sheet10_prog_johan

January 14, 2019

1 Gaussian Processes

In this exercise, you will implement Gaussian process regression and apply it to a toy and a real dataset. We use the notation used in the paper "Rasmussen (2005). Gaussian Processes in Machine Learning" linked on ISIS.

Let us first draw a training set $X = (x_1, \ldots, x_n)$ and a test set $X_\star = (x_1^\star, \ldots, x_m^\star)$ from a d-dimensional input distribution. The Gaussian Process is a model under which the real-valued outputs $\mathbf{f} = (f_1, \ldots, f_n)$ and $\mathbf{f}_\star = (f_1^\star, \ldots, f_m^\star)$ associated to X and $X_\star$ follow the Gaussian distribution:

$$\begin{bmatrix} \mathbf{f} \\ \mathbf{f}_\star \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \begin{bmatrix} \Sigma & \Sigma_\star \\ \Sigma_\star^\top & \Sigma_{\star\star} \end{bmatrix} \right)$$

where

$$\begin{aligned} \Sigma &= k(X, X) + \sigma^2 I \\ \Sigma_\star &= k(X, X_\star) \\ \Sigma_{\star\star} &= k(X_\star, X_\star) + \sigma^2 I \end{aligned}$$

and where $k(\cdot, \cdot)$ is the Gaussian kernel function. (The kernel function is implemented in `utils.py`.) Predicting the output for new data points $X_\star$ is achieved by conditioning the joint probability distribution on the training set. Such conditional distribution called posterior distribution can be written as:

$$\mathbf{f}_\star | \mathbf{f} \sim \mathcal{N}(\underbrace{\Sigma_\star^\top \Sigma^{-1} \mathbf{f}}_{\mu_\star} \ , \ \underbrace{\Sigma_{\star\star} - \Sigma_\star^\top \Sigma^{-1} \Sigma_\star}_{C_\star}) \tag{1}$$

Having inferred the posterior distribution, the log-likelihood of observing for the inputs $X_\star$ the outputs $\mathbf{y}_\star$ is given by evaluating the distribution $\mathbf{f}_\star | \mathbf{f}$ at $\mathbf{y}_\star$:

$$\log p(\mathbf{y}_\star | \mathbf{f}) = -\frac{1}{2}(\mathbf{y}_\star - \boldsymbol{\mu}_\star)^\top C_\star^{-1} (\mathbf{y}_\star - \boldsymbol{\mu}_\star) - \frac{1}{2} \log |C_\star| - \frac{m}{2} \log 2\pi \tag{2}$$

where $|\cdot|$ is the determinant. Note that the likelihood of the data given this posterior distribution can be measured both for the training data and the test data.

1.1 Part 1: Implementing a Gaussian Process (30 P)

Tasks:

- **Create a class `GP_Regressor` that implements a Gaussian process regressor and has the following three methods:**
- `def __init__(self,Xtrain,Ytrain,width,noise):` Initialize a Gaussian process with noise parameter σ and width parameter w. The function must also precompute the matrix Σ^{-1} for subsequent use by the method `predict()` and `loglikelihood()`.
- `def predict(self,Xtest):` For the test set $X_\star$ of m points received as parameter, return the mean vector of size m and covariance matrix of size $m \times m$ of the corresponding output, that is, return the parameters $(\boldsymbol{\mu}_\star, C_\star)$ of the Gaussian distribution $\mathbf{f}_\star|\mathbf{f}$.
- `def loglikelihood(self,Xtest,Ytest):` For a data set $X_\star$ of m test points received as first parameter, return the loglikelihood of observing the outputs $\mathbf{y}_\star$ received as second parameter.

