Pract_regression_professor

September 19, 2019

1 Gaussian Process regression

```
Authors: Miguel Lázaro Gredilla
         Jerónimo Arenas García (jarenas@tsc.uc3m.es)
         Jesús Cid Sueiro

Notebook version: 1.0 (Nov, 07, 2017)

Changes: v.1.0 - First version. Python version
         v.1.1 - Extraction from a longer version ingluding Bayesian regresssion.
                 Python 3 compatibility

Pending changes:
```

[ ]:
```
# Import some libraries that will be necessary for working with data and␣
 ↪displaying plots

# To visualize plots in the notebook
%matplotlib inline

import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import scipy.io       # To read matlab files
from scipy import spatial
import pylab
pylab.rcParams['figure.figsize'] = 8, 5
```

2 1. Introduction

In this exercise the student will review several key concepts of Bayesian regression and Gaussian processes.

For the purpose of this exercise, the regression model is

$$s(\mathbf{x}) = f(\mathbf{x}) + \varepsilon$$

where $s(\mathbf{x})$ is the output corresponding to input $\mathbf{x}$, $f(\mathbf{x})$ is the unobservable latent function, and ε is white zero-mean Gaussian noise, i.e., $\varepsilon \sim \mathcal{N}(0, \sigma_\varepsilon^2)$.

2.0.1 Practical considerations

- Though sometimes unavoidable, it is recommended not to use explicit matrix inversion whenever possible. For instance, if an operation like $\mathbf{A}^{-1}\mathbf{b}$ must be performed, it is preferable to code it using python numpy.linalg.lstsq function (see http://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.lstsq.html), which provides the LS solution to the overdetermined system $\mathbf{A}\mathbf{w} = \mathbf{b}$.

- Sometimes, the computation of $\log|\mathbf{A}|$ (where $\mathbf{A}$ is a positive definite matrix) can overflow available precision, producing incorrect results. A numerically more stable alternative, providing the same result is $2\sum_i \log([\mathbf{L}]_{ii})$, where $\mathbf{L}$ is the Cholesky decomposition of $\mathbf{A}$ (i.e., $\mathbf{A} = \mathbf{L}^\top\mathbf{L}$), and $[\mathbf{L}]_{ii}$ is the ith element of the diagonal of $\mathbf{L}$.

- Non-degenerate covariance matrices, such as the ones in this exercise, are always positive definite. It may happen, as a consequence of chained rounding errors, that a matrix which was mathematically expected to be positive definite, turns out not to be so. This implies its Cholesky decomposition will not be available. A quick way to palliate this problem is by adding a small number (such as 10^{-6}) to the diagonal of such matrix.

2.0.2 Reproducibility of computations

To guarantee the exact reproducibility of the experiments, it may be useful to start your code initializing the seed of the random numbers generator, so that you can compare your results with the ones given in this notebook.

[ ]:
```
np.random.seed(3)
```

3 2. The stocks dataset.

Load and properly normalize data corresponding to the evolution of the stocks of 10 airline companies. This data set is an adaptation of the Stock dataset from http://www.dcc.fc.up.pt/~ltorgo/Regression/DataSets.html, which in turn was taken from the StatLib Repository, http://lib.stat.cmu.edu/

[ ]:
```
# Load data from matlab file DatosLabReg.mat
# matvar = <FILL IN>
matvar = scipy.io.loadmat('DatosLabReg.mat')

# Take main variables, Xtrain, Xtest, Ytrain, Ytest from the corresponding␣
 ↪dictionary entries in matvar:
#ă<SOL>
Xtrain = matvar['Xtrain']
Xtest = matvar['Xtest']
Ytrain = matvar['Ytrain']
Ytest = matvar['Ytest']
# </SOL>

# Data normalization
# <SOL>
```

```
mean_x = np.mean(Xtrain,axis=0)
std_x = np.std(Xtrain,axis=0)
Xtrain = (Xtrain - mean_x) / std_x
Xtest = (Xtest - mean_x) / std_x
# </SOL>
```

After running this code, you will have inside matrix `Xtrain` the evolution of (normalized) price for 9 airlines, whereas vector `Ytrain` will contain a single column with the price evolution of the tenth airline. The objective of the regression task is to estimate the price of the tenth airline from the prices of the other nine.

