Exercise_06

June 7, 2021

1 Probabilistic Machine Learning

University of Tübingen, Summer Term 2021

1.1 Exercise Sheet 6

This sheet is **due on Tuesday 8 June 2021 at 10am sharp.**

---

2 GP Hyperparameter Optimization

Laste exercise we implemented a set of kernels, which encoded properties about the types of functions a Gaussian process can model, e.g. differentiability. These kernels had *hyperparameters*, which defined the shape of those functions such as their periodicity or lengthscale ("wiggliness").

When fitting a Gaussian process using your `GaussianProcess` class, you specified these hyperparameters when passing the kernel. Depending on the choice of hyperparameters, the prediction made by the GP can be quite poor. Today, we will investigate how to automatically find hyperparameter values of the chosen kernel to obtain a better fit to the data.

[1]:
```
# Plotting setup
import matplotlib.pyplot as plt
import datetime

# Make inline plots vector graphics
%matplotlib inline
from IPython.display import set_matplotlib_formats
set_matplotlib_formats("pdf", "svg")

# Package imports
import numpy as np
import pandas as pd
```

2.1 Kernel Hyperparameters

Consider again the exponentiated quadratic / RBF kernel multiplied with an output scale, given by

$$k(x_0, x_1) = o^2 \exp\left(-\frac{\|x_0 - x_1\|^2}{2\ell^2}\right)$$

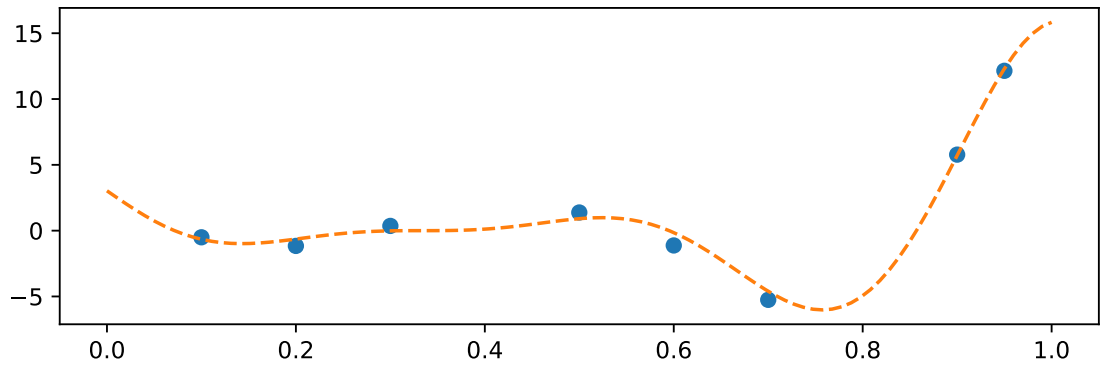
with lengthscale $\ell \in \mathbb{R}$ and output scale o. The lengthscale controls how fast the modelled functions "wiggle" (on the x-axis) while the output scale controls how large (on the y-axis) the wiggles are.

[2]:
```
import scipy.spatial.distance


def expquad(x0, x1=None, outputscale=1.0, lengthscale=1.0):
    if x1 is None:
        x1 = x0
    pdists = scipy.spatial.distance.cdist(x0, x1, metric="euclidean")
    return outputscale ** 2 * np.exp(-(pdists ** 2) / (2.0 * lengthscale ** 2))
```

We'll generate some synthetic data to work with.

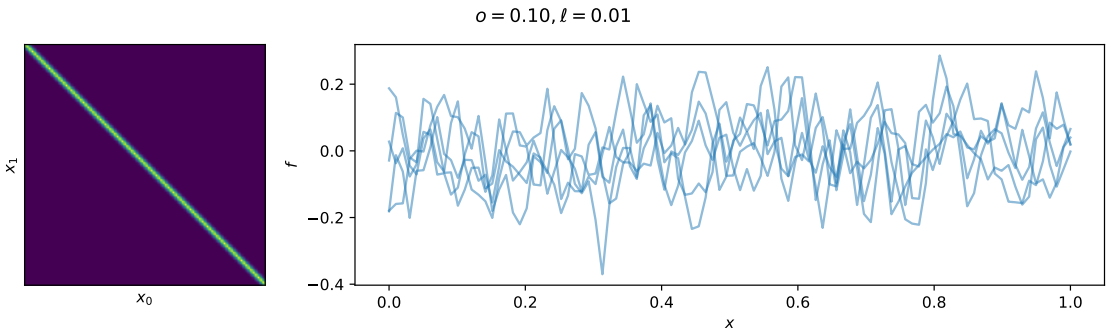
[3]:
```
rng = np.random.default_rng(42)

# Latent function
def f(x):
    return (6 * x - 2) ** 2 * np.sin(12 * x - 4)

# Training data
X_train = np.array([0.1, 0.2, 0.3, 0.5, 0.6, 0.7, 0.9, 0.95])[:, None]
y_train = f(X_train) + 0.5 * rng.normal(size=X_train.shape)

# Test locations
X_test = np.linspace(0, 1, 100)[:, None]
```

