1.7. Gaussian Processes

Gaussian Processes (GP) are a generic supervised learning method designed to solve *regression* and *probabilistic classification* problems.

The advantages of Gaussian processes are:

- The prediction interpolates the observations (at least for regular kernels).
- The prediction is probabilistic (Gaussian) so that one can compute empirical confidence intervals and decide based on those if one should refit (online fitting, adaptive fitting) the prediction in some region of interest.
- Versatile: different kernels can be specified. Common kernels are provided, but it is also possible to specify custom kernels.

The disadvantages of Gaussian processes include:

- They are not sparse, i.e., they use the whole samples/features information to perform the prediction.
- They lose efficiency in high dimensional spaces – namely when the number of features exceeds a few dozens.

1.7.1. Gaussian Process Regression (GPR)

The `GaussianProcessRegressor` implements Gaussian processes (GP) for regression purposes. For this, the prior of the GP needs to be specified. The prior mean is assumed to be constant and zero (for `normalize_y=False`) or the training data's mean (for `normalize_y=True`). The prior's covariance is specified by passing a kernel object. The hyperparameters of the kernel are optimized during fitting of GaussianProcessRegressor by maximizing the log-marginal-likelihood (LML) based on the passed `optimizer`. As the LML may have multiple local optima, the optimizer can be started repeatedly by specifying `n_restarts_optimizer`. The first run is always conducted starting from the initial hyperparameter values of the kernel; subsequent runs are conducted from hyperparameter values that have been chosen randomly from the range of allowed values. If the initial hyperparameters should be kept fixed, `None` can be passed as optimizer.

The noise level in the targets can be specified by passing it via the parameter `alpha`, either globally as a scalar or per datapoint. Note that a moderate noise level can also be helpful for dealing with numeric issues during fitting as it is effectively implemented as Tikhonov regularization, i.e., by adding it to the diagonal of the kernel matrix. An alternative to specifying the noise level explicitly is to include a WhiteKernel component into the kernel, which can estimate the global noise level from the data (see example below).

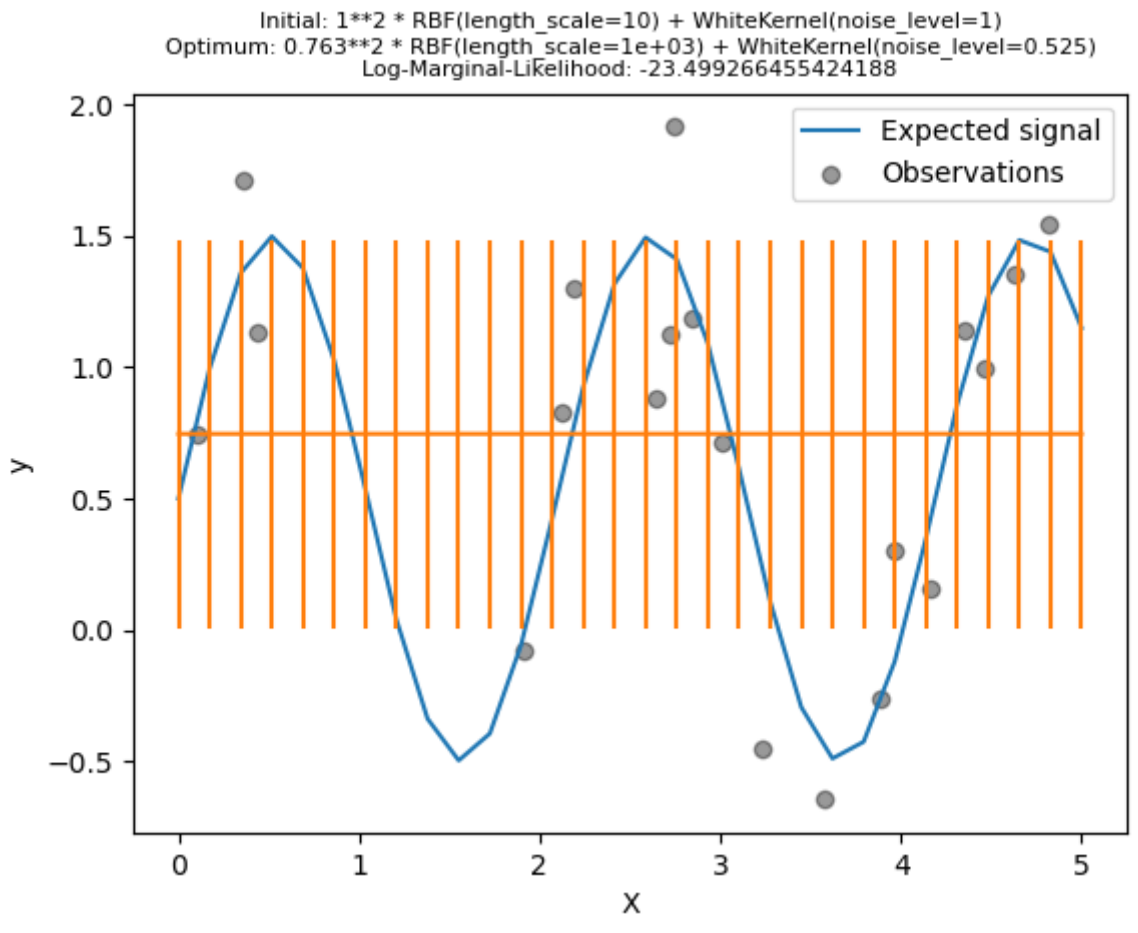
The implementation is based on Algorithm 2.1 of [RW2006]. In addition to the API of standard scikit-learn estimators, GaussianProcessRegressor:

- allows prediction without prior fitting (based on the GP prior)
- provides an additional method `sample_y(X)`, which evaluates samples drawn from the GPR (prior or posterior) at given inputs
- exposes a method `log_marginal_likelihood(theta)`, which can be used externally for other ways of selecting hyperparameters, e.g., via Markov chain Monte Carlo.

