from the posterior and plots them along with the mean, uncertainty region and training data. In a noise-free model, variance at the training points is zero and all random functions drawn from the posterior go through the trainig points.

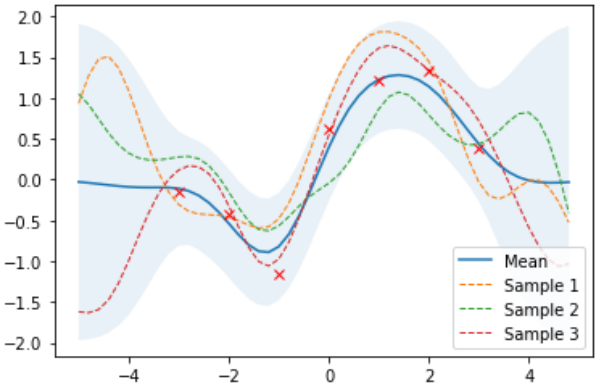
```
# Noise free training data
X_train = np.array([-4, -3, -2, -1, 1]).reshape(-1, 1)
Y_train = np.sin(X_train)

# Compute mean and covariance of the posterior distribution
mu_s, cov_s = posterior(X, X_train, Y_train)

samples = np.random.multivariate_normal(mu_s.ravel(), cov_s, 3)
plot_gp(mu_s, cov_s, X, X_train=X_train, Y_train=Y_train, samples=samples)
```




## Prediction from noisy training data

If some noise is included in the model, training points are only approximated and the variance at the training points is non-zero.

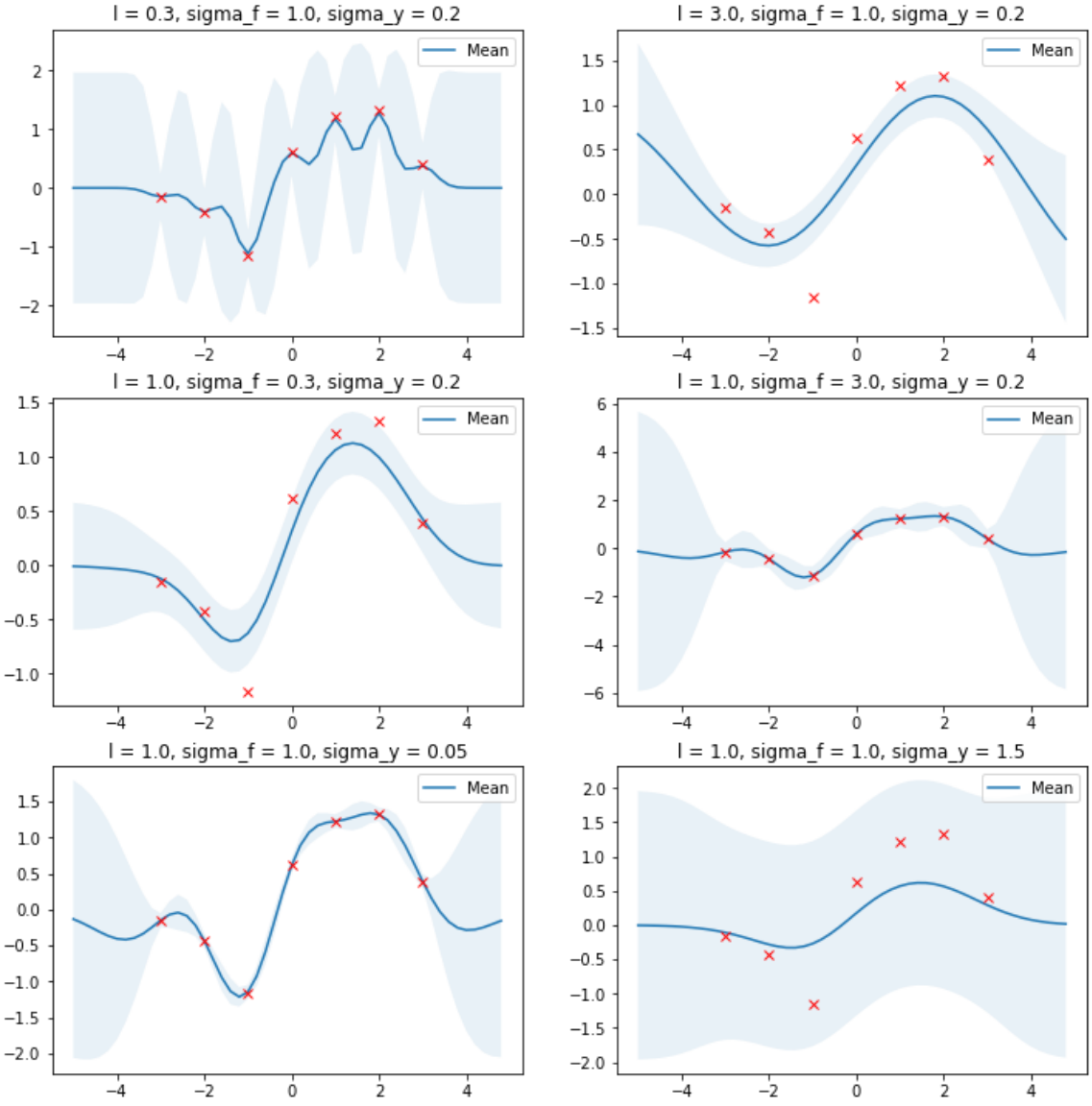
```
noise = 0.4

# Noisy training data
X_train = np.arange(-3, 4, 1).reshape(-1, 1)
Y_train = np.sin(X_train) + noise * np.random.randn(*X_train.shape)

# Compute mean and covariance of the posterior distribution
mu_s, cov_s = posterior(X, X_train, Y_train, sigma_y=noise)

samples = np.random.multivariate_normal(mu_s.ravel(), cov_s, 3)
plot_gp(mu_s, cov_s, X, X_train=X_train, Y_train=Y_train, samples=samples)
```