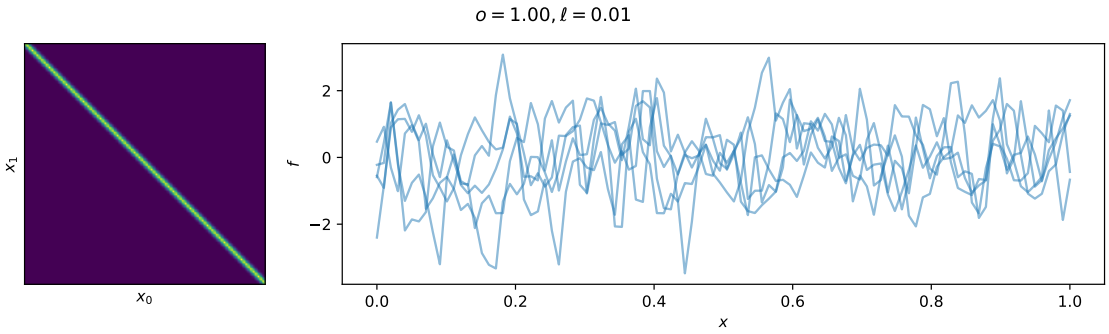
[4]:
```
# Plot
fig, ax = plt.subplots(1, 1, figsize=(8, 2.5))
ax.plot(X_test, f(X_test), color="C1", linestyle="--")
ax.scatter(X_train, y_train)
plt.show()
```

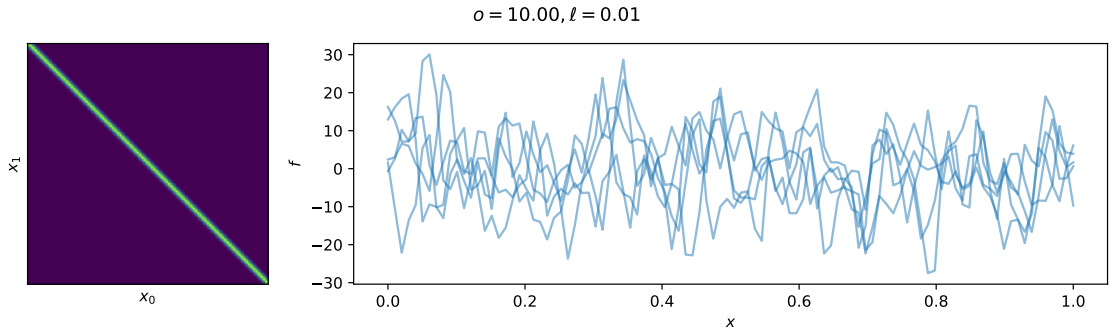


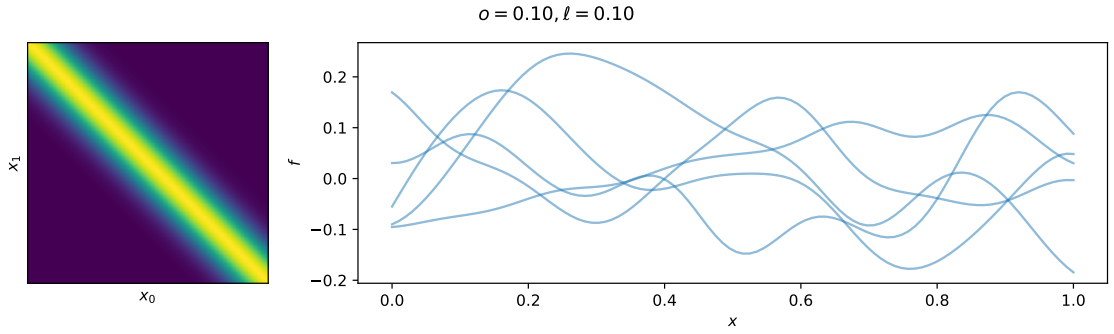
Task: Plot kernel matrices and samples from a GP with the `expquad` kernel using the functions `plot_kernel_matrix` and `plot_prior_samples` for a selection of different values of the outputscale and lengthscale.

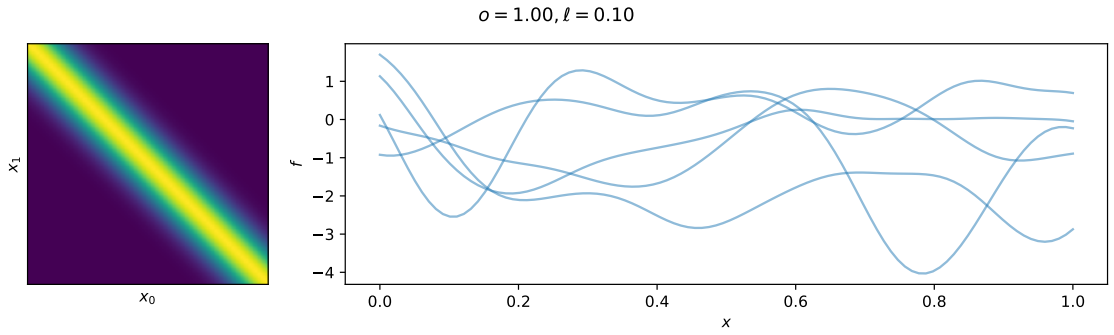
[5]:
```
from ex06 import plot_kernel_matrix, plot_prior_samples

for o, l in zip([0.1, 1, 10, 0.1, 1, 10, 0.1, 1, 10], [0.01, 0.01, 0.01, .1, .
↪1, .1, 1, 1, 1]):
    fig, ax = plt.subplots(1, 2, figsize=(10, 3), constrained_layout=True,␣
↪gridspec_kw={'width_ratios': [1, 3]})
    fig.suptitle(fr"$o = {o:.2f}, \ell = {l:.2f}$")
    plot_kernel_matrix(ax[0], X_test, expquad, outputscale=o, lengthscale=l)
    plot_prior_samples(ax[1], X_test, expquad, outputscale=o, lengthscale=l)
    plt.show()
```