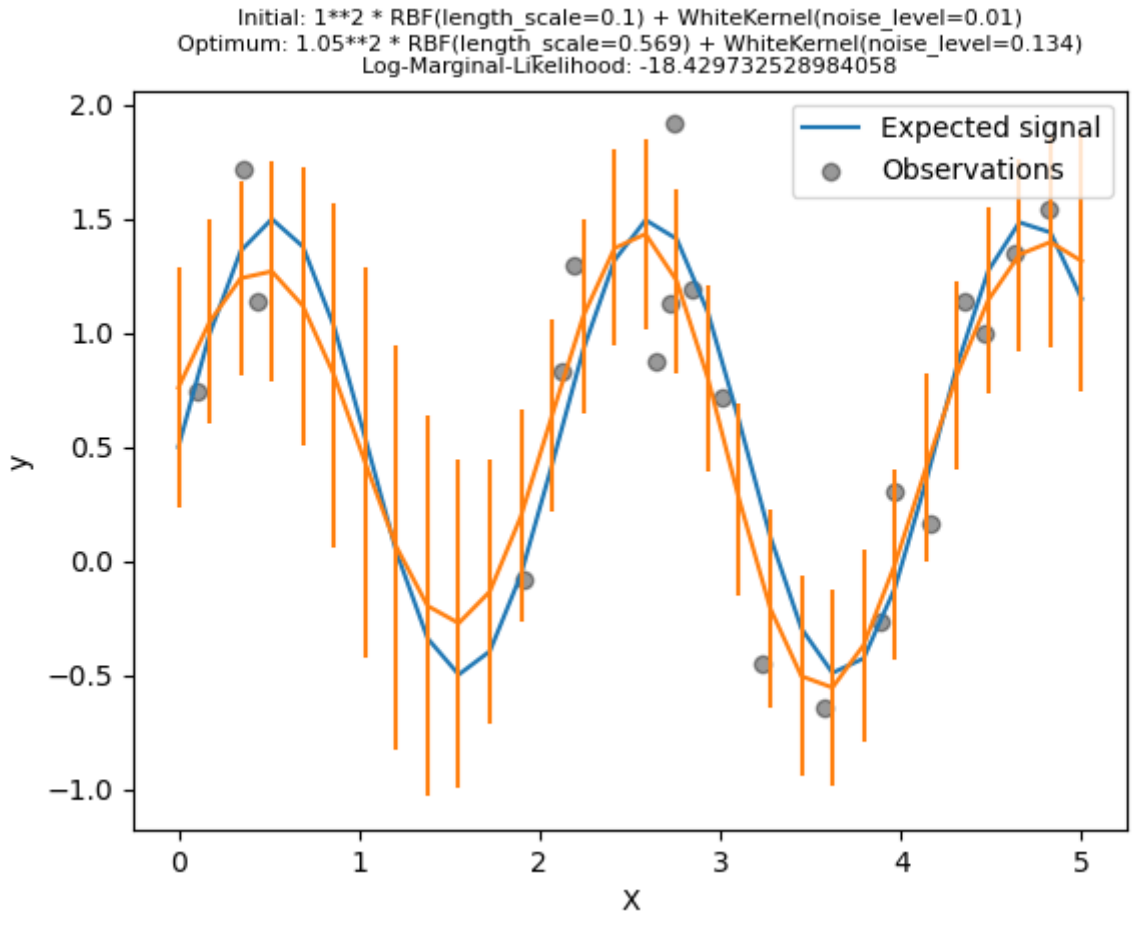
1.7.2. GPR examples

1.7.2.1. GPR with noise-level estimation

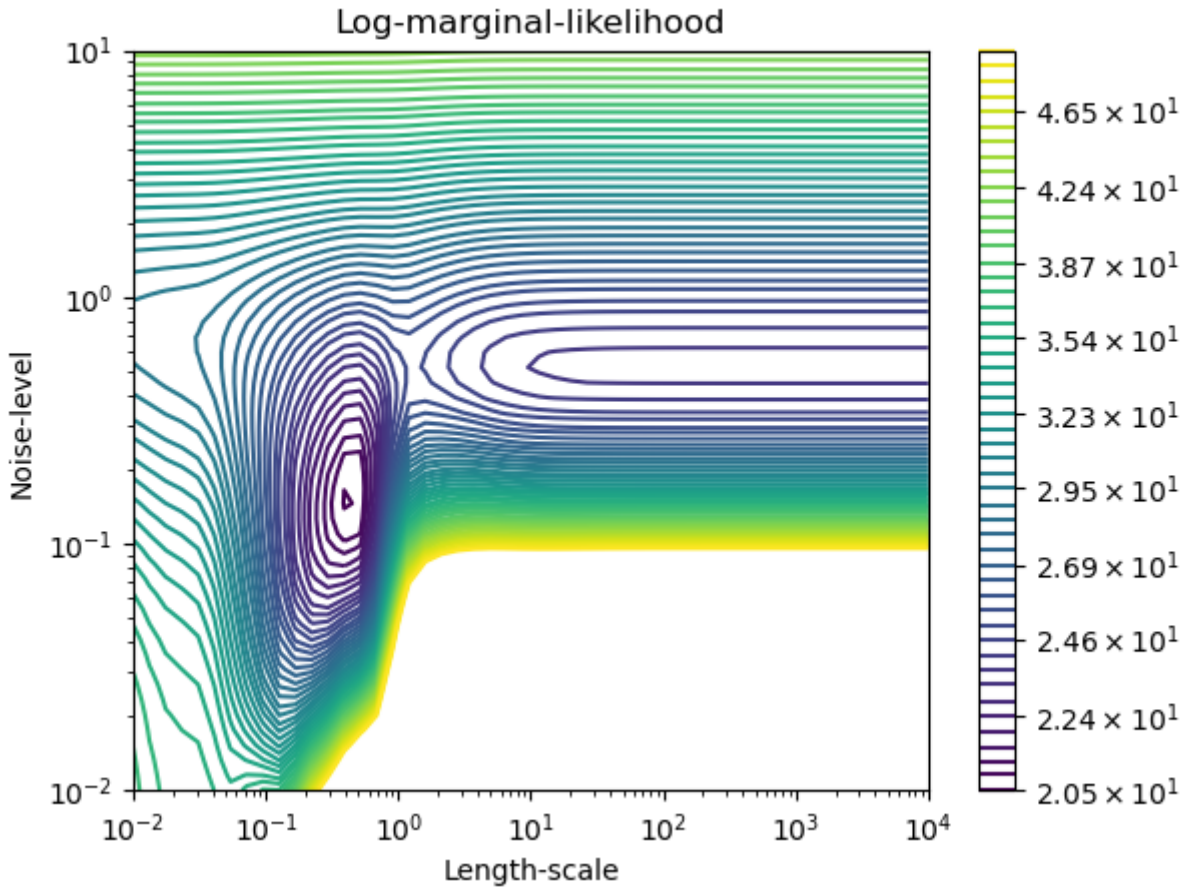
This example illustrates that GPR with a sum-kernel including a WhiteKernel can estimate the noise level of data. An illustration of the log-marginal-likelihood (LML) landscape shows that there exist two local maxima of LML.



The first corresponds to a model with a high noise level and a large length scale, which explains all variations in the data by noise.



The second one has a smaller noise level and shorter length scale, which explains most of the variation by the noise-free functional relationship. The second model has a higher likelihood; however, depending on the initial value for the hyperparameters, the gradient-based optimization might also converge to the high-noise solution. It is thus important to repeat the optimization several times for different initializations.

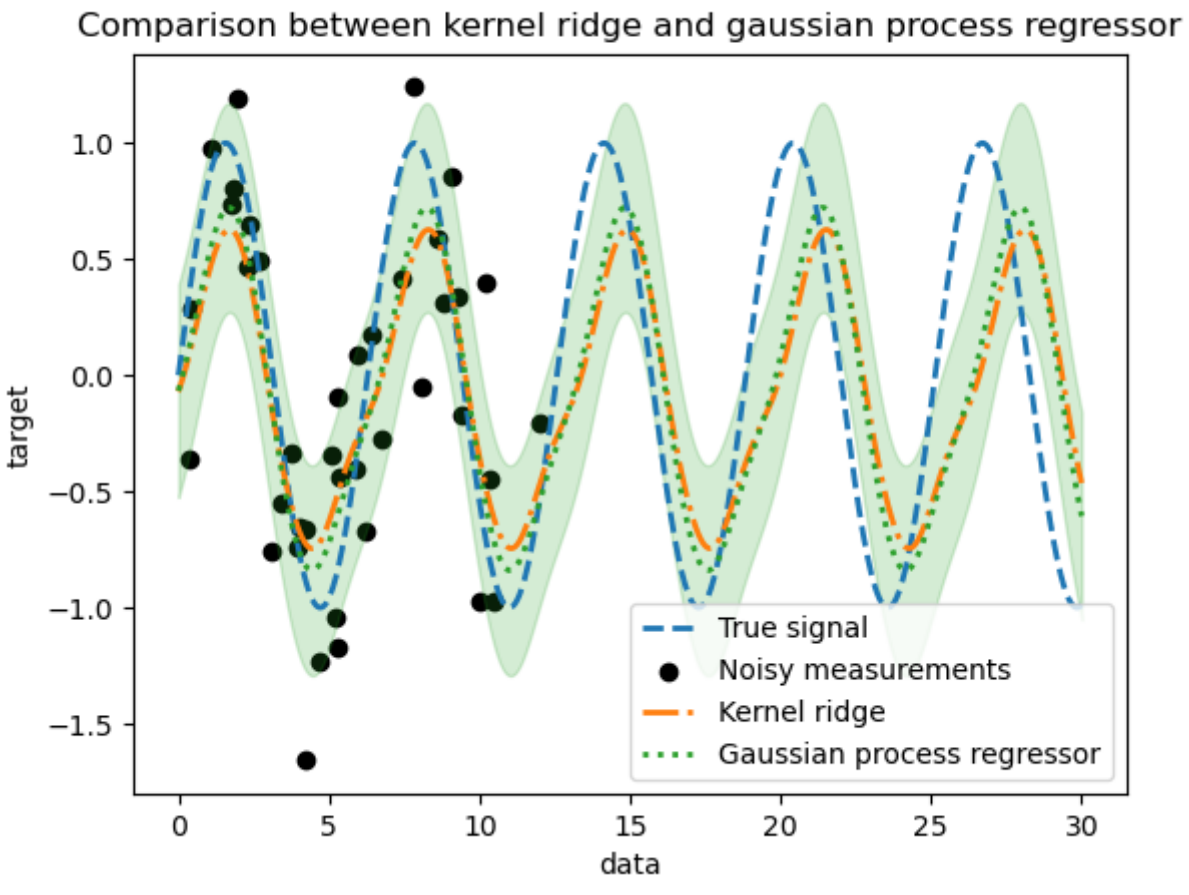


1.7.2.2. Comparison of GPR and Kernel Ridge Regression

Both kernel ridge regression (KRR) and GPR learn a target function by employing internally the “kernel trick”. KRR learns a linear function in the space induced by the respective kernel which corresponds to a non-linear function in the original space. The linear function in the kernel space is chosen based on the mean-squared error loss with ridge regularization. GPR uses the kernel to define the covariance of a prior distribution over the target functions and uses the observed training data to define a likelihood function. Based on Bayes theorem, a (Gaussian) posterior distribution over target functions is defined, whose mean is used for prediction.

A major difference is that GPR can choose the kernel’s hyperparameters based on gradient-ascent on the marginal likelihood function while KRR needs to perform a grid search on a cross-validated loss function (mean-squared error loss). A further difference is that GPR learns a generative, probabilistic model of the target function and can thus provide meaningful confidence intervals and posterior samples along with the predictions while KRR only provides predictions.

The following figure illustrates both methods on an artificial dataset, which consists of a sinusoidal target function and strong noise. The figure compares the learned model of KRR and GPR based on a ExpSineSquared kernel, which is suited for learning periodic functions. The kernel’s hyperparameters control the smoothness (length_scale) and periodicity of the kernel (periodicity). Moreover, the noise level of the data is learned explicitly by GPR by an additional WhiteKernel component in the kernel and by the regularization parameter alpha of KRR.



The figure shows that both methods learn reasonable models of the target function. GPR provides reasonable confidence bounds on the prediction which are not available for KRR. A major difference between the two methods is the time required for fitting and predicting: while fitting KRR is fast in principle, the grid-search for hyperparameter optimization scales exponentially with the number of hyperparameters (“curse of dimensionality”).

