GP - Rasmussen & Williams - Ch. 2: Regression

Outline

GP prior

$$k(x, y) = \exp(-\tfrac{1}{2}|x - y|^2) \quad (1)$$

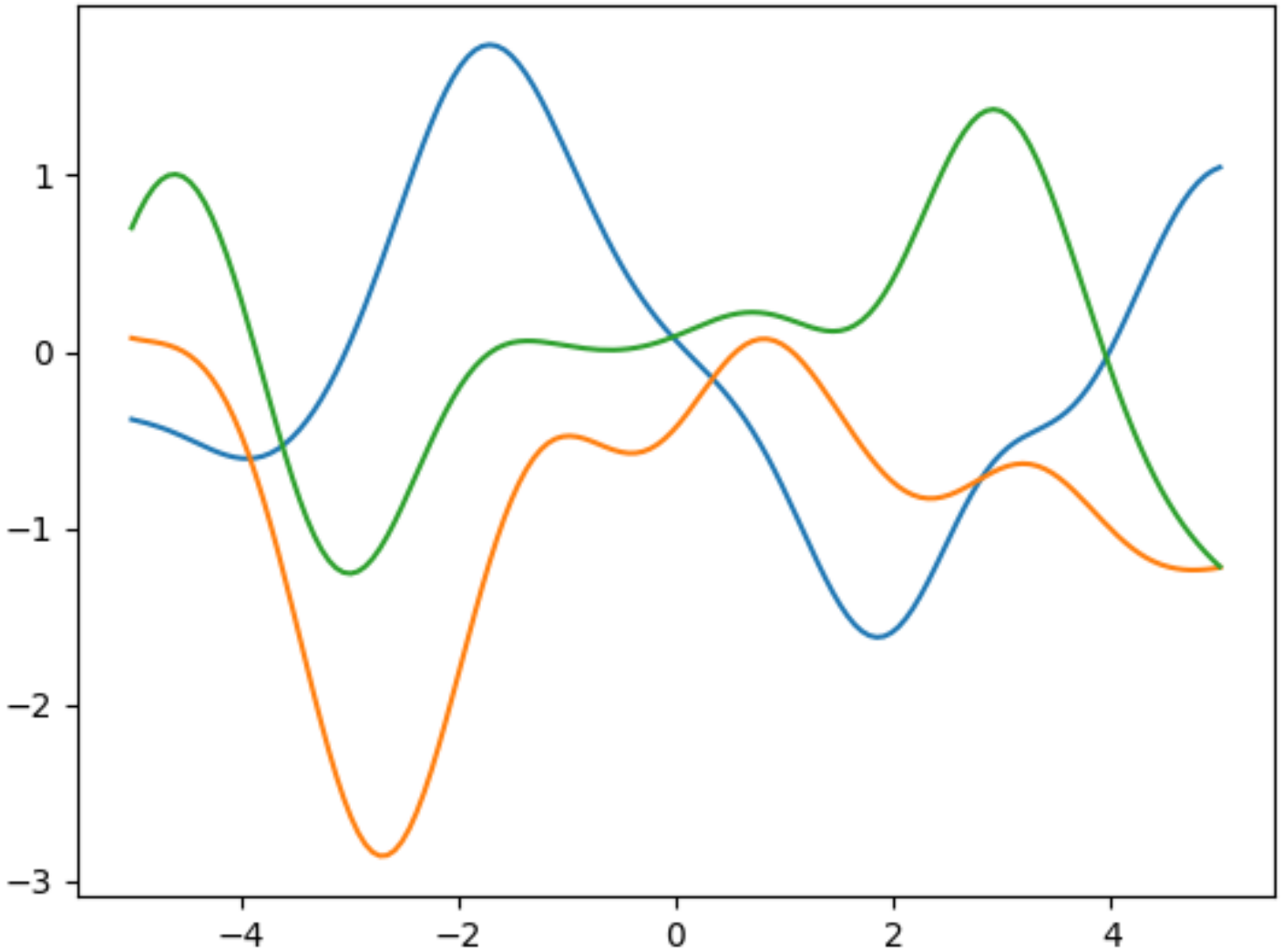
$$\mathbf{f} \sim \mathcal{N}(\mathbf{0}, K(\mathbf{x}, \mathbf{x})) \quad (2)$$