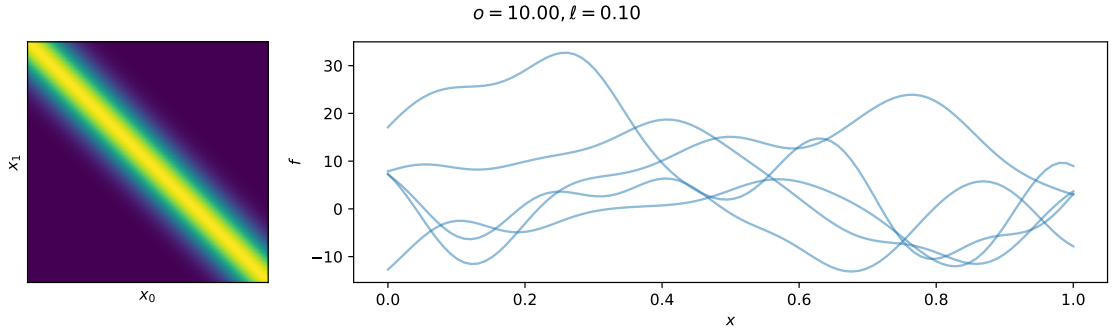


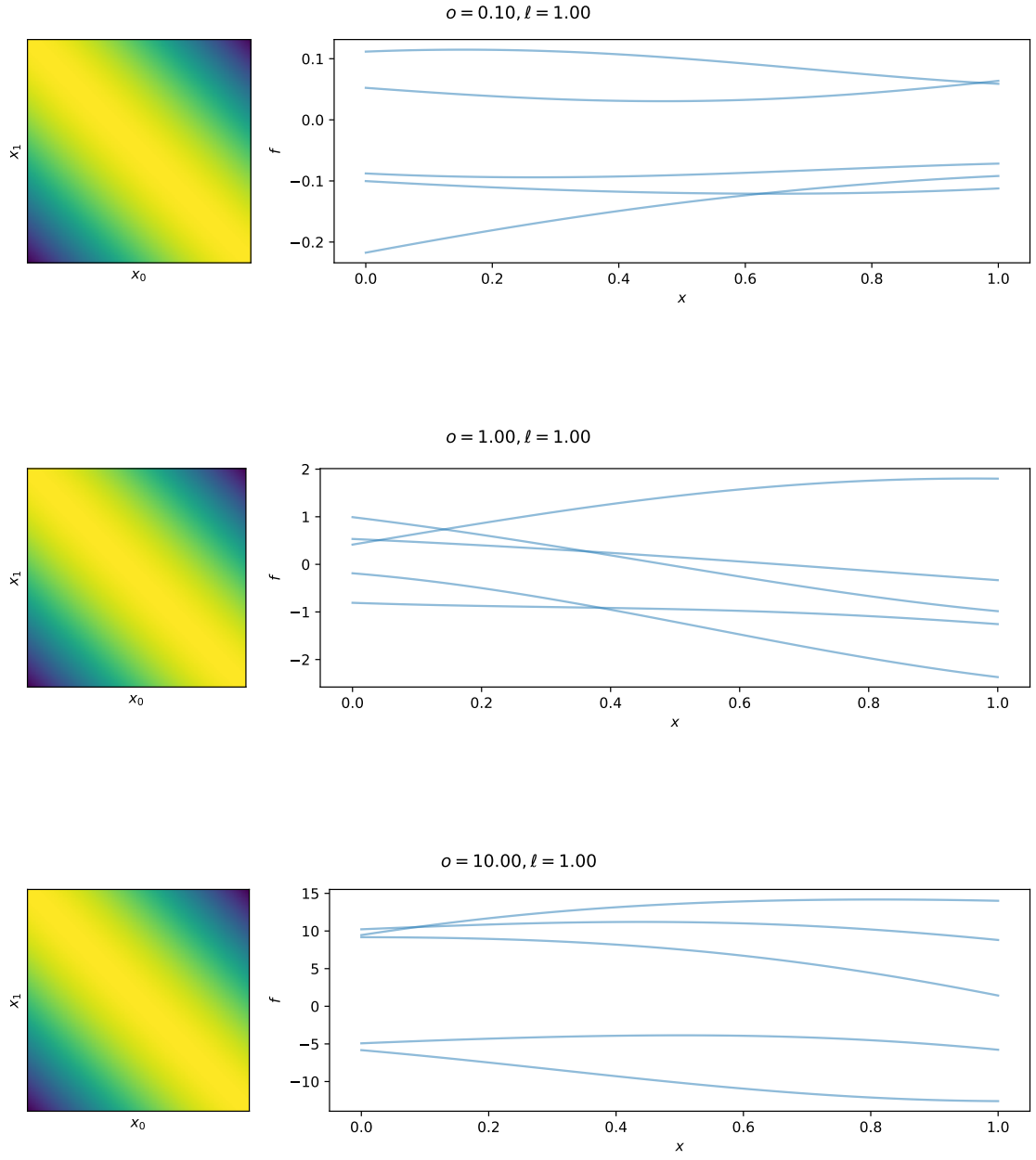



$o = 10.00, \ell = 0.01$
x_1
x_0
f
x


$o = 0.10, \ell = 0.10$
x_1
x_0
f
x


$o = 1.00, \ell = 0.10$
x_1
x_0
f
x


$o = 10.00, \ell = 0.10$
x_1
x_0
f
x



2.2 Log-marginal Likelihood

From a Bayesian viewpoint the fit of our function to the data is described by the *log-marginal likelihood* or *evidence*:

$$\log p(y \mid X, \theta) = -\frac{1}{2}(y^\top K^{-1} y + \log \det K + n \log(2\pi)),$$

where $K = k_\theta(X, X) + \sigma^2 I$ and θ are the kernel hyperparameters (e.g. lengthscale) and σ^2 the observation noise. The two terms involving the training data (X, y) are the data fit $-\frac{1}{2} y^\top K^{-1} y$, measuring how "surprising" it is to see data y assuming kernel k and a complexity penalty $-\frac{1}{2} \log \det K$ describing how "complicated" the functions are your kernel is modelling. These two terms describe an automatic trade-off between fitting the data as well as possible, but also favoring simpler models / explanations for the data (c.f. Occam's razor), which in turn avoids overfitting.

Task: Fit the provided GP implementation `GaussianProcess` to the data for different kernel hyperparameter choices. What's the lowest log-marginal likelihood you find and for which parameters? How does the observation noise σ^2 affect the value of the log-marginal likelihood?

[6]:
```
from ex06 import GaussianProcess
from tabulate import tabulate

O = [1, 2, 5, 10, 15, 20]
L = [0.05, .1, .2, .3]
lml = []
for o in O:
    lml.append([o])
    for l in L:
        # Mean and covariance with hyperparameter choice
        mu = lambda x: np.zeros(shape=(x.shape[0], 1))
        cov = lambda x0, x1=None: expquad(x0=x0, x1=x1, outputscale=o,␣
 ↪lengthscale=l)

        # Define Gaussian process
        gp = GaussianProcess(mu, cov)
        gp.fit(X_train, y_train)

        # Prediction
        y_pred = gp.predict(X_test)

        # Evidence
        lml[-1].append(gp.log_marginal_likelihood())

print(tabulate(lml, headers=["↓ o \ l →"] + L))

result = np.asarray(lml)[:, 1:]
idx = np.unravel_index(np.argmax(result), result.shape)
print(f"\nBest likelihood: {result[idx]:.2f}, for o = {O[idx[0]]}, l =␣
 ↪{L[idx[1]]:.1f}.")
```

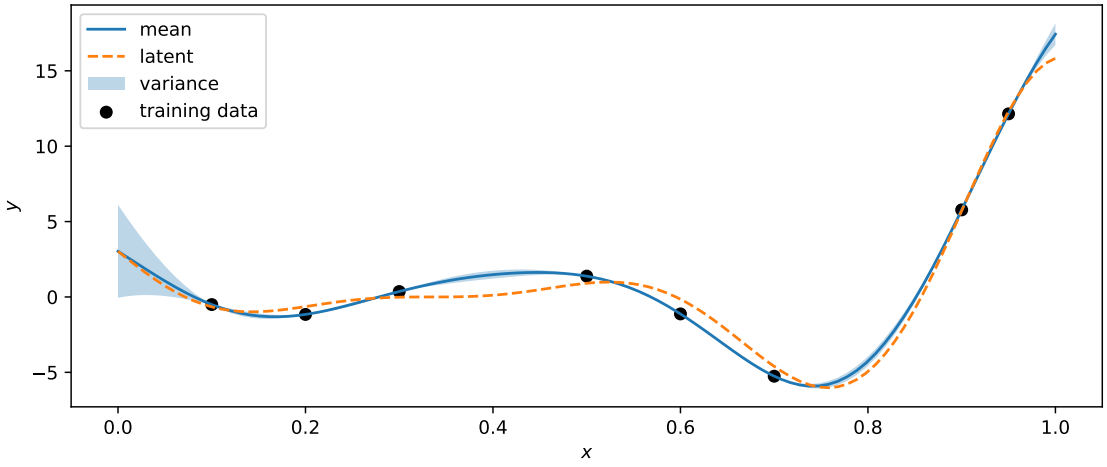
```
  ↓ o \ l →      0.05        0.1        0.2         0.3
-----------  --------  ---------  ---------  ----------
          1  -98.6594  -144.927   -550.575   -41771.3
          2  -35.5237   -45.8827  -142.507   -10442.1
          5  -23.6233   -23.9279   -34.0252   -1675.7
         10  -26.4213   -25.2895   -23.0258    -427.854
```

```
15  -29.1563   -27.7585   -23.2058    -198.988
20  -31.2797   -29.7888   -24.4349    -120.052
```

```
Best likelihood: -23.03, for o = 10, l = 0.2.
```

Task: Plot the GP using the hyperparameters you found empirically using `plot_gp` and compare to the true function. Are the hyperparameters you found good choices?

[7]:
```
from ex06 import plot_gp

mu = lambda x: np.zeros(shape=(x.shape[0], 1))
cov = lambda x0, x1=None: expquad(x0=x0, x1=x1, outputscale=10, lengthscale=0.2)
gp = GaussianProcess(mu, cov)
gp.fit(X_train, y_train)
y_pred = gp.predict(X_test)

plot_gp(X_test, y_pred, traindata=(X_train, y_train), latentfun=f)
```



The learned function passes very well through the training points. However, the variance is too low, because at most points the true function lies (far) outside the 2σ interval.

2.2.1 Optimizing Hyperparameters

Clearly tuning hyperparameters by hand is a tedious approach that does not scale beyond a few hyperparameters. Instead, we will automatically find the optimal hyperparameters with respect to the training data by optimizing the evidence with a gradient-based optimizer.