The gradient-based optimization of the parameters in GPR does not suffer from this exponential scaling and is thus considerably faster on this example with 3-dimensional hyperparameter space. The time for predicting is similar; however, generating the variance of the predictive distribution of GPR takes considerably longer than just predicting the mean.

1.7.2.3. GPR on Mauna Loa CO2 data

This example is based on Section 5.4.3 of [RW2006]. It illustrates an example of complex kernel engineering and hyperparameter optimization using gradient ascent on the log-marginal-likelihood. The data consists of the monthly average atmospheric CO2 concentrations (in parts per million by volume (ppmv)) collected at the Mauna Loa Observatory in Hawaii, between 1958 and 1997. The objective is to model the CO2 concentration as a function of the time t.

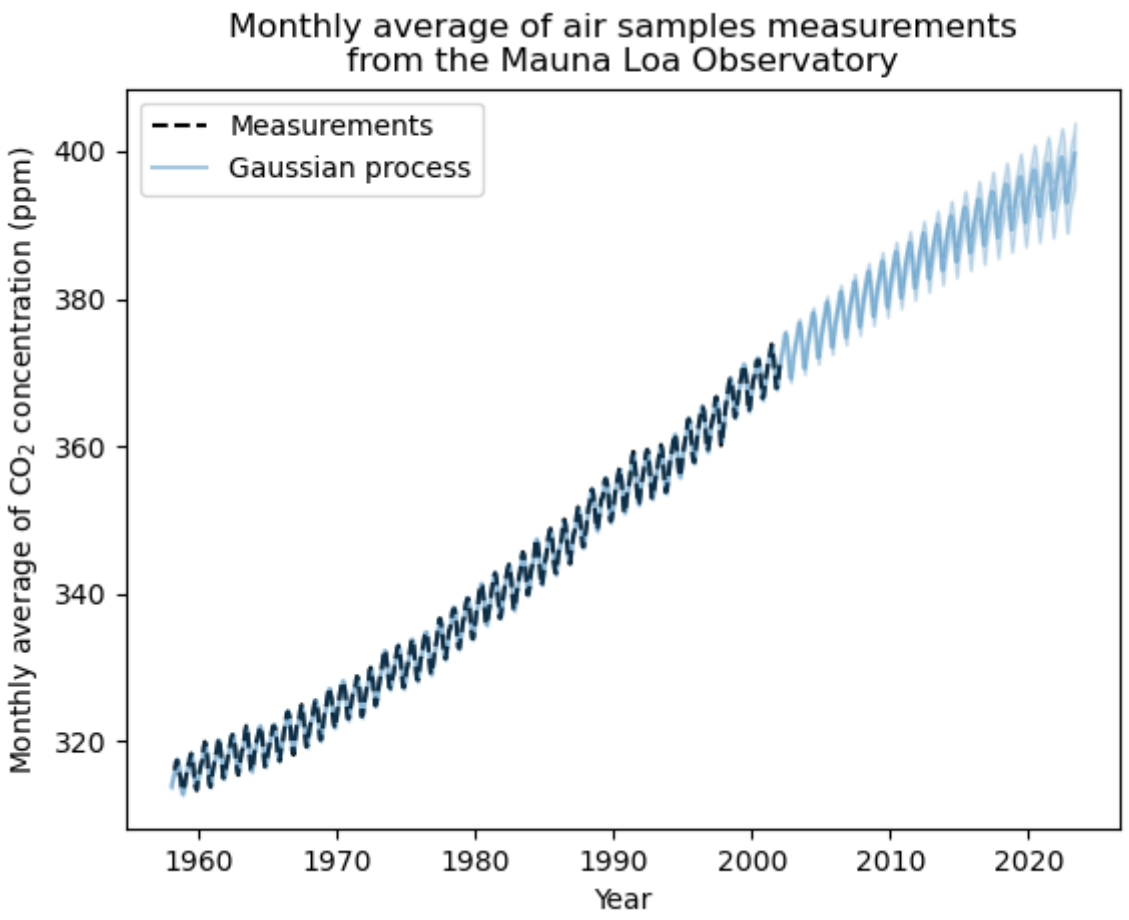
The kernel is composed of several terms that are responsible for explaining different properties of the signal:

- a long term, smooth rising trend is to be explained by an RBF kernel. The RBF kernel with a large length-scale enforces this component to be smooth; it is not enforced that the trend is rising which leaves this choice to the GP. The specific length-scale and the amplitude are free hyperparameters.
- a seasonal component, which is to be explained by the periodic ExpSineSquared kernel with a fixed periodicity of 1 year. The length-scale of this periodic component, controlling its smoothness, is a free parameter. In order to allow decaying away from exact periodicity, the product with an RBF kernel is taken. The length-scale of this RBF component controls the decay time and is a further free parameter.
- smaller, medium term irregularities are to be explained by a RationalQuadratic kernel component, whose length-scale and alpha parameter, which determines the diffuseness of the length-scales, are to be determined. According to [RW2006], these irregularities can better be explained by a RationalQuadratic than an RBF kernel component, probably because it can accommodate several length-scales.
- a “noise” term, consisting of an RBF kernel contribution, which shall explain the correlated noise components such as local weather phenomena, and a WhiteKernel contribution for the white noise. The relative amplitudes and the RBF’s length scale are further free parameters.

Maximizing the log-marginal-likelihood after subtracting the target’s mean yields the following kernel with an LML of -83.214:

```
34.4**2 * RBF(length_scale=41.8)
+ 3.27**2 * RBF(length_scale=180) * ExpSineSquared(length_scale=1.44,
                                                   periodicity=1)
+ 0.446**2 * RationalQuadratic(alpha=17.7, length_scale=0.957)
+ 0.197**2 * RBF(length_scale=0.138) + WhiteKernel(noise_level=0.0336)
```

Thus, most of the target signal (34.4ppm) is explained by a long-term rising trend (length-scale 41.8 years). The periodic component has an amplitude of 3.27ppm, a decay time of 180 years and a length-scale of 1.44. The long decay time indicates that we have a locally very close to periodic seasonal component. The correlated noise has an amplitude of 0.197ppm with a length scale of 0.138 years and a white-noise contribution of 0.197ppm. Thus, the overall noise level is very small, indicating that the data can be very well explained by the model. The figure shows also that the model makes very confident predictions until around 2015



1.7.3. Gaussian Process Classification (GPC)

The `GaussianProcessClassifier` implements Gaussian processes (GP) for classification purposes, more specifically for probabilistic classification, where test predictions take the form of class probabilities. GaussianProcessClassifier places a GP prior on a latent function f, which is then squashed through a link function to obtain the probabilistic classification. The latent function f is a so-called nuisance function, whose values are

not observed and are not relevant by themselves. Its purpose is to allow a convenient formulation of the model, and f is removed (integrated out) during prediction. GaussianProcessClassifier implements the logistic link function, for which the integral cannot be computed analytically but is easily approximated in the binary case.

In contrast to the regression setting, the posterior of the latent function f is not Gaussian even for a GP prior since a Gaussian likelihood is inappropriate for discrete class labels. Rather, a non-Gaussian likelihood corresponding to the logistic link function (logit) is used. GaussianProcessClassifier approximates the non-Gaussian posterior with a Gaussian based on the Laplace approximation. More details can be found in Chapter 3 of [RW2006].

The GP prior mean is assumed to be zero. The prior's covariance is specified by passing a kernel object. The hyperparameters of the kernel are optimized during fitting of GaussianProcessRegressor by maximizing the log-marginal-likelihood (LML) based on the passed `optimizer`. As the LML may have multiple local optima, the optimizer can be started repeatedly by specifying `n_restarts_optimizer`. The first run is always conducted starting from the initial hyperparameter values of the kernel; subsequent runs are conducted from hyperparameter values that have been chosen randomly from the range of allowed values. If the initial hyperparameters should be kept fixed, `None` can be passed as optimizer.

`GaussianProcessClassifier` supports multi-class classification by performing either one-versus-rest or one-versus-one based training and prediction. In one-versus-rest, one binary Gaussian process classifier is fitted for each class, which is trained to separate this class from the rest. In "one_vs_one", one binary Gaussian process classifier is fitted for each pair of classes, which is trained to separate these two classes. The predictions of these binary predictors are combined into multi-class predictions. See the section on multi-class classification for more details.

In the case of Gaussian process classification, "one_vs_one" might be computationally cheaper since it has to solve many problems involving only a subset of the whole training set rather than fewer problems on the whole dataset. Since Gaussian process classification scales cubically with the size of the dataset, this might be considerably faster. However, note that "one_vs_one" does not support predicting probability estimates but only plain predictions. Moreover, note that `GaussianProcessClassifier` does not (yet) implement a true multi-class Laplace approximation internally, but as discussed above is based on solving several binary classification tasks internally, which are combined using one-versus-rest or one-versus-one.

