SML_assignment4_ex1

January 11, 2017

1 Exercise 1 - Gaussian Processes

1.1 1

Not much to say it does what it says on the tin; implements equation (3) of the pdf.

```
In [1]: from pylab import *
        from numpy import *

        close('all')
        #  q1
        def kernel(x1, x2, theta):
            """
            Code for equation 3
            """
            sigma = np.zeros((len(x1), len(x2)))
            for idx, i in enumerate(x1):
                for jdx, j in enumerate(x2):
                    sigma[ idx, jdx ] = \
                        theta[ 0 ] *\
                        np.exp(- .5 * theta[ 1 ] * np.sqrt((i - j)**2)) + \
                        theta[ 2 ] + theta[ 3 ] * i * j
            return sigma
```

1.2 2

Define theta and the linearly spaced points and use the kernel function from 1.

```
In [2]: #q2
        theta = ones(4)
        N = 101
        x = linspace(-1,1, N)
        K = kernel(x, x, theta)
```

1.3 3

The Gramm matrix would be 101x101, since it should reflect the kernel between all the datapoints. In order to show that a matrix is semipositive definite we need to show that all eigenvalues are non-negative, $\lambda_i \geq 0$.

```
In [3]: print('Shape of K', K.shape)
        eigenValues, eigenVectors = linalg.eig(K)
        print(r'are all eigenvalues non-negative?', all(eigenValues >= 0))
```

```
Shape of K (101, 101)
are all eigenvalues non-negative? True
```

As we can see all the eigenvalues are non-negative. Hence, our K is semi-positive definite.

1.4 4

Please see figure 1.

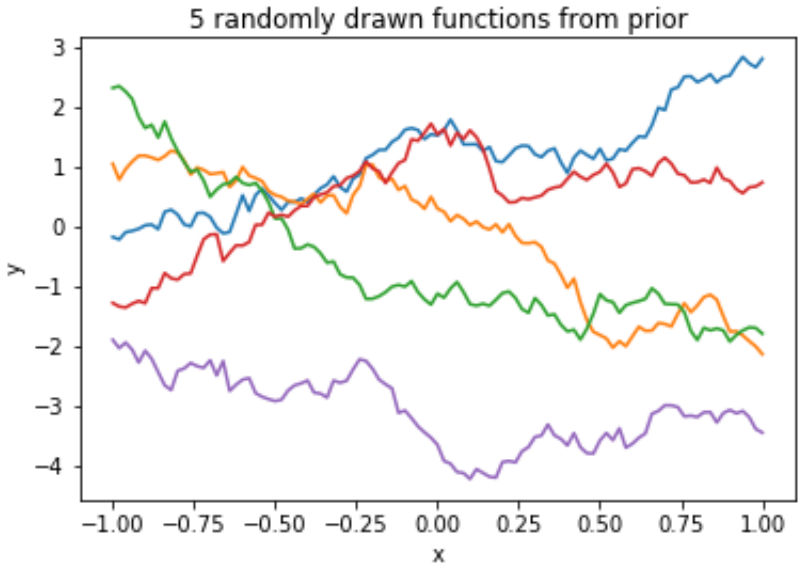

Figure 1: Results for 1.4; 5 randomly drawn functions drawn from the prior over the input space.

```
In [4]: # q4
        # import multivariate normal object
        from scipy.stats import multivariate_normal as mv
        # define mu / sigma
        mu = zeros(len(x))
        # prior object
        prior = mv(mu, K)
        # sample from prior
        samples = prior.rvs(5)
```

```
In [5]: # show samples from the prior
        fig, ax = subplots(1, 1)
        for i in samples:
            ax.plot(x, i)
```

```
savefig('../Figures/ex1q5')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_title('5 randomly drawn functions from prior')
savefig('../Figures/1.4.png')
```