Note: In some settings you may know the true settings for hyperparameters based on domain knowledge. For example, you may observe a weekly pattern in your data, as we did last week in the COVID data. This means your periodic component of your kernel should be fixed at period 7 and not optimized. One can also specify distributions over hyperparameters via hyperpriors, which incorporate prior knowledge about which parameter settings are plausible. This leads to hierarchical Bayesian inference.

Derivative of the log-Marginal Likelihood The derivative of the log-marginal likelihood with respect to the kernel hyperparameters θ is given by: (see Ch. 5.4.1 in GPML for details)

$$\frac{\partial}{\partial \theta} \log p(y \mid X, \theta) = \frac{1}{2} y^\top K^{-1} \frac{\partial K}{\partial \theta} K^{-1} y - \frac{1}{2} \operatorname{tr}\left(K^{-1} \frac{\partial K}{\partial \theta}\right)$$

To evaluate the derivative we need to evaluate the derivative of the kernel matrix with respect to its parameters (element-wise).

Derivatives of the Exponentiated Quadratic Kernel **Task:** Compute the derivatives of $K = k_{\text{RBF}}(X, X) + \sigma^2 I$ with respect to the noise σ and outputscale o analytically and implement them.

[8]:
```
# Note: theta = [outputscale, lengthscale, sigma] is assumed to be a list of␣
 ↪the hyperparameters.

def dK_dsigma(X, theta) -> float:
    """
    Derivative of K = k(X, X) + sigma^2 I with respect to sigma.

    Parameters
    ----------
    X :
        Training inputs. shape=(n x d)
    theta :
        List of hyperparameters [outputscale, lengthscale, sigma]. shape=(3,)
    """
    return 2 * theta[2] * np.eye(len(X))


def dK_do(X, theta) -> float:
    """
    Derivative of K = k(X, X) + sigma^2 I with respect to the outputscale.

    Parameters
    ----------
    X :
        Training inputs. shape=(n x d)
    theta :
        List of hyperparameters [outputscale, lengthscale, sigma]. shape=(3,)
    """
    return 2 * expquad(X, outputscale=theta[0], lengthscale=theta[1]) / theta[0]
```

Task: Derive the expression for the RBF kernel differentiated with respect to the lengthscale

$$\frac{\partial}{\partial \ell} K = \frac{\partial}{\partial \ell} k_{\text{RBF}}(x_0, x_1) = \frac{\partial}{\partial \ell} o^2 \exp\left(-\frac{\|x_0 - x_1\|^2}{2\ell^2}\right) = o^2 \exp\left(-\frac{\|x_0 - x_1\|^2}{2\ell^2}\right) \frac{\|x_0 - x_1\|^2}{\ell^3}$$

and implement it.

[9]:
```
def dK_dl(X, theta) -> float:
    """
    Derivative of K = k(X, X) + sigma^2 I with respect to the lengthscale.

    Parameters
    ----------
    X :
        Training inputs. shape=(n x d)
    theta :
        List of hyperparameters [outputscale, lengthscale, sigma]. shape=(3,)
    """
    pdists = scipy.spatial.distance.cdist(X, X, metric="euclidean")
    return expquad(X, outputscale=theta[0], lengthscale=theta[1]) * pdists**2 /␣
↪theta[1]**3
```

Task: Implement a function computing the *negative* log-marginal likelihood and its derivatives for the RBF kernel.

[10]:
```
from typing import Tuple
from scipy.linalg import solve, det

def nlogML(X, y, theta, dK_dtheta=None) -> Tuple[float, np.ndarray]:
    """
    Negative log-marginal likelihood and its derivatives for the RBF kernel.

    Parameters
    ----------
    X :
        Training inputs. shape=(n x d)
    y :
        Observed data. shape=(n x 1)
    theta :
        List of hyperparameters [outputscale, lengthscale, sigma]. shape=(3,)
    dK_dtheta :
        List of callables computing the derivatives (wrt. outputscale,␣
↪lengthscale, sigma)
        of the kernel matrix.

    Returns
    -------
    nlogML :
        Negative log-marginal likelihood.
    dnlogML_dtheta :
        Array of derivatives of the negative log-marginal likelihood. shape=(3,)
    """
    if dK_dtheta is not None:
```