## Effect of kernel parameters and noise parameter

The following example shows the effect of kernel parameters $l$ and $\sigma_f$ as well as the noise parameter $\sigma_y$. Higher $l$ values lead to smoother functions and therefore to coarser approximations of the training data. Lower $l$ values make functions more wiggly with wide uncertainty regions between training data points. $\sigma_f$ controls the vertical variation of functions drawn from the GP. This can be seen by the wide uncertainty regions outside the training data region in the right figure of the second row. $\sigma_y$ represents the amount of noise in the training data. Higher $\sigma_y$ values make more coarse approximations which avoids overfitting to noisy data.

```
import matplotlib.pyplot as plt

params = [
    (0.3, 1.0, 0.2),
    (3.0, 1.0, 0.2),
    (1.0, 0.3, 0.2),
    (1.0, 3.0, 0.2),
    (1.0, 1.0, 0.05),
    (1.0, 1.0, 1.5),
]

plt.figure(figsize=(12, 5))

for i, (l, sigma_f, sigma_y) in enumerate(params):
    mu_s, cov_s = posterior(X, X_train, Y_train, l=l,
                            sigma_f=sigma_f,
                            sigma_y=sigma_y)
    plt.subplot(3, 2, i + 1)
    plt.subplots_adjust(top=2)
    plt.title(f'l = {l}, sigma_f = {sigma_f}, sigma_y = {sigma_y}')
    plot_gp(mu_s, cov_s, X, X_train=X_train, Y_train=Y_train)
```

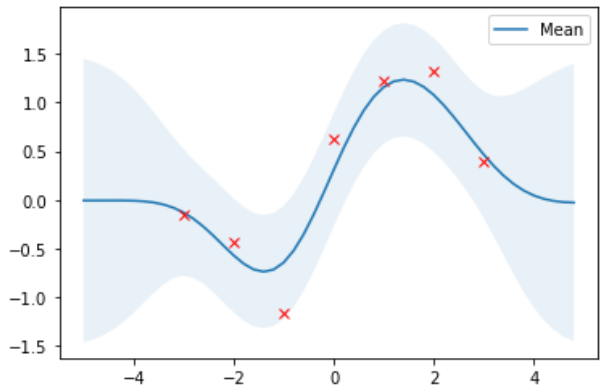



Optimal values for these parameters can be estimated by maximizing the log marginal likelihood which is given by[1][3]

$$\log p(\mathbf{y}|\mathbf{X}) = \log \mathcal{N}(\mathbf{y}|\mathbf{0}, \mathbf{K}_y) = -\frac{1}{2}\mathbf{y}^T\mathbf{K}_y^{-1}\mathbf{y} - \frac{1}{2}\log|\mathbf{K}_y| - \frac{N}{2}\log(2\pi) \quad (11)$$