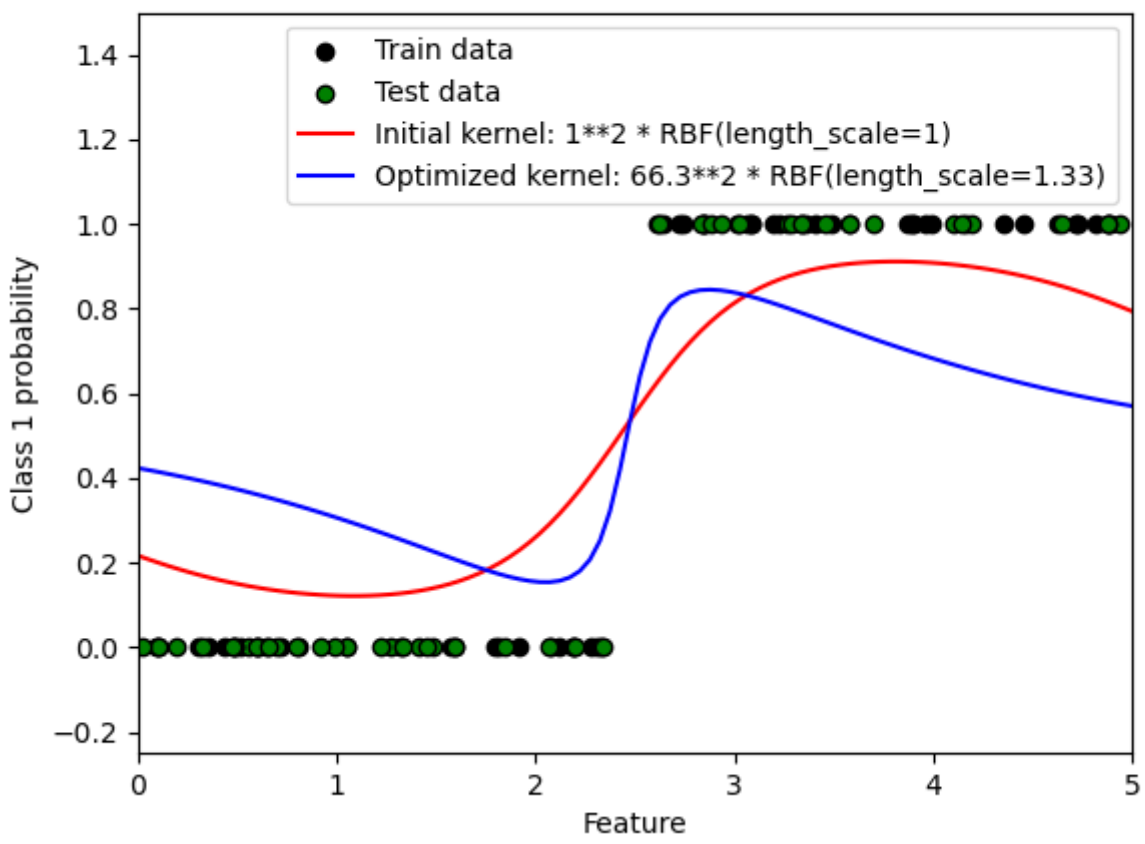
1.7.4. GPC examples

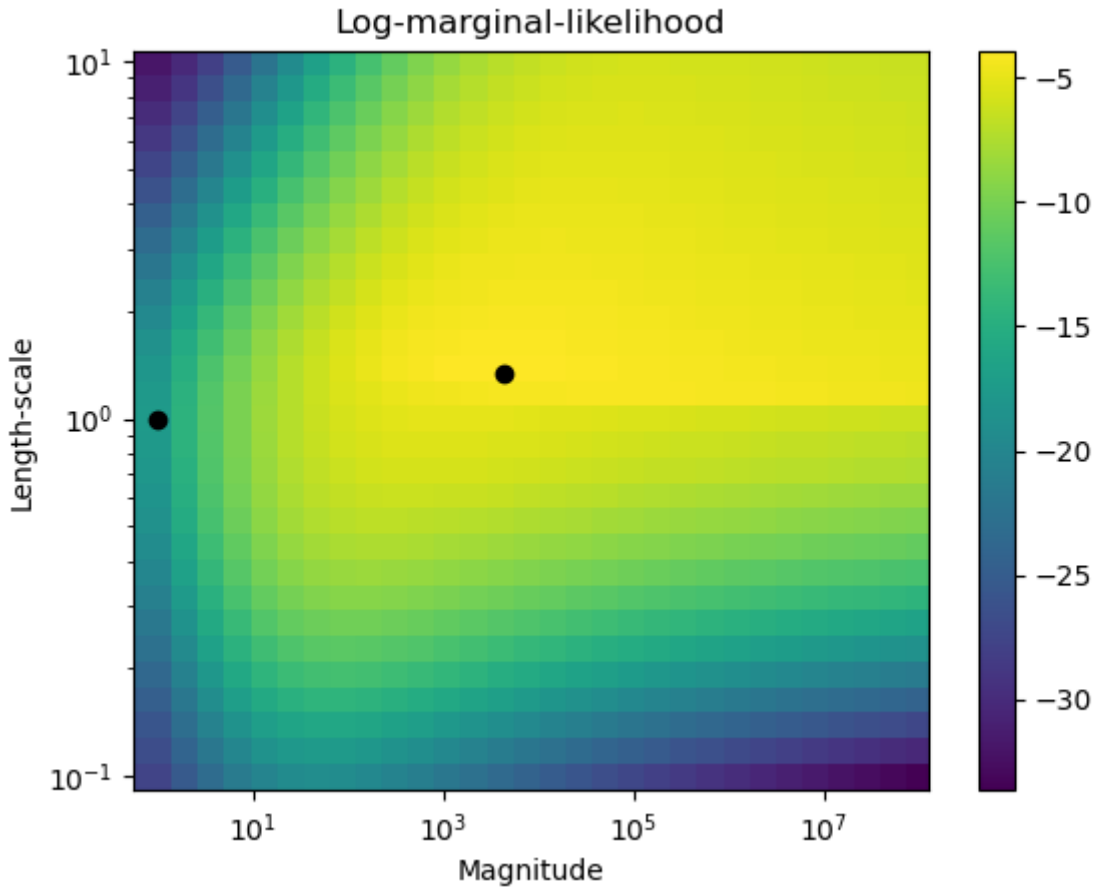
1.7.4.1. Probabilistic predictions with GPC

This example illustrates the predicted probability of GPC for an RBF kernel with different choices of the hyperparameters. The first figure shows the predicted probability of GPC with arbitrarily chosen hyperparameters and with the hyperparameters corresponding to the maximum log-marginal-likelihood (LML).

While the hyperparameters chosen by optimizing LML have a considerably larger LML, they perform slightly worse according to the log-loss on test data. The figure shows that this is because they exhibit a steep change of the class probabilities at the class boundaries (which is good) but have predicted probabilities close to 0.5 far away from the class boundaries (which is bad) This undesirable effect is caused by the Laplace approximation used internally by GPC.

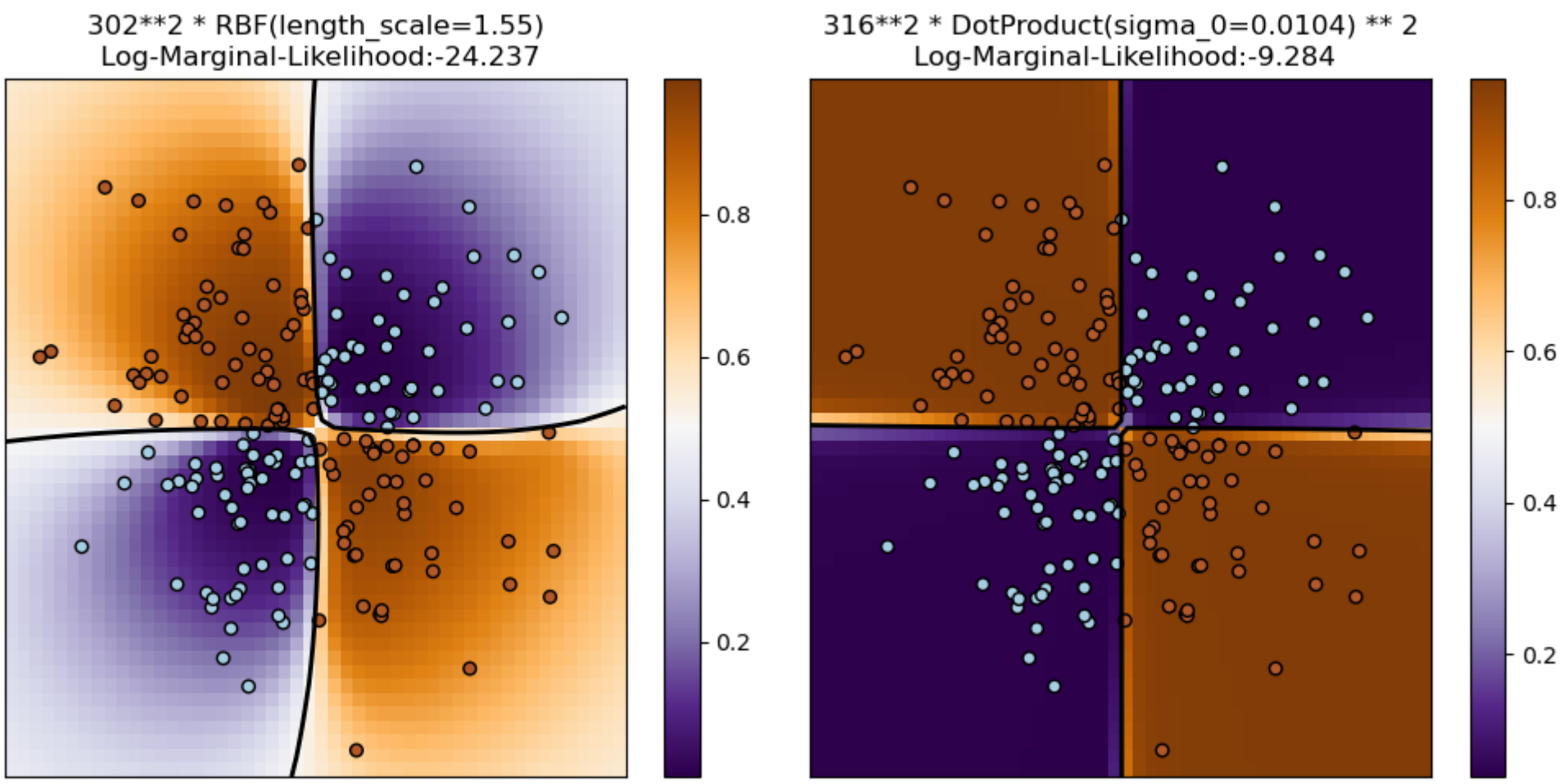
The second figure shows the log-marginal-likelihood for different choices of the kernel's hyperparameters, highlighting the two choices of the hyperparameters used in the first figure by black dots.