Sampling from prior: Python code

```
from numpy import sum, eye, exp #, zeros
from numpy.linalg import cholesky
from numpy.random import normal #, multivariate_normal

def rbf(length_scale):
    def k(x,y):
        if len(x.shape)==1:
            d = 1
        else:
            d  = x.shape[1]
        lx = x.shape[0]
        ly = y.shape[0]
        dists =  sum(((x.T.reshape([d,lx,1]) -  y.T.reshape([d,1,ly]))/length_scale)**2,0)
        return exp(-.5 * dists)
    return k

def genSamplesSimple(x, k):
    n = x.shape[0]
    L = cholesky(k(x,x)+eye(n)*1e-8)
    return L.dot(normal(size=n))
# Same as:
#    return multivariate_normal(zeros(n), k(x,x) + eye(n)*1e-8)
```

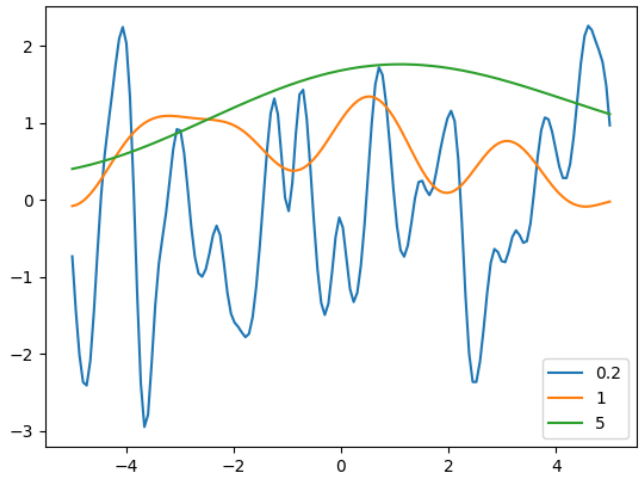
```
from matplotlib.pyplot import figure, plot, savefig, close, legend
from numpy import linspace

figure()
x = linspace(-5,5,150)
k = rbf(1)
for i in range(3): plot(x, genSamplesSimple(x,k));
```

Random functions in 1D



Different length scales

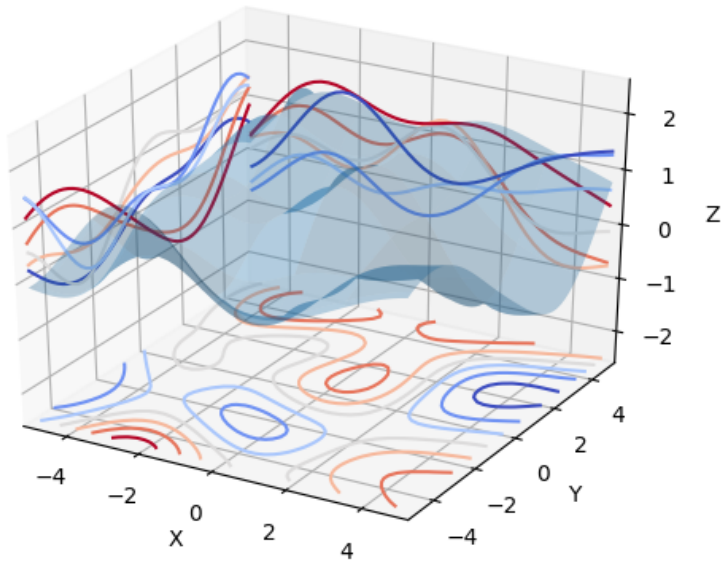
```
scales = [0.2, 1, 5]
for i in scales:
  plot(x, genSamplesSimple(x,rbf(i)))
legend(scales)
```



Two dimensions

```
from numpy import meshgrid, concatenate

x = linspace(5, -5, 50)
xx, yy = meshgrid(x, x)
xy = concatenate([xx.reshape([1, -1]),
                  yy.reshape([1, -1])]).T
z = genSamplesSimple(xy, rbf(2)).reshape([50, 50]
```

```
fig = figure()
ax = fig.gca(projection='3d')
ax.plot_surface(xx, yy, z, rstride=8,
                cstride=8, alpha=0.3)
cset = ax.contour(xx, yy, z, zdir='z',
                  offset=-2.5, cmap=cm.coolwarm)
cset = ax.contour(xx, yy, z, zdir='x',
                  offset=-5, cmap=cm.coolwarm)
cset = ax.contour(xx, yy, z, zdir='y',
                  offset=5, cmap=cm.coolwarm)

ax.set_xlabel('X')
ax.set_xlim(-5, 5)
ax.set_ylabel('Y')
ax.set_ylim(-5, 5)
ax.set_zlabel('Z')
ax.set_zlim(-2.5, 2.5)
```