In [23]:
```
import numpy as np
import utils

class GP_Regressor:

    def __init__(self, Xtrain, Ytrain, width, noise):
        self.X = Xtrain
        self.Y = Ytrain
        self.width = width
        self.noise = noise
        self.sigma = utils.gaussianKernel(Xtrain, Xtrain, width)
        self.sigma += self.noise**2*np.eye(*self.sigma.shape)
        self.sigma_inv = np.linalg.inv(self.sigma)

    def mu(self, X, x, Y = None):
        if Y is None: Y = self.Y
        sigma_ = utils.gaussianKernel(X, x, self.width)
        mu = sigma_.T.dot(self.sigma_inv).dot(Y)

        return mu

    def C(self, X, x):

        sigma_ = utils.gaussianKernel(X, x, self.width)
        sigma__ = utils.gaussianKernel(x, x, self.width)
        sigma__ += self.noise**2*np.eye(*sigma__.shape)
        C_ = sigma__ - sigma_.T.dot(self.sigma_inv).dot(sigma_)

        return C_

    def predict(self, Xtest):
```

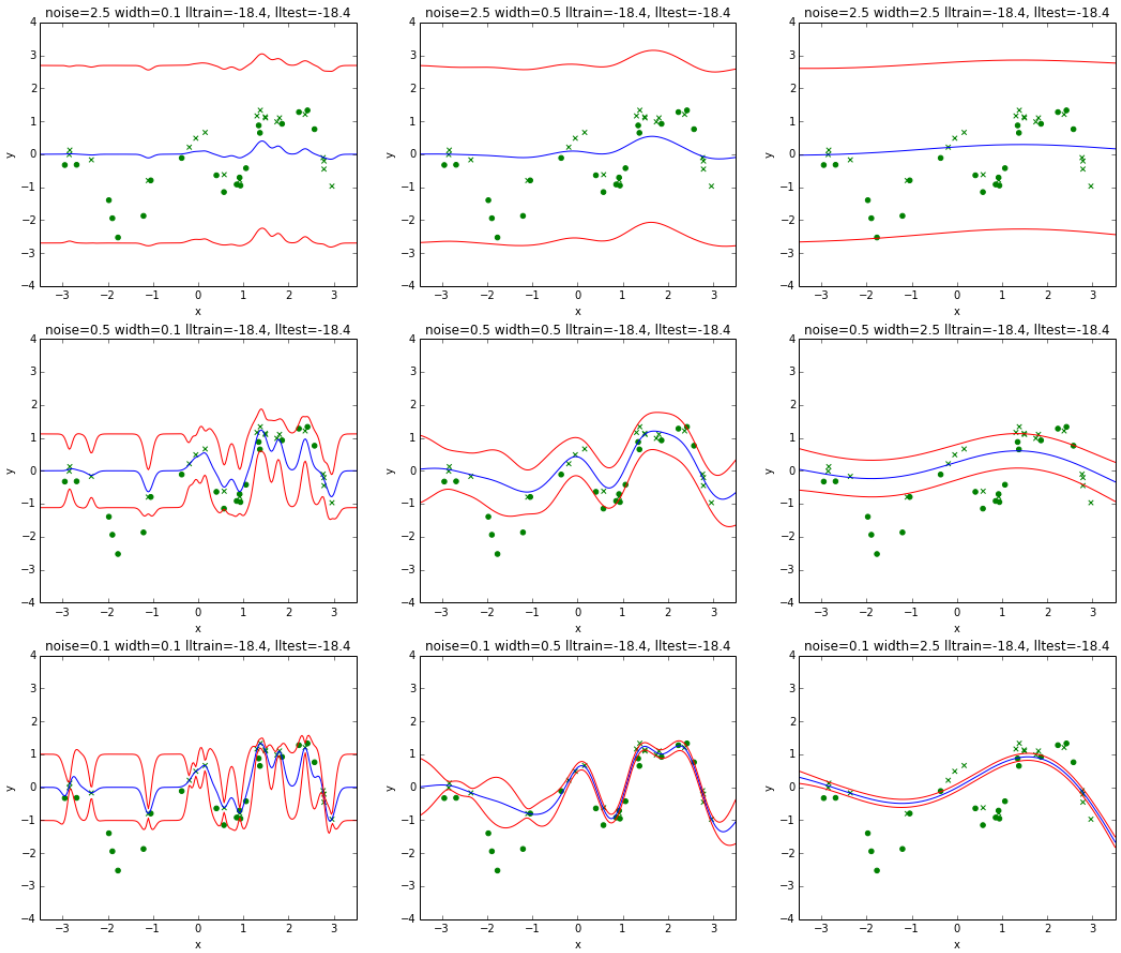
```
            return self.mu(self.X, Xtest), self.C(self.X, Xtest)

        def loglikelihood(self, Xtest, Ytest):
            x = Xtest
            m = Ytest.shape[0]
            mu, C = self.predict(x)
            Cinv = np.linalg.inv(C)
            Cdet = np.abs(np.linalg.det(C)) or 1e-9
            y_ = Ytest
            distance = y_ - mu

            first = -(1/2)*distance.T.dot(Cinv).dot(distance)
            second = -(1/2)*np.log(Cdet) - (m/2)*np.log(2*np.pi)

            return first+second
```