1.5 5

The code is self-explanatory; feed the kernel the training set with different theta values. The results are displayed in **??**.

```
In [6]: # q5
        thetas = np.array([ [ 1, 4, 0, 0 ],\
                            [ 9, 4, 0, 0 ],\
                            [ 1, 64, 0, 0 ],
                           [ 1, 0.25, 0, 0 ],\
                            [ 1, 4, 10, 0 ],\
                            [ 1, 4, 0, 5 ] ])

        # we want at most 3 rows
        nRows = 3
        # get the the number of columns
        nCols = thetas.shape[ 0 ] // nRows

        # plot the prior for different theta
        fig, ax = subplots(nrows=nRows, ncols=nCols, sharex='all')
        for idx, thetai in enumerate(thetas):
            # compute the Gramm matrix
            K = kernel(x, x, thetai)
            # generate prior object
            prior = mv(mu, K, allow_singular=1)
            # sample from the prior
            samples = prior.rvs(5)
            for i in samples:
                ax.flatten()[ idx ].plot(x, i)
            ax.flatten() [ idx ].set_title(r'$\theta$ = {0}'.format(thetai))
            ax.flatten() [ idx ].set_xlabel('x')
            ax.flatten() [ idx ].set_ylabel('y')
        fig.suptitle(r'Samples from prior with different $\theta$ ')
        fig.tight_layout()
        fig.subplots_adjust(top=0.85)
        savefig('../Figures/1.5.png')
```

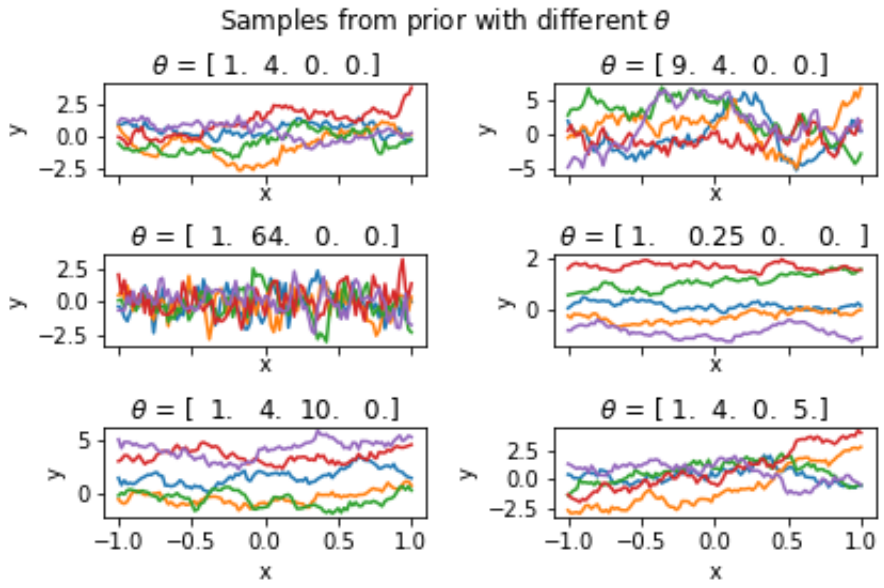

Figure 2: Results for 1.5. Samples drawn from the prior for different values of theta. An explanation for the different settings of θ is given in 1.5.1.

1.5.1 Bonus explanation

From the left column top two plots, and middle right we see that $\theta[1]$ regulates the amount of 'jitter' in the random process. From (3) we see that $\theta[1]$ weighs the euclidean distance between x, x'. Comparing the top row plots, we see that the non-linearity of the random process is controlled by increasing $\theta[0]$, i.e. (global) curvature of the lines. In other words, the gaussian part of the kernel favors clustered points, i.e. if the euclidean distance between them is small, it will tend to pull the process towards these points; creating these jitters we see here. The off-set of the random process is controlled by $\theta[2]$ (trivially) this can be seen in comparing the bottom and top plot in the left column. Finally, $\theta[3]$ controls the amount of linearity of change in the random process. This can be seen in bottom right plot, compared to the top row of plots. So in all we can summarize it as:

1. - Given the kernel function in (3):

    - $\theta[0]$ controls the non-linear component of the random process
    - $\theta[1]$ controls the 'jitterness' of the random process, i.e. weigh the euclidean distance in the random process
    - $\theta[2]$ controls the mean off set in the random process
    - $\theta[3]$ controls the linear component of change of the random process

1.6 6

In computing C we will use equation 6.62 from Bisschop.

```
In [7]: # q6
        # define training set; inputs, targets
        xTrain = array([[-.5, .2, .3 , -.1]]).T
```

```
        tTrain = array([[.5, -1, 3, -2.5]]).T
        KTrain = kernel(xTrain, xTrain, theta)
        beta   = 1
        C      = KTrain + 1/beta * eye(len(xTrain))
        print('C:\n ',C)

        # show the training points
        fig, ax = subplots()
        ax.scatter(xTrain, tTrain)
        ax.set_title('Scatter plot of training points')
        ax.set_ylabel('Targets')
        ax.set_xlabel('Input')
        savefig('../Figures/1.6.png')
```

```
C:
  [[ 3.25        1.60468809  1.52032005  1.86873075]
 [ 1.60468809  3.04        2.01122942  1.84070798]
 [ 1.52032005  2.01122942  3.09        1.78873075]
 [ 1.86873075  1.84070798  1.78873075  3.01      ]]
```

2 7

In order to compute the μ at x = 0, we will to compute the Gramm matrix using the training points and the new point (see Bisschop p.307); i.e. we compute K = K($\boldsymbol{X_n}, \boldsymbol{X_{N+1}}$), where $\boldsymbol{X_n}$ is the set $i = 1, ..., N$ and $X_{N+1} = 0$.

```
In [8]: xNew       = array([[0]])
        xTrainNew  = vstack((xTrain, xNew))
        c          = kernel(xNew, xNew, theta) + 1/beta
        k          = kernel(xTrain, xNew, theta)
        invC       = linalg.inv(C)
        # Bisschop 6.66
        mu_new     = k.T.dot(invC).dot(tTrain)
        # Bisschop 6.67
        sigma_new  = c - k.T.dot(invC).dot(k)
        print('mu at x = {0}:\n{1}'.format(xNew[0], mu_new) )
        print('sigma^2 at x = {0}:\n{1}'.format(xNew[0], sigma_new) )
```

```
mu at x = [0]:
[[-0.20721389]]
sigma^2 at x = [0]:
[[ 1.32520089]]
```

3 8

The mean of $p(t \mid (t)))$ will not go to zero as $x \to \pm\infty$. If x approaches $\pm\infty$ the gaussian part of (3) will go to zero but the linear part ($x^T x'$) will blow up to $\pm\infty$ (depending on the sign of x'. Thus to make the kernel function to go to zero we would need to suppress $x^T x'$ by setting $\theta[3] = \theta[2] = 0$ (the bias, $\theta[2]$ will trivially cause non-zero if set to anything else than zero).

As an example, we take a large value for x and print the mean by setting $\theta[2] = \theta[3] = 0$.

```
In [9]: # take a very large number
        xNew       = array([[1e30]])
        xTrainNew  = vstack((xTrain, xNew))
        c          = kernel(xNew, xNew, theta) + 1/beta
        thetaEdit  = theta.copy()
        thetaEdit[-2:] = 0
        k          = kernel(xTrain, xNew, thetaEdit)
        invC       = linalg.inv(C)
        # Bisschop 6.66
        mu_new     = k.T.dot(invC).dot(tTrain)
        # Bisschop 6.67
        sigma_new  = c - k.T.dot(invC).dot(k)
        print('mu at x  = {0}:\n{1}\nand theta:\n {2}'\
              .format(xNew[0], mu_new, thetaEdit) )

mu at x  = [  1.00000000e+30]:
[[ 0.]]
and theta:
 [ 1.  1.  0.  0.]
```