Computing posterior

$$
\begin{aligned}
\begin{bmatrix} \mathbf{y} \\ \mathbf{f}_* \end{bmatrix} &\sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} K(X,X)+\sigma^2 I & K(X,X_*) \\ K(X_*,X) & K(X_*,X_*) \end{bmatrix}\right) \\
\mathbf{f}_*|X,\mathbf{y},X_* &\sim \mathcal{N}(\bar{\mathbf{f}}_*, \mathrm{cov}(\mathbf{f}_*)) \\
\bar{\mathbf{f}}_* &= K(X_*,X)[K(X,X)+\sigma^2 I]^{-1}\mathbf{y} \\
&= K(X_*,X)\alpha \\
\mathrm{cov}(\mathbf{f}_*) &= K(X_*,X_*) - K(X_*,X)[K(X,X)+\sigma^2 I]^{-1}K(X,X_*) \\
&= K(X_*,K_*) - V^T V
\end{aligned}
$$

Where

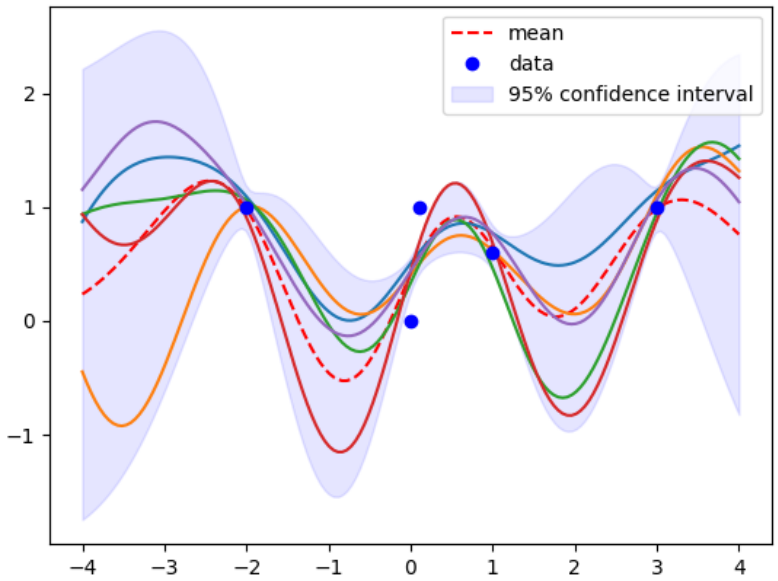
$$
\begin{aligned}
L &= \mathrm{chol}(K(X,X)+\sigma^2 I) \rightarrow LL^T = K(X,X)+\sigma^2 I \\
\alpha &= [K(X,X)+\sigma^2 I]^{-1}\mathbf{y} = L^{-T}L^{-1}\mathbf{y} \\
V &= L^{-1}K(X,X_*)
\end{aligned}
$$

Python code

```
from numpy import pi, eye, log, diag
from numpy.random import normal
from numpy.linalg import cholesky, solve #, inv
# solve(A,b) equals inv(A)*v, but it is more robust

def compPosterior(y, x, k, X, snoise):
    n = x.shape[0]
    K = k(x,X)
    L = cholesky(k(x, x) + eye(n)*(snoise + 1e-8))
    alpha = solve(L.T,solve(L,y))
    f_mean = K.T.dot(alpha)
    v = solve(L,K)
    V = k(X,X) - v.T.dot(v)
    log_p = -.5*y.T.dot(alpha) - sum(log(diag(L))) - .5*n*log(2*pi)
    return f_mean, V, log_p

def genSamples(x, m, K):
    n = x.shape[0]
    L = cholesky(K+eye(n)*1e-8)
    return m + L.dot(normal(size=n))
```

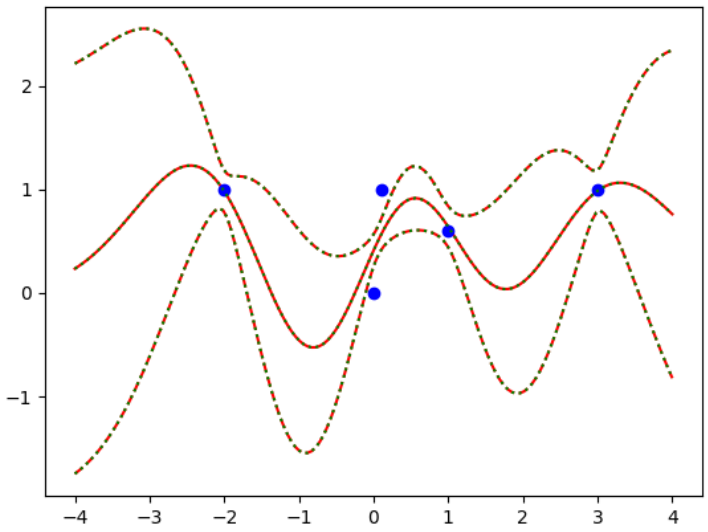
Fitting some data

```
from numpy import array, sqrt

x = array([-2, 0, 0.1,   1, 3])
y = array([ 1, 0,   1, 0.6, 1])
X = linspace(-4, 4, 150)

k = rbf(1)
f_m, V, _ = compPosterior(y, x, k, X, 0.01)

s = sqrt(diag(V))

plot(X, f_m, '--r', label='mean')
fill_between(X, f_m - 2*s, f_m + 2*s, color =
       alpha = 0.1, label='95% confidence inte
for i in range(5):
  plot(X, genSamples(X, f_m, V))
plot(x, y, 'ob', label='data')
legend()
```