- **Test your implementation by running the code below (it visualizes the mean and variance of the prediction at every location of the input space) and compares the behavior of the Gaussian process for various noise parameters σ and width parameters w.**

In [24]:
```
import datasets,numpy
import matplotlib.pyplot as plt
%matplotlib inline

# Open the toy data
Xtrain,Ytrain,Xtest,Ytest = utils.split(*datasets.toy())
print("Xtrain,Ytrain,Xtest,Ytest", Xtrain.shape,Ytrain.shape,Xtest.shape,Ytest.shape)

# Create an analysis distribution
Xrange = numpy.arange(-3.5,3.51,0.025)[:,numpy.newaxis]

f = plt.figure(figsize=(18,15))

# Loop over several parameters:
for i,noise in enumerate([2.5,0.5,0.1]):
    for j,width in enumerate([0.1,0.5,2.5]):

        # Create Gaussian process regressor object
        gp = GP_Regressor(Xtrain, Ytrain, width, noise)

        # Compute the predicted mean and variance for test data
        mean,cov = gp.predict(Xrange)
        #print("mean,cov",mean.shape,cov.shape)
        var = cov.diagonal()
        #print("var", var.shape)

        # Compute the log-likelihood of training and test data
```

```
            lltrain = gp.loglikelihood(Xtrain, Ytrain)
            lltest  = gp.loglikelihood(Xtest, Ytest)

            # Plot the data
            p = f.add_subplot(3,3,3*i+j+1)
            p.set_title('noise=%.1f width=%.1f lltrain=%.1f, lltest=%.1f'%(noise,width,lltr
            p.set_xlabel('x')
            p.set_ylabel('y')
            p.scatter(Xtrain,Ytrain,color='green',marker='x') # training data
            p.scatter(Xtest,Ytest,color='green',marker='o')   # test data
            p.plot(Xrange,mean,color='blue')                  # GP mean
            p.plot(Xrange,mean+var**.5,color='red')           # GP mean + std
            p.plot(Xrange,mean-var**.5,color='red')           # GP mean - std
            p.set_xlim(-3.5,3.5)
            p.set_ylim(-4,4)
```

```
('Xtrain,Ytrain,Xtest,Ytest', (20, 1), (20,), (20, 1), (20,))
```



##Part 2: Application to the Yacht Hydrodynamics Data Set (10 P)

In the second part, we would like to apply the Gaussian process regressor that you have implemented to a real dataset: the Yacht Hydrodynamics Data Set available on the UCI repository at the webpage http://archive.ics.uci.edu/ml/datasets/Yacht+Hydrodynamics. As stated on the web page, the input variables for this regression problem are:

1. Longitudinal position of the center of buoyancy
2. Prismatic coefficient
3. Length-displacement ratio
4. Beam-draught ratio
5. Length-beam ratio
6. Froude number

and we would like to predict from these variables the residuary resistance per unit weight of displacement (last column in the file `yacht_hydrodynamics.data`).