In the following we will minimize the negative log marginal likelihood w.r.t. parameters $l$ and $\sigma_f$, $\sigma_y$ is set to the known noise level of the data. If the noise level is unknown, $\sigma_y$ can be estimated as well along with the other parameters.

```
from numpy.linalg import cholesky, det
from scipy.linalg import solve_triangular
from scipy.optimize import minimize

def nll_fn(X_train, Y_train, noise, naive=True):
    """
    Returns a function that computes the negative log marginal
    likelihood for training data X_train and Y_train and given
    noise level.

    Args:
        X_train: training locations (m x d).
        Y_train: training targets (m x 1).
        noise: known noise level of Y_train.
        naive: if True use a naive implementation of Eq. (11), if
               False use a numerically more stable implementation.

    Returns:
        Minimization objective.
    """

    Y_train = Y_train.ravel()

    def nll_naive(theta):
        # Naive implementation of Eq. (11). Works well for the examples
        # in this article but is numerically less stable compared to
        # the implementation in nll_stable below.
        K = kernel(X_train, X_train, l=theta[0], sigma_f=theta[1]) + \
            noise**2 * np.eye(len(X_train))
        return 0.5 * np.log(det(K)) + \
               0.5 * Y_train.dot(inv(K).dot(Y_train)) + \
               0.5 * len(X_train) * np.log(2*np.pi)

    def nll_stable(theta):
        # Numerically more stable implementation of Eq. (11) as described
        # in http://www.gaussianprocess.org/gpml/chapters/RW2.pdf, Section
        # 2.2, Algorithm 2.1.

        K = kernel(X_train, X_train, l=theta[0], sigma_f=theta[1]) + \
            noise**2 * np.eye(len(X_train))
        L = cholesky(K)

        S1 = solve_triangular(L, Y_train, lower=True)
        S2 = solve_triangular(L.T, S1, lower=False)

        return np.sum(np.log(np.diagonal(L))) + \
               0.5 * Y_train.dot(S2) + \
               0.5 * len(X_train) * np.log(2*np.pi)

    if naive:
        return nll_naive
    else:
        return nll_stable

# Minimize the negative log-likelihood w.r.t. parameters l and sigma_f.
# We should actually run the minimization several times with different
# initializations to avoid local minima but this is skipped here for
# simplicity.
res = minimize(nll_fn(X_train, Y_train, noise), [1, 1],
               bounds=((1e-5, None), (1e-5, None)),
               method='L-BFGS-B')

# Store the optimization results in global variables so that we can
# compare it later with the results from other implementations.
l_opt, sigma_f_opt = res.x

# Compute posterior mean and covariance with optimized kernel parameters and plot the results
mu_s, cov_s = posterior(X, X_train, Y_train, l=l_opt, sigma_f=sigma_f_opt, sigma_y=noise)
plot_gp(mu_s, cov_s, X, X_train=X_train, Y_train=Y_train)
```

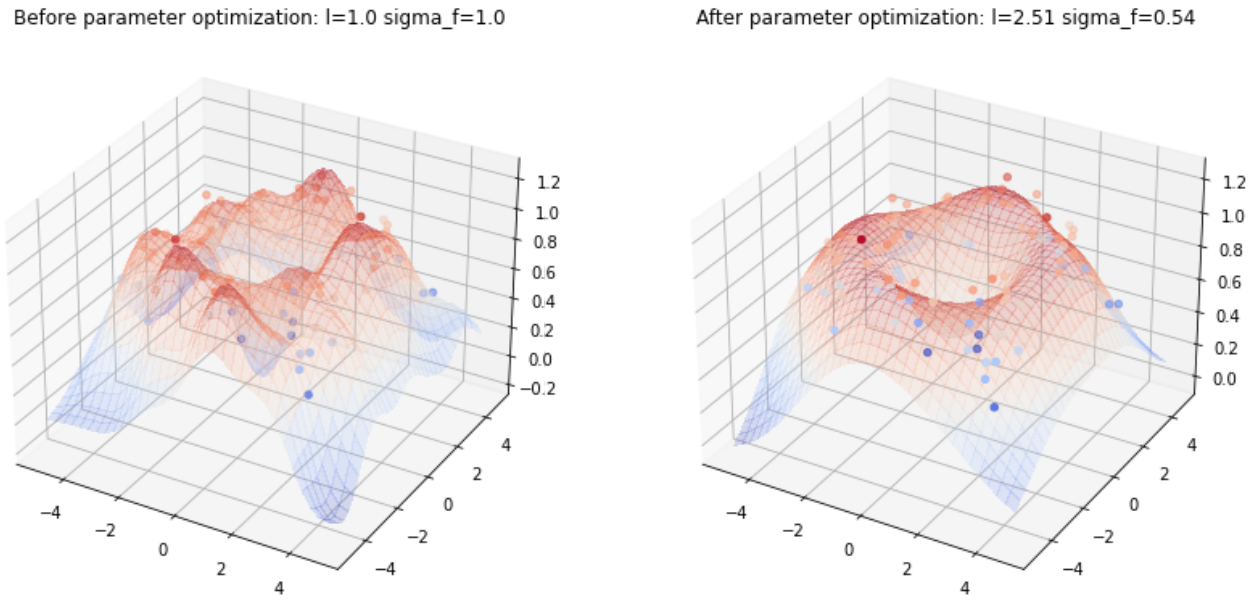



With optimized kernel parameters, training data are reasonably covered by the 95% highest density interval and the mean of the posterior is a good approximation.