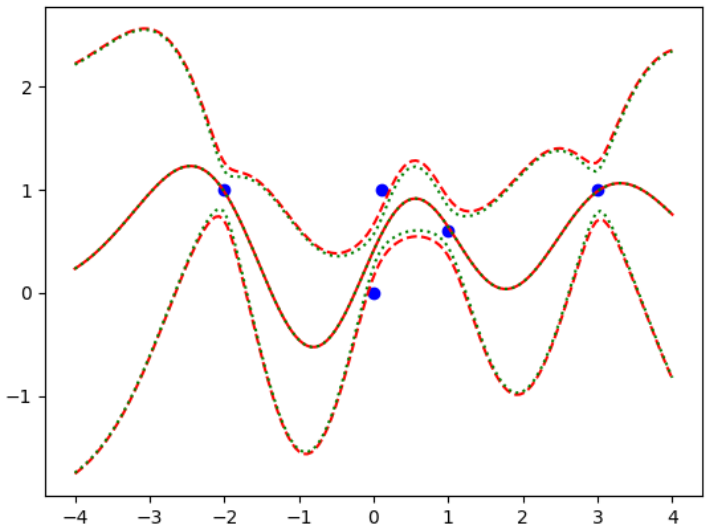
scikit-learn

```
from sklearn.gaussian_process import GaussianProcessRegressor as GPR
from sklearn.gaussian_process.kernels import RBF
      # other kernels: Matern, WhiteKernel, ConstantKernel

k = RBF(1) # exactly the same behaviour as before
# but many kernels can be combined more easily:
# ConstantKernel() + Matern(length_scale=2, nu=3/2)
#     + WhiteKernel(noise_level=1)

x = x.reshape([-1,1])  # but now x must be a 2D array
X = X.reshape([-1,1])

gp = GPR(alpha = 0.01, kernel=k, optimizer = None)
# it has many more options, in particular, optimizer
# must be set to None to prevent ML kernel estimation

gp.fit(x, y)

y_pred, sigma = gp.predict(X, return_std=True)

plot(x,y,'ob')
plot(X, y_pred, 'r')
plot(X, y_pred + 2*sigma, 'r--'); plot(X, y_pred - 2*sigma, 'r--')

# same results as raw python code
plot(X, f_m, 'g:')
plot(X, f_m + 2*s, 'g:'); plot(X, f_m - 2*s, 'g:')
```



GPFlow

```
import gpflow

Y = y.reshape(-1,1)

k = gpflow.kernels.RBF(1)

m = gpflow.models.gpr(x, Y, kern=k)

m.likelihood.variance = 0.01

f, sgpf =  m.predict_y(X)


plot(x,y,'ob')
plot(X, f, 'r')
plot(X, f + 2*sqrt(sgpf), 'r--'); plot(X, f - 2*sqrt(s

# similar (not identical) to previous results
plot(X, f_m, 'g:')
plot(X, f_m + 2*s, 'g:'); plot(X, f_m - 2*s, 'g:')
```



PyMC3

Edward

PyStan

Sampling from prior: Haskell code

```haskell
import Numeric.LinearAlgebra

type DVector = Vector Double
type DMatrix = Matrix Double

type CovarianceMatrix = Matrix Double
type Kernel = DVector -> DVector -> CovarianceMatrix

rbf :: Double -> Kernel
rbf k x y = exp $ -((asColumn x - asRow y) / scalar k )^2

genSamplesSimple :: DVector -> Kernel -> IO DVector
genSamplesSimple x k = do
  r <- randn n 1 -- also see multivariate normal: gaussianSample
  return $ flatten (tr l <> r)
  where
    n = size x
    l = chol . sym $ k x x + ident n * 1e-8
```

```haskell
import Graphics.Matplotlib

let x =  linspace 150 (-5, 5) :: DVector
let k = rbf 1

y <- sequence [genSamplesSimple x k | i <- [1..3]]
file name $ plot x (y!!0) % plot x (y!!1) % plot x (y!!2)
```

Random functions in 1D