4 3. Non-linear regression with Gaussian Processes

4.1 3.1. Multidimensional regression

Rather than using a parametric form for $f(\mathbf{x})$, in this section we will use directly the values of the latent function that we will model with a Gaussian process

$$f(\mathbf{x}) \sim \mathcal{GP}\left(0, k_f(\mathbf{x}_i, \mathbf{x}_j)\right),$$

where we are assuming a zero mean, and where we will use the Ornstein-Uhlenbeck covariance function, which is defined as:

$$k_f(\mathbf{x}_i, \mathbf{x}_j) = \sigma_0^2 \exp\left(-\frac{1}{l}\|\mathbf{x}_i - \mathbf{x}_j\|\right)$$

First, we will use the following gross estimation for the hyperparameters:

[ ]:
```
sigma_0 = np.std(Ytrain)
sigma_eps = sigma_0 / np.sqrt(10)
l = 8

print('sigma_0 = {0}'.format(sigma_0))
print('sigma_eps = {0}'.format(sigma_eps))
```

As we studied in a previous session, the joint distribution of the target values in the training set, $\mathbf{s}$, and the latent values corresponding to the test points, $\mathbf{f}^*$, is given by

$$\begin{bmatrix} \mathbf{s} \\ \mathbf{f}^* \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} \mathbf{K} + \sigma_\varepsilon^2 \mathbf{I} & \mathbf{K}_*^\top \\ \mathbf{K}_* & \mathbf{K}_{**} \end{bmatrix}\right)$$

Using this model, obtain the posterior of $\mathbf{s}^*$ given $\mathbf{s}$. In particular, calculate the a posteriori predictive mean and standard deviations, $\mathbb{E}\{s(\mathbf{x}^*) \mid \mathbf{s}\}$ and $\sqrt{\mathbb{V}\{s(\mathbf{x}^*) \mid \mathbf{s}\}}$ for each test sample $\mathbf{x}^*$.

Obtain the MSE and NLPD.

[ ]:
```
# Compute Kernel matrices.
# You may find spatial.distance.cdist() usefull to compute the euclidean␣
 ↪distances required by Gaussian kernels.
# <SOL>
# Compute appropriate distances
dist = spatial.distance.cdist(Xtrain, Xtrain, 'euclidean')
```

```
dist_ss = spatial.distance.cdist(Xtest, Xtest, 'euclidean')
dist_s = spatial.distance.cdist(Xtest, Xtrain, 'euclidean')

# Compute Kernel matrices
K = (sigma_0**2)*np.exp(-dist/l)
K_ss = (sigma_0**2)*np.exp(-dist_ss/l)
K_s = (sigma_0**2)*np.exp(-dist_s/l)
# </SOL>

# Compute predictive mean
# m_y = <FILL IN>
m_y = K_s.dot(np.linalg.inv(K + sigma_eps**2 * np.eye(K.shape[0]))).dot((Ytrain))

# Compute predictive variance
# v_y = <FILL IN>
v_y = np.diagonal(K_ss - K_s.dot(np.linalg.inv(K + sigma_eps**2 * np.eye(K.
 ↪shape[0]))).dot(K_s.T)) + sigma_eps**2

# Compute MSE
# MSE = <FILL IN>
MSE = np.mean((m_y - Ytest)**2)

# Compute NLPD
# NLPD = <FILL IN>
NLPD = 0.5 * np.mean(((Ytest - m_y)**2)/(np.matrix(v_y).T) + 0.5*np.log(2*np.
 ↪pi*np.matrix(v_y).T))

print(m_y.T)
```

You should obtain the following results:

[ ]:
```
print('MSE = {0}'.format(MSE))
print('NLPD = {0}'.format(NLPD))
```

4.2 3.2. Unidimensional regression

Use now only the first company to compute the non-linear regression. Obtain the posterior distribution of $f(\mathbf{x}^*)$ evaluated at the test values $\mathbf{x}^*$, i.e, $p(f(\mathbf{x}^*) \mid \mathbf{s})$.

This distribution is Gaussian, with mean $\mathbb{E}\{f(\mathbf{x}^*) \mid \mathbf{s}\}$ and a covariance matrix $\mathrm{Cov}[f(\mathbf{x}^*) \mid \mathbf{s}]$. Sample 50 random vectors from the distribution and plot them vs. the values x^*, together with the test samples.

The Bayesian model does not provide a single function, but a pdf over functions, from which we extracted 50 possible functions.