1.7.4.2. Illustration of GPC on the XOR dataset

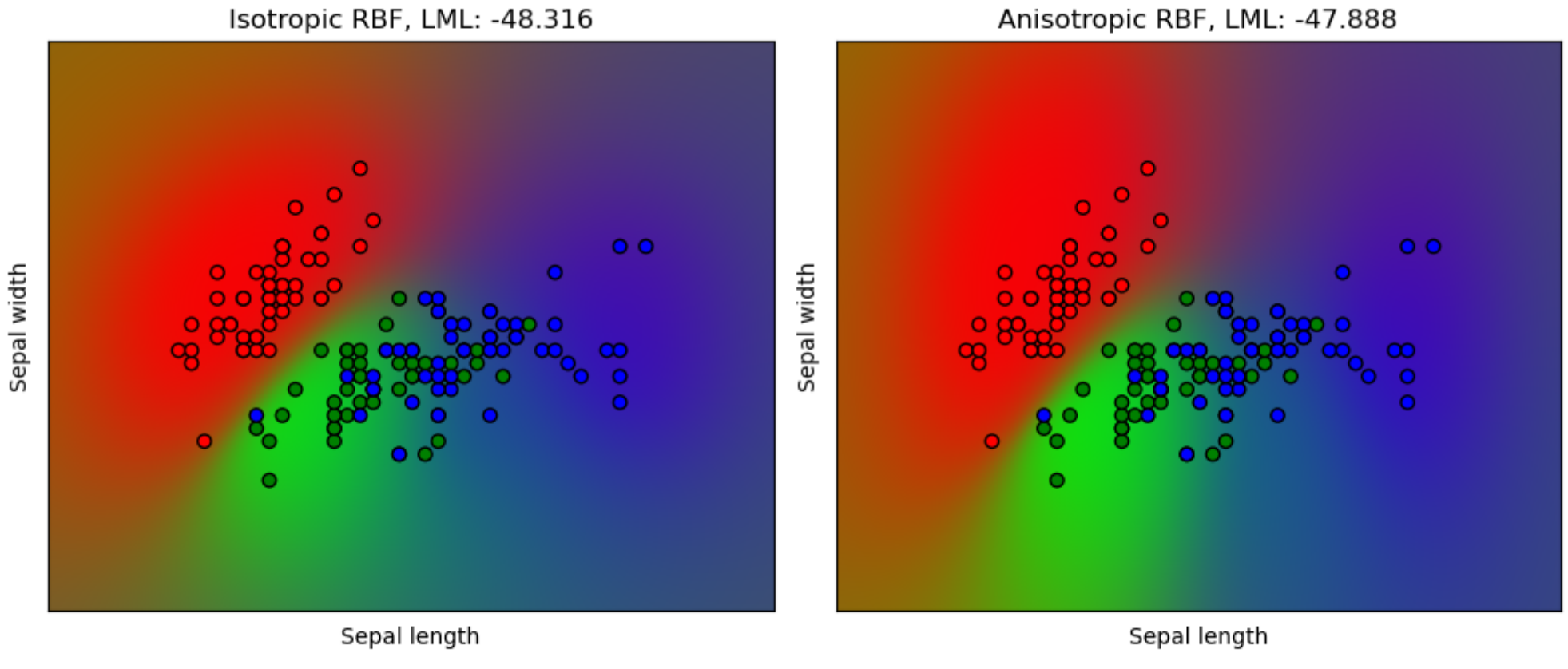
This example illustrates GPC on XOR data. Compared are a stationary, isotropic kernel (`RBF`) and a non-stationary kernel (`DotProduct`). On this particular dataset, the `DotProduct` kernel obtains considerably better results because the class-boundaries are linear and coincide with the coordinate axes. In practice, however, stationary kernels such as `RBF` often obtain better results.



1.7.4.3. Gaussian process classification (GPC) on iris dataset

This example illustrates the predicted probability of GPC for an isotropic and anisotropic RBF kernel on a two-dimensional version for the iris-dataset. This illustrates the applicability of GPC to non-binary classification. The anisotropic RBF kernel obtains slightly higher log-marginal-likelihood by assigning different length-scales to the two feature dimensions.

Toggle Menu



1.7.5. Kernels for Gaussian Processes

Kernels (also called "covariance functions" in the context of GPs) are a crucial ingredient of GPs which determine the shape of prior and posterior of the GP. They encode the assumptions on the function being learned by defining the "similarity" of two datapoints combined with the assumption that similar datapoints should have similar target values. Two categories of kernels can be distinguished: stationary kernels depend only on the distance of two datapoints and not on their absolute values $k(x_i, x_j) = k(d(x_i, x_j))$ and are thus invariant to translations in the input space, while non-stationary kernels depend also on the specific values of the datapoints. Stationary kernels can further be subdivided into isotropic and anisotropic kernels, where isotropic kernels are also invariant to rotations in the input space. For more details, we refer to Chapter 4 of [RW2006]. For guidance on how to best combine different kernels, we refer to [Duv2014].

1.7.5.1. Gaussian Process Kernel API

The main usage of a `Kernel` is to compute the GP's covariance between datapoints. For this, the method `__call__` of the kernel can be called. This method can either be used to compute the "auto-covariance" of all pairs of datapoints in a 2d array X, or the "cross-covariance" of all combinations of datapoints of a 2d array X with datapoints in a 2d array Y. The following identity holds true for all kernels k (except for the `WhiteKernel`): `k(X) == K(X, Y=X)`

If only the diagonal of the auto-covariance is being used, the method `diag()` of a kernel can be called, which is more computationally efficient than the equivalent call to `__call__`: `np.diag(k(X, X)) == k.diag(X)`

Kernels are parameterized by a vector θ of hyperparameters. These hyperparameters can for instance control length-scales or periodicity of a kernel (see below). All kernels support computing analytic gradients of the kernel's auto-covariance with respect to $log(\theta)$ via setting `eval_gradient=True` in the `__call__` method. That is, a `(len(X), len(X), len(theta))` array is returned where the entry `[i, j, l]` contains $\frac{\partial k_\theta(x_i, x_j)}{\partial log(\theta_l)}$. This gradient is used by the Gaussian process (both regressor and classifier) in computing the gradient of the log-marginal-likelihood, which in turn is used to determine the value of θ, which maximizes the log-marginal-likelihood, via gradient ascent. For each hyperparameter, the initial value and the bounds need to be specified when creating an instance of the kernel. The current value of θ can be get and set via the property `theta` of the kernel object. Moreover, the bounds of the hyperparameters can be accessed by the property `bounds` of the kernel. Note that both properties (theta and bounds) return log-transformed values of the internally used values since those are typically more amenable to gradient-based optimization. The specification of each hyperparameter is stored in the form of an instance of `Hyperparameter` in the respective kernel. Note that a kernel using a hyperparameter with name "x" must have the attributes self.x and self.x_bounds.