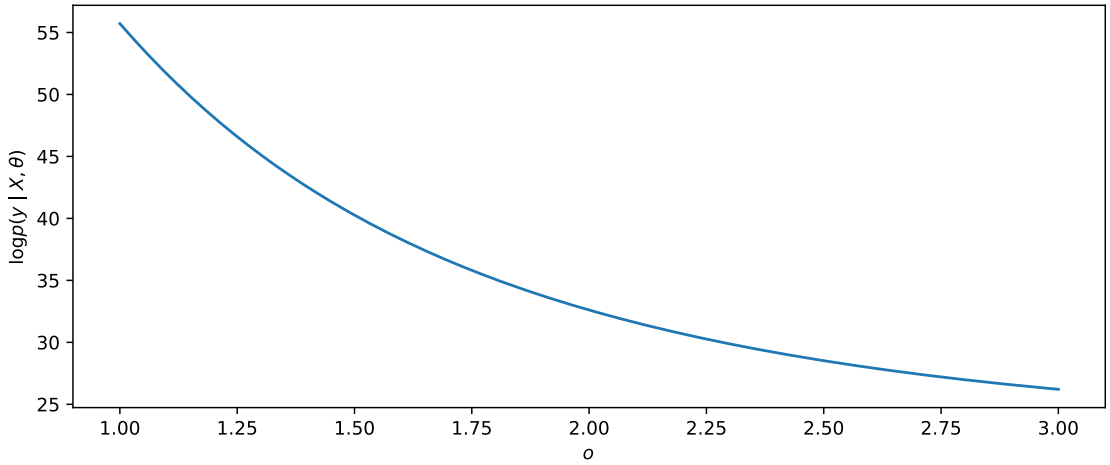
```
        ko, kl, ks = dK_dtheta
    else:
        ko, kl, ks = dK_do, dK_dl, dK_dsigma

    K = expquad(X, outputscale=theta[0], lengthscale=theta[1]) + theta[2]**2 *␣
↪np.eye(len(X))
    nlogML = 1/2 * (y.T @ solve(K, y) + np.linalg.slogdet(K)[1] + len(X)*np.
↪log(2*np.pi))
    dnlogML_do = -1/2 * y.T @ (kts := solve(K, ko(X, theta))) @ solve(K, y) + 1/
↪2 * np.trace(kts)
    dnlogML_dl = -1/2 * y.T @ (kts := solve(K, kl(X, theta))) @ solve(K, y) + 1/
↪2 * np.trace(kts)
    dnlogML_ds = -1/2 * y.T @ (kts := solve(K, ks(X, theta))) @ solve(K, y) + 1/
↪2 * np.trace(kts)

    return nlogML.item(), np.array([dnlogML_do.item(), dnlogML_dl.item(),␣
↪dnlogML_ds.item()])
```

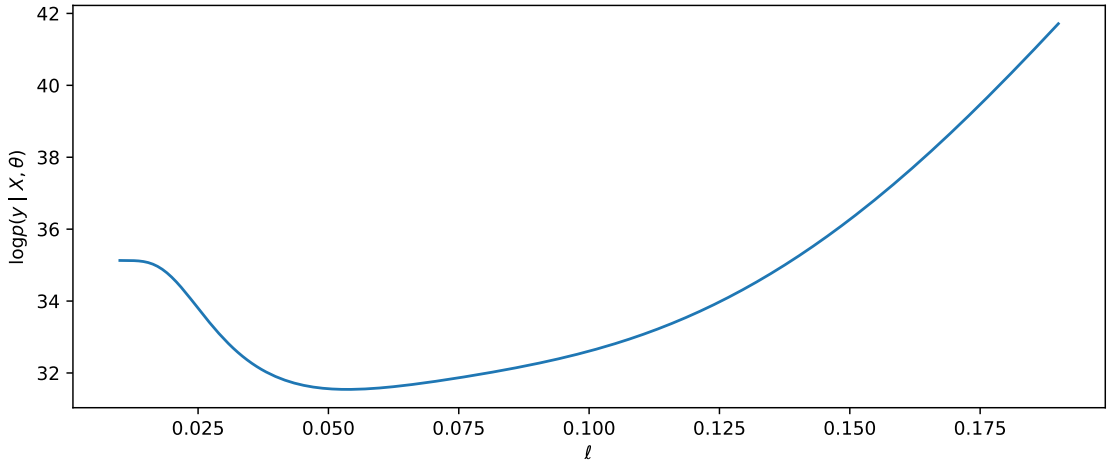
Task: Evaluate the negative log-marginal likelihood and its derivatives at $o = 2$, $\ell = 0.1$ and $\sigma = 1.0$. What does the sign of the gradients tell you?