[ ]:
```
# <SOL>
X_1d = np.matrix(Xtrain[:,0]).T
Xt_1d = np.matrix(Xtest[:,0]).T
Xt_1d = np.sort(Xt_1d,axis=0) #We sort the vector for representational purposes
```

```
dist = spatial.distance.cdist(X_1d,X_1d,'euclidean')
dist_ss = spatial.distance.cdist(Xt_1d,Xt_1d,'euclidean')
dist_s = spatial.distance.cdist(Xt_1d,X_1d,'euclidean')

K = (sigma_0**2)*np.exp(-dist/l)
K_ss = (sigma_0**2)*np.exp(-dist_ss/l)
K_s = (sigma_0**2)*np.exp(-dist_s/l)

m_y = K_s.dot(np.linalg.inv(K + sigma_eps**2 * np.eye(K.shape[0]))).dot((Ytrain))
v_f = K_ss - K_s.dot(np.linalg.inv(K + sigma_eps**2 * np.eye(K.shape[0]))).
 ↪dot(K_s.T)

L = np.linalg.cholesky(v_f+1e-10*np.eye(v_f.shape[0]))

for iter in range(50):
    f_ast = L.dot(np.random.randn(len(Xt_1d),1)) + m_y
    plt.plot(np.array(Xt_1d)[:,0],f_ast[:,0],'c:');

# Plot as well the test points
plt.plot(np.array(Xtest[:,0]),Ytest[:,0],'r.',markersize=12);
plt.plot(np.array(Xt_1d)[:,0],m_y[:,0],'b-',linewidth=3,label='Predictive mean');

plt.legend(loc='best')
plt.xlabel('x',fontsize=18);
plt.ylabel('s',fontsize=18);
# </SOL>
```

Plot again the previous figure, this time including in your plot the confidence interval delimited by two standard deviations of the prediction. You can observe how 95.45% of observed data fall within the designated area.

[ ]:
```
# <SOL>
X_1d = np.matrix(Xtrain[:,0]).T
Xt_1d = np.matrix(Xtest[:,0]).T
idx = np.argsort(Xt_1d,axis=0) # We sort the vector for representational purposes
Xt_1d = np.sort(Xt_1d,axis=0)
idx = np.array(idx).flatten().T
Ytest = Ytest[idx]

dist = spatial.distance.cdist(X_1d,X_1d,'euclidean')
dist_ss = spatial.distance.cdist(Xt_1d,Xt_1d,'euclidean')
dist_s = spatial.distance.cdist(Xt_1d,X_1d,'euclidean')

K = (sigma_0**2)*np.exp(-dist/l)
K_ss = (sigma_0**2)*np.exp(-dist_ss/l)
K_s = (sigma_0**2)*np.exp(-dist_s/l)
```

```
m_y = K_s.dot(np.linalg.inv(K + sigma_eps**2 * np.eye(K.shape[0]))).dot((Ytrain))
v_f = K_ss - K_s.dot(np.linalg.inv(K + sigma_eps**2 * np.eye(K.shape[0]))).
 ↪dot(K_s.T)
v_f_diag = np.diagonal(v_f)

L = np.linalg.cholesky(v_f+1e-10*np.eye(v_f.shape[0]))

for iter in range(50):
    f_ast = L.dot(np.random.randn(len(Xt_1d),1)) + m_y
    plt.plot(np.array(Xt_1d)[:,0],f_ast[:,0],'c:');

# Plot as well the test points
plt.plot(np.array(Xtest[:,0]),Ytest[:,0],'r.',markersize=12);
plt.plot(np.array(Xt_1d)[:,0],m_y[:,0],'b-',linewidth=3,label='Predictive mean');
plt.plot(np.array(Xt_1d)[:,0],m_y[:,0]+2*v_f_diag,'m--',label='Predictive mean␣
 ↪of f $\pm$ 2std',linewidth=3);
plt.plot(np.array(Xt_1d)[:,0],m_y[:,0]-2*v_f_diag,'m--',linewidth=3);

#Plot now the posterior mean and posterior mean \pm 2 std for s (i.e., adding␣
 ↪the noise variance)
plt.plot(np.array(Xt_1d)[:,0],m_y[:,0]+2*v_f_diag+2*sigma_eps,'m:
 ↪',label='Predictive mean of s $\pm$ 2std',linewidth=3);
plt.plot(np.array(Xt_1d)[:,0],m_y[:,0]-2*v_f_diag-2*sigma_eps,'m:',linewidth=3);

plt.legend(loc='best')
plt.xlabel('x',fontsize=18);
plt.ylabel('s',fontsize=18);
# </SOL>
```

Compute now the MSE and NLPD of the model. The correct results are given below:

[ ]:
```
# <SOL>
MSE = np.mean((m_y - Ytest)**2)
v_y = np.diagonal(v_f) + sigma_eps**2
NLPD = 0.5 * np.mean(((Ytest - m_y)**2)/(np.matrix(v_y).T) + 0.5*np.log(2*np.
 ↪pi*np.matrix(v_y).T))
# </SOL>

print('MSE = {0}'.format(MSE))
print('NLPD = {0}'.format(NLPD))
```