The abstract base class for all kernels is `Kernel`. Kernel implements a similar interface as `Estimator`, providing the methods `get_params()`, `set_params()`, and `clone()`. This allows setting kernel values also via meta-estimators such as `Pipeline` or `GridSearch`. Note that due to the nested structure of kernels (by applying kernel operators, see below), the names of kernel parameters might become relatively complicated. In general, for a binary kernel operator, parameters of the left operand are prefixed with `k1__` and parameters of the right operand with `k2__`. An additional convenience method is `clone_with_theta(theta)`, which returns a cloned version of the kernel but with the hyperparameters set to `theta`. An illustrative example:

Toggle Menu

```
>>> from sklearn.gaussian_process.kernels import ConstantKernel, RBF
>>> kernel = ConstantKernel(constant_value=1.0, constant_value_bounds=(0.0, 10.0)) * RBF(length_scale=0.5,
length_scale_bounds=(0.0, 10.0)) + RBF(length_scale=2.0, length_scale_bounds=(0.0, 10.0))
>>> for hyperparameter in kernel.hyperparameters: print(hyperparameter)
Hyperparameter(name='k1__k1__constant_value', value_type='numeric', bounds=array([[ 0., 10.]]), n_elements=1, fixed=False)
Hyperparameter(name='k1__k2__length_scale', value_type='numeric', bounds=array([[ 0., 10.]]), n_elements=1, fixed=False)
Hyperparameter(name='k2__length_scale', value_type='numeric', bounds=array([[ 0., 10.]]), n_elements=1, fixed=False)
>>> params = kernel.get_params()
>>> for key in sorted(params): print("%s : %s" % (key, params[key]))
k1 : 1**2 * RBF(length_scale=0.5)
k1__k1 : 1**2
k1__k1__constant_value : 1.0
k1__k1__constant_value_bounds : (0.0, 10.0)
k1__k2 : RBF(length_scale=0.5)
k1__k2__length_scale : 0.5
k1__k2__length_scale_bounds : (0.0, 10.0)
k2 : RBF(length_scale=2)
k2__length_scale : 2.0
k2__length_scale_bounds : (0.0, 10.0)
>>> print(kernel.theta)  # Note: log-transformed
[ 0.         -0.69314718  0.69314718]
>>> print(kernel.bounds)  # Note: log-transformed
[[      -inf 2.30258509]
 [      -inf 2.30258509]
 [      -inf 2.30258509]]
```

All Gaussian process kernels are interoperable with `sklearn.metrics.pairwise` and vice versa: instances of subclasses of `Kernel` can be passed as `metric` to `pairwise_kernels` from `sklearn.metrics.pairwise`. Moreover, kernel functions from pairwise can be used as GP kernels by using the wrapper class `PairwiseKernel`. The only caveat is that the gradient of the hyperparameters is not analytic but numeric and all those kernels support only isotropic distances. The parameter `gamma` is considered to be a hyperparameter and may be optimized. The other kernel parameters are set directly at initialization and are kept fixed.

1.7.5.2. Basic kernels

The `ConstantKernel` kernel can be used as part of a `Product` kernel where it scales the magnitude of the other factor (kernel) or as part of a `Sum` kernel, where it modifies the mean of the Gaussian process. It depends on a parameter $constant\_value$. It is defined as:

$$k(x_i, x_j) = constant\_value \;\forall\; x_1, x_2$$

The main use-case of the `WhiteKernel` kernel is as part of a sum-kernel where it explains the noise-component of the signal. Tuning its parameter $noise\_level$ corresponds to estimating the noise-level. It is defined as:

$$k(x_i, x_j) = noise\_level \text{ if } x_i == x_j \text{ else } 0$$

1.7.5.3. Kernel operators

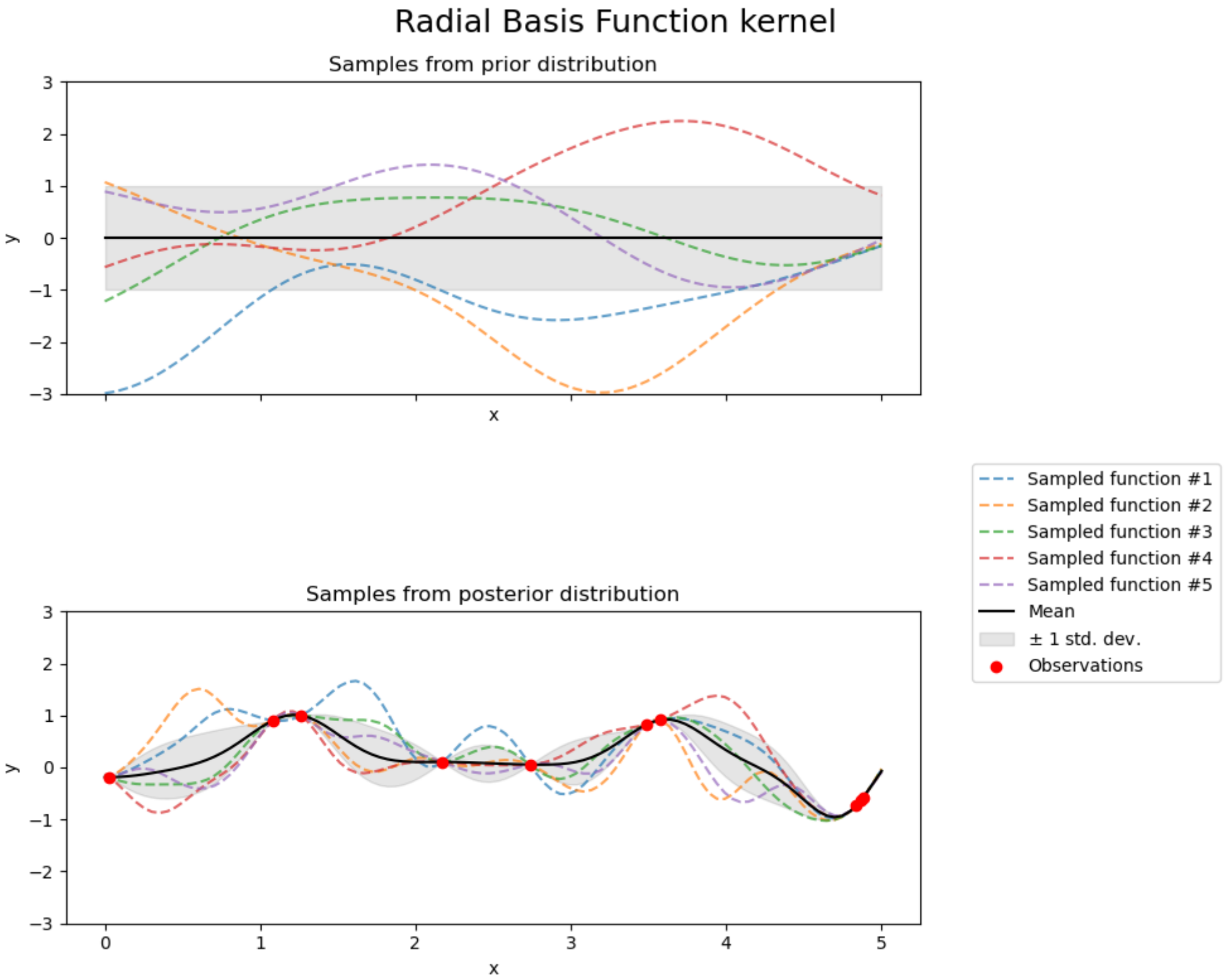
Kernel operators take one or two base kernels and combine them into a new kernel. The `Sum` kernel takes two kernels k_1 and k_2 and combines them via $k_{sum}(X, Y) = k_1(X, Y) + k_2(X, Y)$. The `Product` kernel takes two kernels k_1 and k_2 and combines them via $k_{product}(X, Y) = k_1(X, Y) * k_2(X, Y)$. The `Exponentiation` kernel takes one base kernel and a scalar parameter p and combines them via $k_{exp}(X, Y) = k(X, Y)^p$. Note that magic methods `__add__`, `__mul___` and `__pow__` are overridden on the Kernel objects, so one can use e.g. `RBF() + RBF()` as a shortcut for `Sum(RBF(), RBF())`.

1.7.5.4. Radial basis function (RBF) kernel

The `RBF` kernel is a stationary kernel. It is also known as the “squared exponential” kernel. It is parameterized by a length-scale parameter $l > 0$, which can either be a scalar (isotropic variant of the kernel) or a vector with the same number of dimensions as the inputs x (anisotropic variant of the kernel). The kernel is given by:

$$k(x_i, x_j) = \exp\left(-\frac{d(x_i, x_j)^2}{2l^2}\right)$$

where $d(\cdot, \cdot)$ is the Euclidean distance. This kernel is infinitely differentiable, which implies that GPs with this kernel as covariance function have mean square derivatives of all orders, and are thus very smooth. The prior and posterior of a GP resulting from an RBF kernel are shown in the following figure:



1.7.5.5. Matérn kernel

The `Matern` kernel is a stationary kernel and a generalization of the `RBF` kernel. It has an additional parameter ν which controls the smoothness of the resulting function. It is parameterized by a length-scale parameter $l > 0$, which can either be a scalar (isotropic variant of the kernel) or a vector with the same number of dimensions as the inputs x (anisotropic variant of the kernel). The kernel is given by:

$$k(x_i, x_j) = \frac{1}{\Gamma(\nu)2^{\nu-1}}\left(\frac{\sqrt{2\nu}}{l}d(x_i, x_j)\right)^\nu K_\nu\left(\frac{\sqrt{2\nu}}{l}d(x_i, x_j)\right),$$

where $d(\cdot,\cdot)$ is the Euclidean distance, $K_\nu(\cdot)$ is a modified Bessel function and $\Gamma(\cdot)$ is the gamma function. As $\nu \to \infty$, the Matérn kernel converges to the RBF kernel. When $\nu = 1/2$, the Matérn kernel becomes identical to the absolute exponential kernel, i.e.,

$$k(x_i, x_j) = \exp\left(-\frac{1}{l}d(x_i, x_j)\right) \qquad \nu = \tfrac{1}{2}$$

In particular, $\nu = 3/2$:

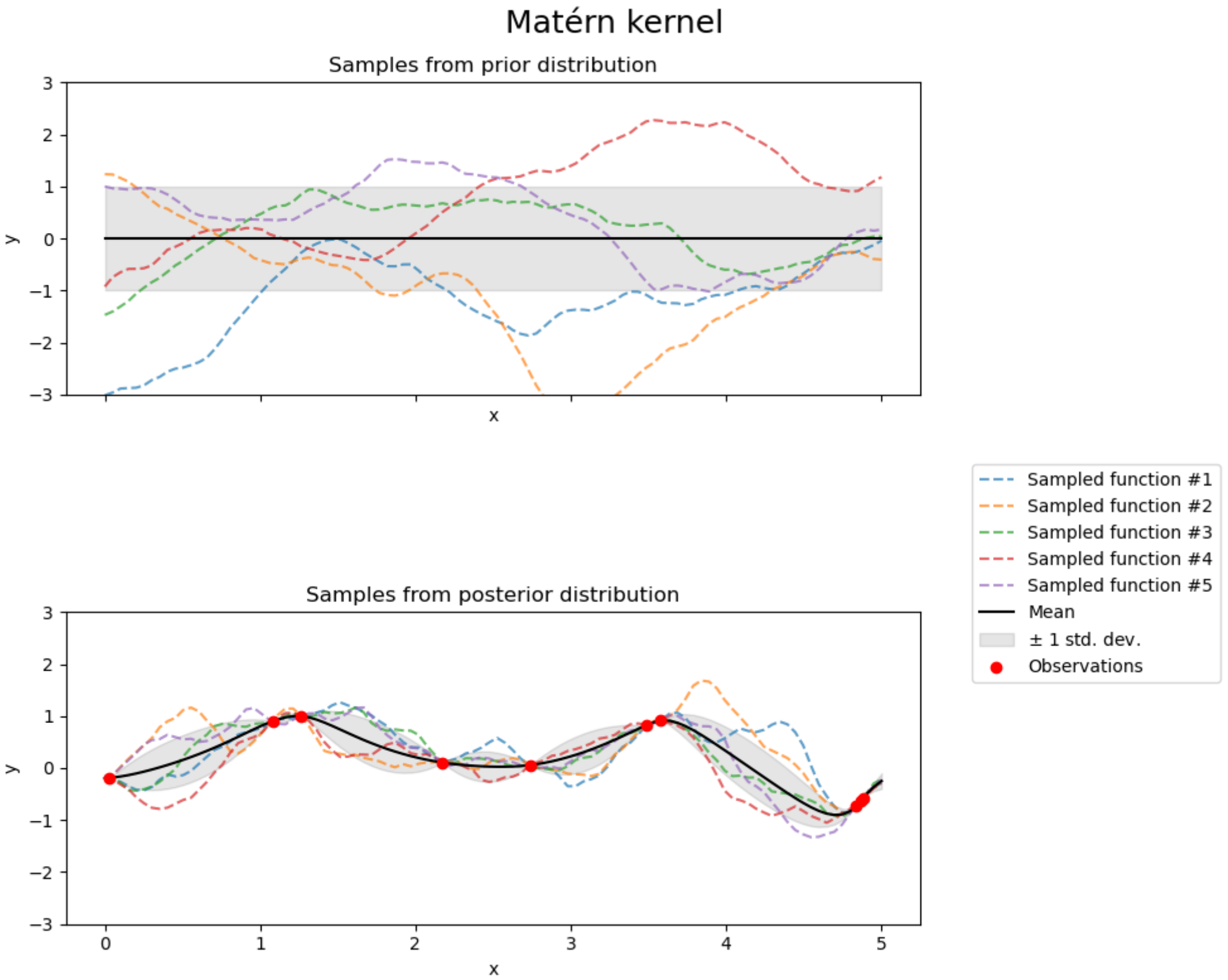
$$k(x_i, x_j) = \left(1 + \frac{\sqrt{3}}{l}d(x_i, x_j)\right)\exp\left(-\frac{\sqrt{3}}{l}d(x_i, x_j)\right) \qquad \nu = \tfrac{3}{2}$$

and $\nu = 5/2$:

$$k(x_i, x_j) = \left(1 + \frac{\sqrt{5}}{l}d(x_i, x_j) + \frac{5}{3l}d(x_i, x_j)^2\right)\exp\left(-\frac{\sqrt{5}}{l}d(x_i, x_j)\right) \qquad \nu = \tfrac{5}{2}$$

are popular choices for learning functions that are not infinitely differentiable (as assumed by the RBF kernel) but at least once ($\nu = 3/2$) or twice differentiable ($\nu = 5/2$).

The flexibility of controlling the smoothness of the learned function via ν allows adapting to the properties of the true underlying functional relation. The prior and posterior of a GP resulting from a Matérn kernel are shown in the following figure:



See [RW2006], pp84 for further details regarding the different variants of the Matérn kernel.

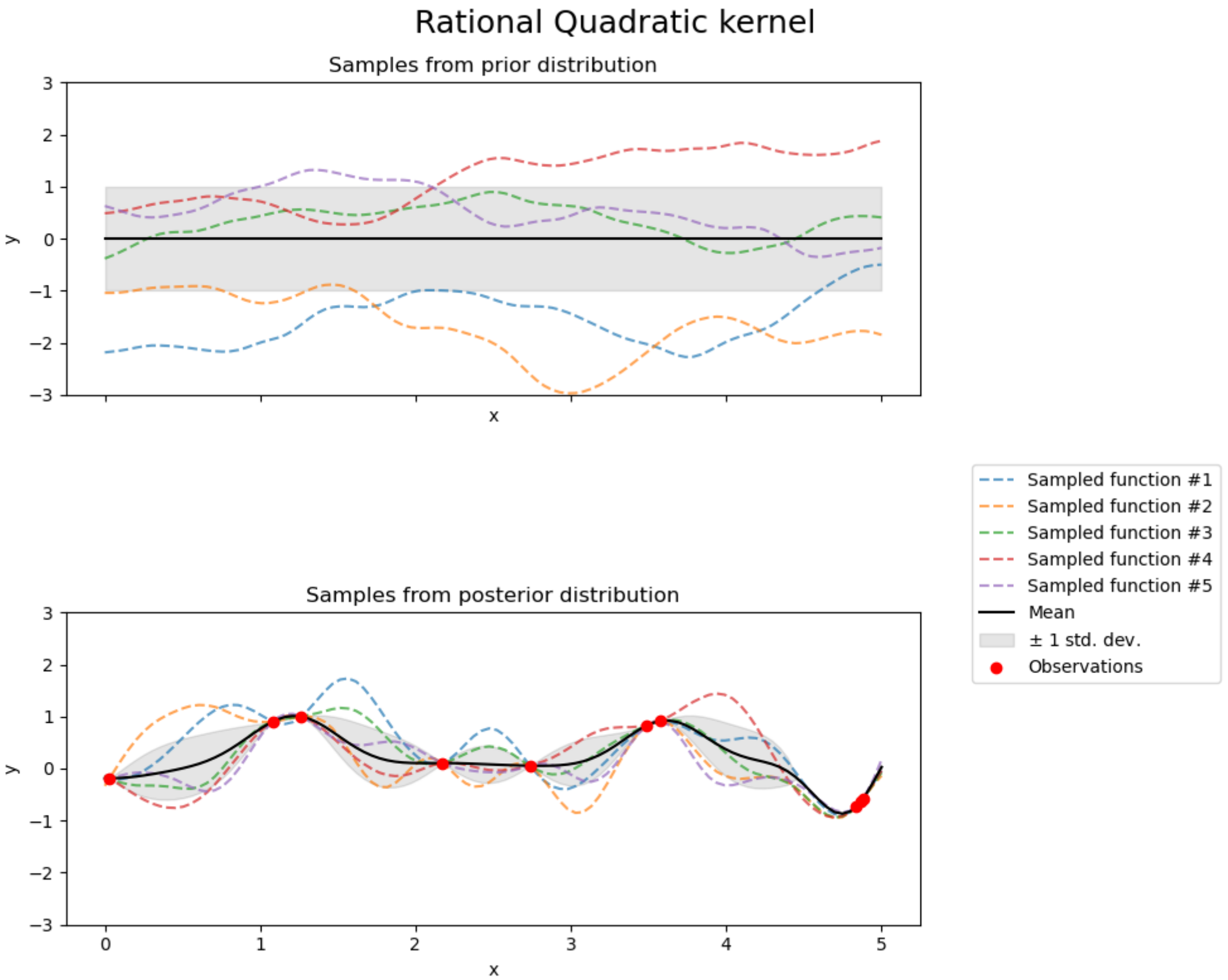
1.7.5.6. Rational quadratic kernel

The `RationalQuadratic` kernel can be seen as a scale mixture (an infinite sum) of `RBF` kernels with different characteristic length-scales. It is parameterized by a length-scale parameter $l > 0$ and a scale mixture parameter $\alpha > 0$ Only the isotropic variant where l is a scalar is supported at the moment. The kernel is given by:

$$k(x_i, x_j) = \left(1 + \frac{d(x_i, x_j)^2}{2\alpha l^2}\right)^{-\alpha}$$

The prior and posterior of a GP resulting from a `RationalQuadratic` kernel are shown in the following figure:

Toggle Menu



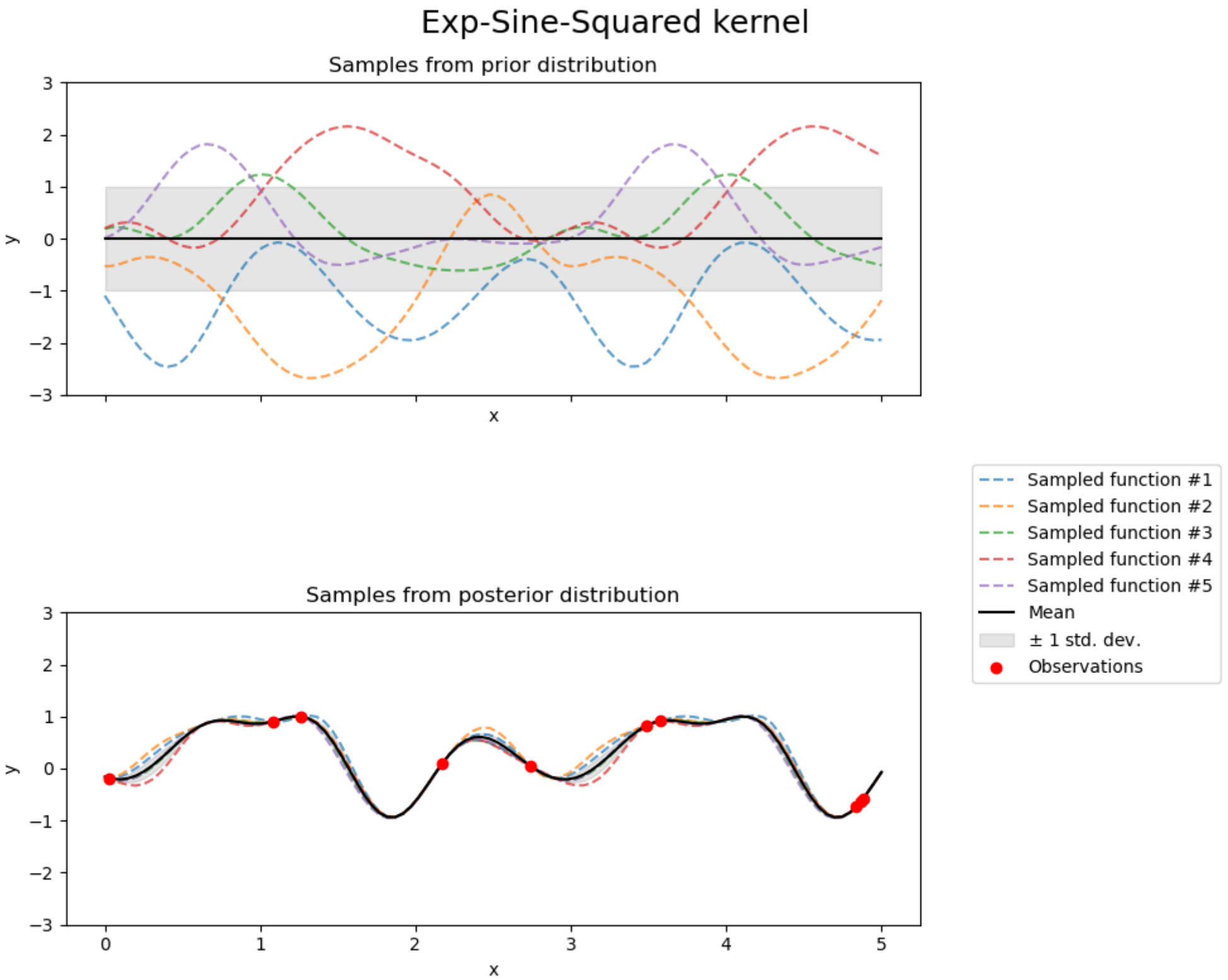
1.7.5.7. Exp-Sine-Squared kernel

The `ExpSineSquared` kernel allows modeling periodic functions. It is parameterized by a length-scale parameter $l > 0$ and a periodicity parameter $p > 0$. Only the isotropic variant where l is a scalar is supported at the moment. The kernel is given by:

$$k(x_i, x_j) = \exp\left(-\frac{2\sin^2(\pi d(x_i, x_j)/p)}{l^2}\right)$$

The prior and posterior of a GP resulting from an ExpSineSquared kernel are shown in the following figure:

Toggle Menu

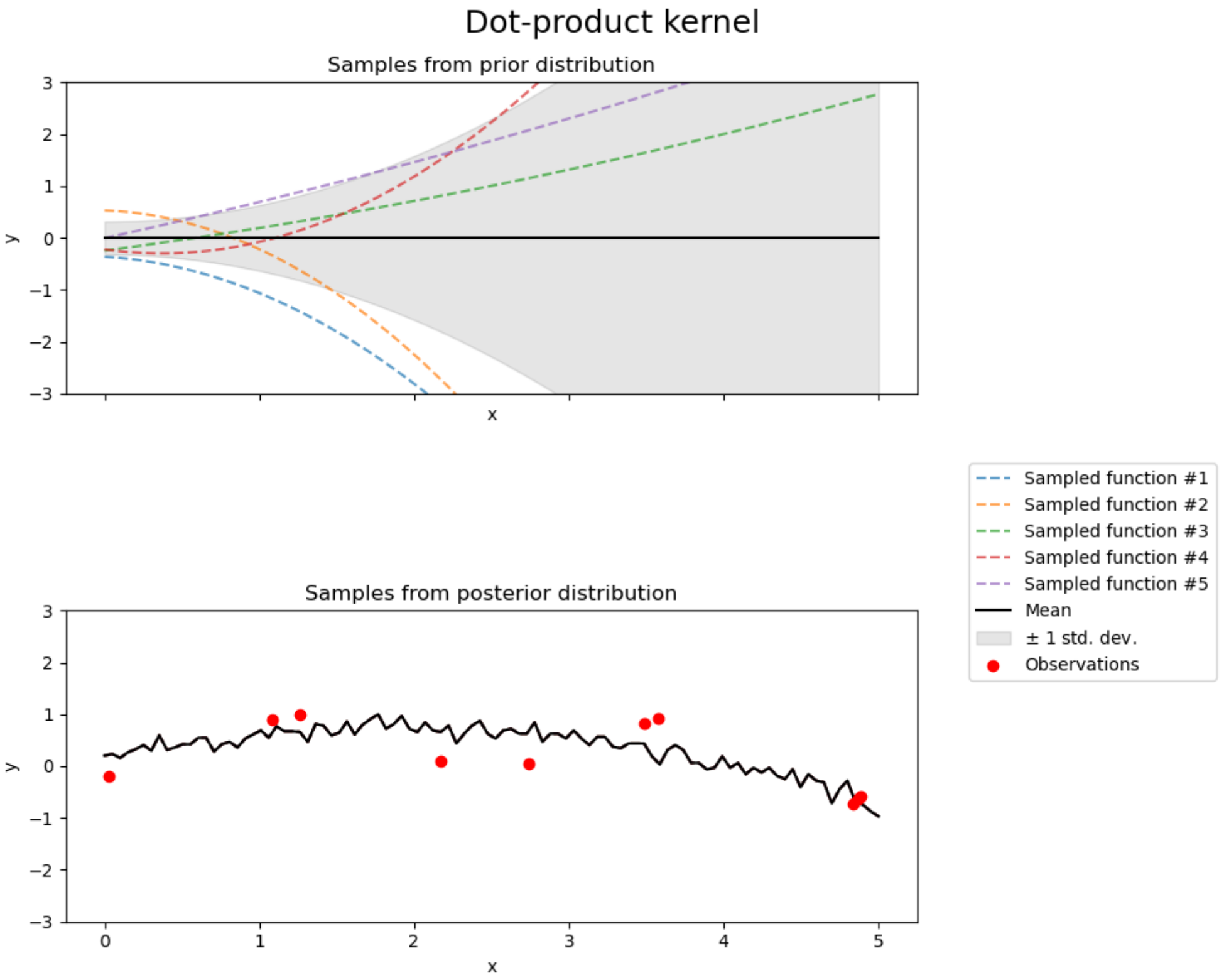


1.7.5.8. Dot-Product kernel

The `DotProduct` kernel is non-stationary and can be obtained from linear regression by putting $N(0, 1)$ priors on the coefficients of $x_d (d = 1, \ldots, D)$ and a prior of $N(0, \sigma_0^2)$ on the bias. The `DotProduct` kernel is invariant to a rotation of the coordinates about the origin, but not translations. It is parameterized by a parameter σ_0^2. For $\sigma_0^2 = 0$, the kernel is called the homogeneous linear kernel, otherwise it is inhomogeneous. The kernel is given by

$$k(x_i, x_j) = \sigma_0^2 + x_i \cdot x_j$$

The `DotProduct` kernel is commonly combined with exponentiation. An example with exponent 2 is shown in the following figure:



1.7.5.9. References

[RW2006] (1,2,3,4,5,6)
Carl E. Rasmussen and Christopher K.I. Williams, “Gaussian Processes for Machine Learning”, MIT Press 2006

[Duv2014]
David Duvenaud, “The Kernel Cookbook: Advice on Covariance functions”, 2014