## Higher dimensions

The above implementation can also be used for higher input data dimensions. Here, a GP is used to fit noisy samples from a sine wave originating at $\mathbf{0}$ and expanding in the x-y plane. The following plots show the noisy samples and the posterior mean before and after kernel parameter optimization.

```
from gaussian_processes_util import plot_gp_2D

noise_2D = 0.1

rx, ry = np.arange(-5, 5, 0.3), np.arange(-5, 5, 0.3)
gx, gy = np.meshgrid(rx, rx)

X_2D = np.c_[gx.ravel(), gy.ravel()]

X_2D_train = np.random.uniform(-4, 4, (100, 2))
Y_2D_train = np.sin(0.5 * np.linalg.norm(X_2D_train, axis=1)) + \
             noise_2D * np.random.randn(len(X_2D_train))

plt.figure(figsize=(14,7))

mu_s, _ = posterior(X_2D, X_2D_train, Y_2D_train, sigma_y=noise_2D)
plot_gp_2D(gx, gy, mu_s, X_2D_train, Y_2D_train,
           f'Before parameter optimization: l={1.00} sigma_f={1.00}', 1)

res = minimize(nll_fn(X_2D_train, Y_2D_train, noise_2D), [1, 1],
               bounds=((1e-5, None), (1e-5, None)),
               method='L-BFGS-B')

mu_s, _ = posterior(X_2D, X_2D_train, Y_2D_train, *res.x, sigma_y=noise_2D)
plot_gp_2D(gx, gy, mu_s, X_2D_train, Y_2D_train,
           f'After parameter optimization: l={res.x[0]:.2f} sigma_f={res.x[1]:.2f}', 2)
```




Note how the true sine wave is approximated much better after parameter optimization.

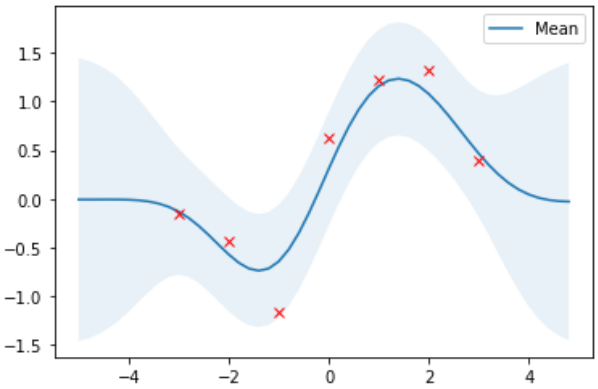
# Libraries that implement GPs

This section shows two examples of libraries that provide implementations of GPs. I'll provide only a minimal setup here, just enough for reproducing the above results. For further details please consult the documentation of these libraries.

## Scikit-learn

Scikit-learn provides a `GaussianProcessRegressor` for implementing GP regression models (http://scikit-learn.org/stable/modules/gaussian_process.html#gaussian-process-regression-gpr). It can be configured with pre-defined kernels and user-defined kernels (http://scikit-learn.org/stable/modules/gaussian_process.html#gp-kernels). Kernels can also be composed. The squared exponential kernel is the `RBF` kernel in scikit-learn. The `RBF` kernel only has a `length_scale` parameter which corresponds to the $l$ parameter above. To have a $\sigma_f$ parameter as well, we have to compose the `RBF` kernel with a `ConstantKernel`.