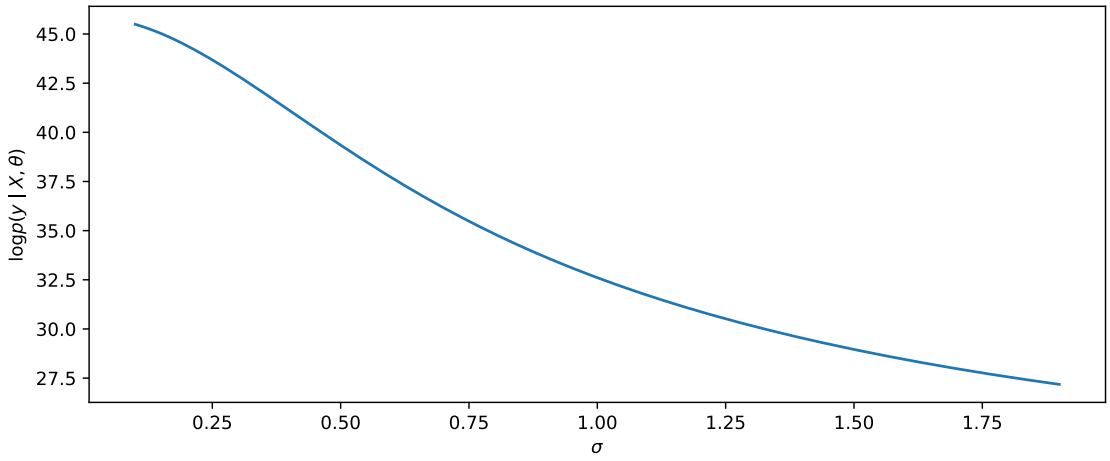
[11]:
```
outputscale = 2.0
lengthscale = 0.1
sigma = 1.0

nll, (do, dl, ds) = nlogML(X_train, y_train, [outputscale, lengthscale, sigma])
print(f"NLL: {nll:.3f}, d/do: {do:.3f}, d/dl: {dl:.3f}, d/ds: {ds:.3f}")

# Plot likelihood in neighborhood when varying components, calculate numerical␣
↪partial derivatives
ndo = (nlogML(X_train, y_train, [outputscale + 1e-6, lengthscale, sigma])[0]
       - nlogML(X_train, y_train, [outputscale - 1e-6, lengthscale, sigma])[0])␣
↪/ 2e-6
ndl = (nlogML(X_train, y_train, [outputscale, lengthscale + 1e-6, sigma])[0]
       - nlogML(X_train, y_train, [outputscale, lengthscale - 1e-6, sigma])[0])␣
↪/ 2e-6
nds = (nlogML(X_train, y_train, [outputscale, lengthscale, sigma + 1e-6])[0]
       - nlogML(X_train, y_train, [outputscale, lengthscale, sigma - 1e-6])[0])␣
↪/ 2e-6
print(f"Numerical partial derivative wrt. outputscale: {ndo:.3f}; Error: {(do -␣
↪ndo):.3f}")
print(f"Numerical partial derivative wrt. lengthscale: {ndl:.3f}; Error: {(dl -␣
↪ndl):.3f}")
print(f"Numerical partial derivative wrt. sigma: {nds:.3f}; Error: {(ds - nds):.
↪3f}")
```

```
ox = np.linspace(-1, 1, 200) + outputscale
nll = np.empty(200)
for i, o in enumerate(ox):
    nll[i] = nlogML(X_train, y_train, [o, lengthscale, sigma])[0]
fig, ax = plt.subplots(figsize=(10, 4))
ax.set_xlabel(r"$o$")
ax.set_ylabel(r"$\log p(y \mid X, \theta)$")
ax.plot(ox, nll)
plt.show()

lx = np.linspace(-0.09, 0.09, 200) + lengthscale
nll = np.empty(200)
for i, l in enumerate(lx):
    nll[i] = nlogML(X_train, y_train, [outputscale, l, sigma])[0]
fig, ax = plt.subplots(figsize=(10, 4))
ax.set_xlabel(r"$\ell$")
ax.set_ylabel(r"$\log p(y \mid X, \theta)$")
ax.plot(lx, nll)
plt.show()

sx = np.linspace(-0.9, 0.9, 200) + sigma
nll = np.empty(200)
for i, s in enumerate(sx):
    nll[i] = nlogML(X_train, y_train, [outputscale, lengthscale, s])[0]
fig, ax = plt.subplots(figsize=(10, 4))
ax.set_xlabel(r"$\sigma$")
ax.set_ylabel(r"$\log p(y \mid X, \theta)$")
ax.plot(sx, nll)
plt.show()
```

```
NLL: 32.607, d/do: -10.881, d/dl: 39.164, d/ds: -9.730
Numerical partial derivative wrt. outputscale: -10.881; Error: -0.000
Numerical partial derivative wrt. lengthscale: 39.164; Error: -0.000
Numerical partial derivative wrt. sigma: -9.730; Error: -0.000
```


55
50
45
40
35
30
25
$\log p(y \mid X, \theta)$
1.00
1.25
1.50
1.75
2.00
2.25
2.50
2.75
3.00
o


42
40
38
36
34
32
$\log p(y \mid X, \theta)$
0.025
0.050
0.075
0.100
0.125
0.150
0.175
ℓ


45.0
42.5
40.0
37.5
35.0
32.5
30.0
27.5
$\log p(y \mid X, \theta)$
0.25
0.50
0.75
1.00
1.25
1.50
1.75
σ

The sign of the gradients gives information about whether the current parameter is too large or too small. In this case, outputscale and sigma are too large, while lengthscale is too small.

Gradient-based Optimizer **Task:** Use SciPy's `scipy.optimize.minimize` to find the optimal hyperparameters for the RBF kernel given the training data using the `L-BFGS-B` optimizer. Initialize with the hyperparameters given above and set a lower bound of 0.01 for each of the parameters.

How do the optimized hyperparameters compare to the ones you found empirically? Is the negative log-marginal likelihood lower?

[12]:
```
from scipy.optimize import minimize

# Define loss via negative log-marginal likelihood and derivative
def loss(theta):
    return nlogML(X_train, y_train, theta)

# Optimize using Quasi-Newton method (with constraints)
theta_opt = minimize(loss, np.r_[2.0, 0.1, 1.0], method='L-BFGS-B',
                     jac=True, bounds=[(0.01, None) for i in range(3)]).x
```

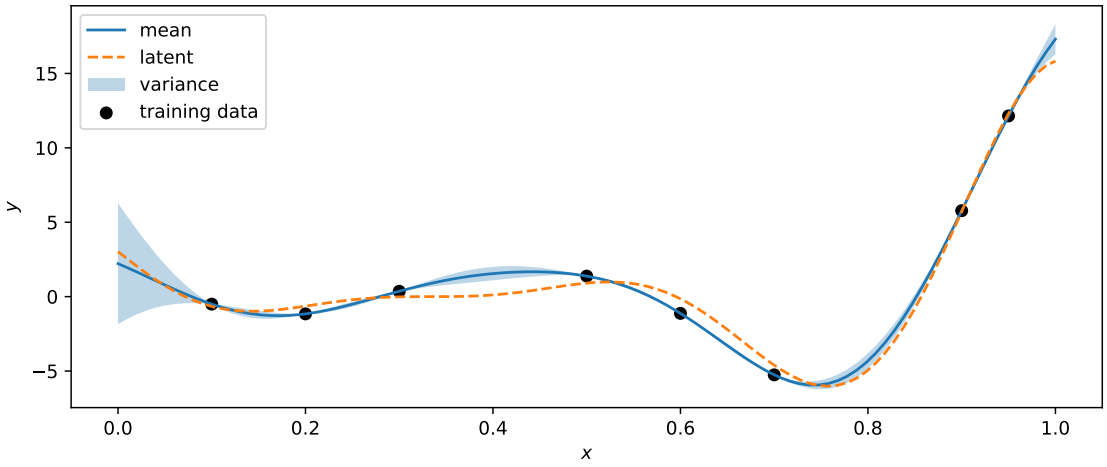
Task: Plot the GP with the optimized hyperparameters.