Tasks:

- **Load the data using `datasets.yacht()` and partition the data between training and test set using the function `utils.split()`. Normalize the data (center and rescale) so that the training data and labels have mean 0 and standard deviation 1 over the dataset for each dimension.**
- **Train several Gaussian processes on the regression task using various width and noise parameters.**
- **Draw two contour plots where the training and test log-likelihood are plotted as a function of the noise and width parameters. Choose suitable ranges of parameters so that the best parameter combination for the test set is in the plot. Use the same ranges and contour levels for training and test plots.**

```
In [25]: import utils, datasets, numpy as np
         %matplotlib inline

         xtemp,ytemp = datasets.yacht()
         Xtrain, Ytrain, Xtest, Ytest = utils.split(xtemp, ytemp)

         #delete last element for successful alignment
         Xtrain -= np.mean(Xtrain, axis = 0)
         Xtrain /= Xtrain.std(axis=0)
         Xtest -= np.mean(Xtest, axis = 0)
         Xtest /= Xtest.std(axis=0)
         Ytrain -= np.mean(Ytrain, axis = 0)
         Ytrain /= Ytrain.std(axis=0)
         Ytest -= np.mean(Ytest, axis = 0)
         Ytest /= Ytest.std(axis=0)
         #np.set_printoptions(formatter={'float_kind':lambda x: "%.3f" % x})
         print("Xtrain,Ytrain,Xtest,Ytest", Xtrain.shape,Ytrain.shape,Xtest.shape,Ytest.shape)

         print("Xtrain.mean(axis=0): " + str(Xtrain.mean(axis=0)))
```

```
        print("Xtest.mean(axis=0): " + str(Xtest.mean(axis=0)) + "\r\n")
        print("Xtrain.std(axis=0): " + str(Xtrain.std(axis=0)))
        print("Xtest.std(axis=0): " + str(Xtest.std(axis=0)) + "\r\n")

        print("Ytrain.mean(axis=0): " + str(Ytrain.mean(axis=0)))
        print("Ytest.mean(axis=0): " + str(Ytest.mean(axis=0)) + "\r\n")
        print("Ytrain.std(axis=0): " + str(Ytrain.std(axis=0)))
        print("Ytest.std(axis=0): " + str(Ytest.std(axis=0)) + "\r\n")

        #plot the function
        noise = np.arange(0.005,.04, .005)#0.002)
        width = np.arange(0.05, 2, .1)#0.02)

        print("Noise params: " + str(noise))
        print("Width params: " + str(width))
        #print(Xtrain)
        lltrains = []
        lltests = []
        for i, n in enumerate(noise):
            for j, w in enumerate(width):
                gp = GP_Regressor(Xtrain,Ytrain,w,n)
                # Compute the predicted mean and variance for test data
                #print(gp.sigma)
                mean,cov = gp.predict(Xtrain)
                #print("mean,cov",mean,cov)
                var = cov.diagonal()
                #print(mean,cov)

                # Compute the log-likelihood of training and test data
                lltrain = gp.loglikelihood(Xtrain, Ytrain)
                lltest  = gp.loglikelihood(Xtest, Ytest)
                lltrains.append(lltrain)
                lltests.append(lltest)

        X, Y = np.meshgrid(width, noise)
        plt.contour(X, Y, np.array(lltrains).reshape(X.shape))
        plt.plot()
        plt.contour(X, Y, np.array(lltests).reshape(X.shape))
        plt.plot()
        noise
```

```
('Xtrain,Ytrain,Xtest,Ytest', (153, 6), (153,), (154, 6), (154,))
Xtrain.mean(axis=0): [ 1.34532908e-15  2.54115447e-14  2.11304286e-14 -3.55425566e-15
  1.90914822e-14 -2.55423859e-16]
Xtest.mean(axis=0): [ 6.87964296e-16  1.16663533e-14  2.69034434e-14 -4.21452195e-15
  2.40453400e-14  1.36398829e-15]

Xtrain.std(axis=0): [1. 1. 1. 1. 1. 1.]
```

```
Xtest.std(axis=0): [1. 1. 1. 1. 1. 1.]

Ytrain.mean(axis=0): -5.805087710458335e-17
Ytest.mean(axis=0): 5.767392335715098e-17

Ytrain.std(axis=0): 1.0
Ytest.std(axis=0): 1.0

Noise params: [0.005 0.01  0.015 0.02  0.025 0.03  0.035 0.04 ]
Width params: [0.05 0.15 0.25 0.35 0.45 0.55 0.65 0.75 0.85 0.95 1.05 1.15 1.25 1.35
 1.45 1.55 1.65 1.75 1.85 1.95]
```

Out[25]: array([0.005, 0.01 , 0.015, 0.02 , 0.025, 0.03 , 0.035, 0.04])