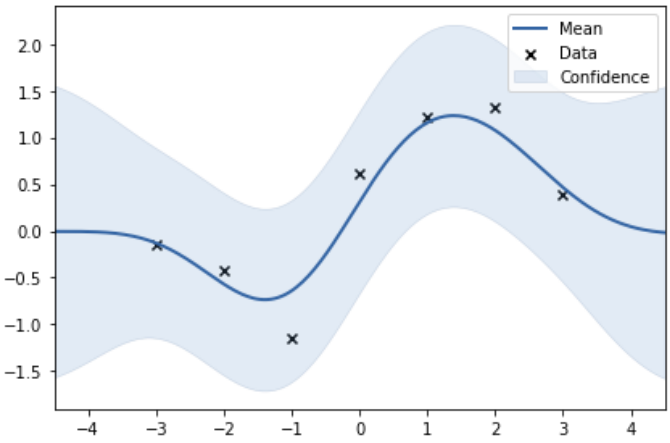
```
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import ConstantKernel, RBF

rbf = ConstantKernel(1.0) * RBF(length_scale=1.0)
gpr = GaussianProcessRegressor(kernel=rbf, alpha=noise**2)

# Reuse training data from previous 1D example
gpr.fit(X_train, Y_train)

# Compute posterior mean and covariance
mu_s, cov_s = gpr.predict(X, return_cov=True)

# Obtain optimized kernel parameters
l = gpr.kernel_.k2.get_params()['length_scale']
sigma_f = np.sqrt(gpr.kernel_.k1.get_params()['constant_value'])

# Compare with previous results
assert(np.isclose(l_opt, l))
assert(np.isclose(sigma_f_opt, sigma_f))

# Plot the results
plot_gp(mu_s, cov_s, X, X_train=X_train, Y_train=Y_train)
```




## GPy

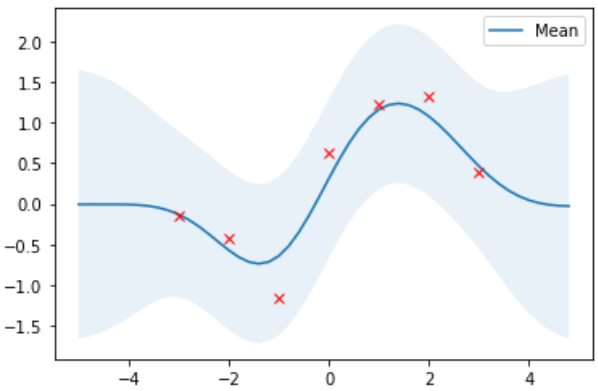
GPy (http://sheffieldml.github.io/GPy/) is a Gaussian processes framework from the Sheffield machine learning group. It provides a `GPRegression` class for implementing GP regression models. By default, `GPRegression` also estimates the noise parameter $\sigma_y$ from data, so we have to `fix()` this parameter to be able to reproduce the above results.

```
import GPy

rbf = GPy.kern.RBF(input_dim=1, variance=1.0, lengthscale=1.0)
gpr = GPy.models.GPRegression(X_train, Y_train, rbf)

# Fix the noise variance to known value
gpr.Gaussian_noise.variance = noise**2
gpr.Gaussian_noise.variance.fix()

# Run optimization
gpr.optimize();

# Obtain optimized kernel parameters
l = gpr.rbf.lengthscale.values[0]
sigma_f = np.sqrt(gpr.rbf.variance.values[0])

# Compare with previous results
assert(np.isclose(l_opt, l))
assert(np.isclose(sigma_f_opt, sigma_f))

# Plot the results with the built-in plot function
gpr.plot();
```




Although we can reproduce the kernel parameter optimization results, the variances of the predictions are still higher compared to previous results. This is because `GPy` includes noise variance $\sigma_y^2$ into predictions. This can be easily reproduced using Equation $(9)$

```
mu_s, cov_s = posterior(X, X_train, Y_train, l=l_opt, sigma_f=sigma_f_opt, sigma_y=noise)

# Include noise into predictions using Equation (9).
cov_s = cov_s + noise**2 * np.eye(*cov_s.shape)

plot_gp(mu_s, cov_s, X, X_train=X_train, Y_train=Y_train)
```



Thanks for reading up to here :-) In another article, I'll show how Gaussian processes can be used for black-box optimization.

# References

[1] Kevin P. Murphy. Machine Learning, A Probabilistic Perspective (https://mitpress.mit.edu/books/machine-learning-0), Chapters 4, 14 and 15.
[2] Christopher M. Bishop. Pattern Recognition and Machine Learning (http://www.springer.com/de/book/9780387310732), Chapter 6.
[3] Carl Edward Rasmussen and Christopher K. I. Williams. Gaussian Processes for Machine Learning (http://www.gaussianprocess.org/gpml/).

← PREVIOUS POST (/2018/02/07/DEEP-FACE-RECOGNITION/)

NEXT POST → (/2018/03/21/BAYESIAN-OPTIMIZATION/)

(http://martin-krasser.com/resume/resume.pdf) (https://twitter.com/mrt1nz)

(https://github.com/krasserm) (https://linkedin.com/in/krasserm)

(/atom.xml)