[13]:
```
# Mean and covariance with hyperparameter choice
mu = lambda x: np.zeros(shape=(x.shape[0], 1))
cov = lambda x0, x1=None: expquad(x0=x0, x1=x1, outputscale=theta_opt[0],␣
 ↪lengthscale=theta_opt[1])

# Define Gaussian process
g = GaussianProcess(mu, cov, sigma=theta_opt[2])
g.fit(X_train, y_train)

# Prediction
y_pred = g.predict(X_test)

# Negative log-marginal likelihood
print(f"Negative Log-Marginal Likelihood: {-g.log_marginal_likelihood()}")
print(f"Outputscale: {theta_opt[0]}")
print(f"Lengthscale: {theta_opt[1]}")
print(f"Noise: {theta_opt[2]}")
```

```
Negative Log-Marginal Likelihood: 22.577803546751433
Outputscale: 9.84549857238099
Lengthscale: 0.18411262492685923
Noise: 0.01
```

[14]:
```
mu = lambda x: np.zeros(shape=(x.shape[0], 1))
```

```
cov = lambda x0, x1=None: expquad(x0=x0, x1=x1, outputscale=theta_opt[0],␣
 ↪lengthscale=theta_opt[1])
gp = GaussianProcess(mu, cov, sigma=theta_opt[2])
gp.fit(X_train, y_train)
y_pred = gp.predict(X_test)

plot_gp(X_test, y_pred, traindata=(X_train, y_train), latentfun=f)
```



2.2.2 Automatic Differentiation

Modern automatic differentiation frameworks have made it very simple to optimize hyperparameters of machine learning models since only the model itself and the loss function (in our case the log-marginal likelihood) have to be defined. The derivative of the loss is then computed via automatic differentiation, which in simple terms applies the chain rule to the implementation of the loss. To see how convenient automatic differentiation can be install GPyTorch via `pip install gpytorch` and run the code segment below.

Note: Should you have trouble installing ***`gpytorch`*** *on your system, you may skip this part without losing credit.*

[15]:
```
import torch
import gpytorch
torch.set_default_dtype(torch.double)

# Training data
train_x = torch.from_numpy(X_train.ravel())
train_y = torch.from_numpy(y_train.ravel())
test_x = torch.from_numpy(X_test.ravel())
```

[16]:
```
# Define the GP model
class ExactGPModel(gpytorch.models.ExactGP):
```

```
    def __init__(self, train_x, train_y, likelihood):
        super(ExactGPModel, self).__init__(train_x, train_y, likelihood)
        self.mean = gpytorch.means.ConstantMean()
        self.cov = gpytorch.kernels.ScaleKernel(gpytorch.kernels.RBFKernel())

    def forward(self, x):
        return gpytorch.distributions.MultivariateNormal(self.mean(x), self.
 ↪cov(x))

likelihood = gpytorch.likelihoods.GaussianLikelihood(
    noise_constraint=gpytorch.constraints.GreaterThan(10 ** -2)
)
model = ExactGPModel(train_x, train_y, likelihood)
```

[17]:
```
# Initialize hyperparameters
outputscale = 2.0
lengthscale = 0.1
sigma = 1.0
model.cov.initialize(outputscale=outputscale)
model.cov.base_kernel.initialize(lengthscale=lengthscale)
likelihood.initialize(noise=sigma ** 2)

# Training mode
model.train()
likelihood.train()

# Define optimizer and loss
optimizer = torch.optim.LBFGS(model.parameters())
mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)

# Training loop
training_iter = 100
for i in range(training_iter):
    def closure():
        loss = -mll(model(train_x), train_y)
        optimizer.zero_grad()
        loss.backward()
        return loss

    optimizer.step(closure)
```

Task: Compare the negative log-marginal likelihood and hyperparameters identified by `gpytorch` to the ones found with your approach. Are they different? If yes, what could explain the difference and which approach did better and why?

[18]:
```
print(f"Negative Log-Marginal Likelihood: {-mll(model(train_x), train_y)}")
print(f"Outputscale: {model.cov.outputscale.item()}")
print(f"Lengthscale: {model.cov.base_kernel.lengthscale.item()}")
print(f"Noise: {model.likelihood.noise.item()}")
```

```
Negative Log-Marginal Likelihood: 2.770948521461129
Outputscale: 86.73641281322871
Lengthscale: 0.18398183591848336
Noise: 0.010000006157628544
```

[19]:
```
# Evaluation (predictive posterior) mode
model.eval()
likelihood.eval()

# Predict
with torch.no_grad():
    y_pred = likelihood(model(test_x))
```

[20]:
```
plot_gp(
    X_test,
    (y_pred.mean.numpy(), y_pred.lazy_covariance_matrix.evaluate().numpy()),
    traindata=(X_train, y_train),
    latentfun=f,
)
```