Lab 3: Gaussian process regression

Machine Learning 1, September 2016

- The lab exercises should be made in groups of two people.
- The deadline is October 30th (Sunday) 23:59 after the final exam.
- Assignment should be sent to your teaching assistant. The subject line of your email should be "lab#_lastname1_lastname2_lastname3".
- Put your and your teammates' names in the body of the email.
- Attach the .IPYNB (IPython Notebook) file containing your code and answers. Naming of the file follows the same rule as the subject line. For example, if the subject line is "lab01_Kingma_Hu", the attached file should be "lab01_Kingma_Hu.ipynb". Only use underscores ("_") to connect names, otherwise the files cannot be parsed.

Notes on implementation:

- You should write your code and answers in an IPython Notebook: http://ipython.org/notebook.html (http://ipython.org/notebook.html). If you have problems, please contact us.
- Among the first lines of your notebook should be "%pylab inline". This imports all required modules, and your plots will appear inline.
- NOTE: Make sure we can run your notebook / scripts!

Gaussian process regression

For this Lab we will be refer to Bishop sections 6.4.2 and 6.4.3. You may also want to refer to Rasmussen's Gaussian Process text which is available online at http://www.gaussianprocess.org/gpml/chapters/ (http://www.gaussianprocess.org/gpml/chapters/) and especially to the project found at http://www.automaticstatistician.com/index.php (http://www.automaticstatistician.com/index.php) by Ghahramani for some intuition in GP. To understand Gaussian processes, it is highly recommended understand how marginal, partitioned Gaussian distributions can be converted into conditional Gaussian distributions. This is covered in Bishop 2.3 and summarized in Eqns 2.94-2.98.

Sinusoidal Data

We will use the same data generating function that we used previously for regression. You can change sigma/beta, but keep it reasonable. Definitely play around once you have things working. Make use of these functions as you wish.

```
In [1]: %pylab inline
        import pylab as pp
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
In [2]: sigma = 0.5
        beta  = 1.0 / pow(sigma,2) # this is the beta used in Bishop Eqn. 6.59
        N_test = 100
        x_test = np.linspace(-1,1,N_test);
        mu_test = np.zeros( N_test )
```

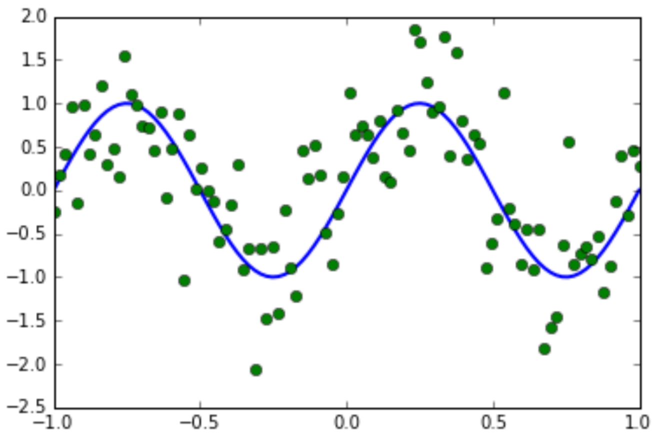
In [3]:
```
def true_mean_function( x ):
    return np.sin( 2*pi*(x+1) )

def add_noise( y, sigma ):
    return y + sigma*np.random.randn(len(y))

def generate_t( x, sigma ):
    return add_noise( true_mean_function( x), sigma )
```

In [4]:
```
y_test = true_mean_function( x_test )
t_test = add_noise( y_test, sigma )
pp.plot( x_test, y_test, 'b-', lw=2)
pp.plot( x_test, t_test, 'go')
```

Out[4]:
```
[<matplotlib.lines.Line2D at 0x10b0d3fd0>]
```



1. Sampling from the Gaussian process prior (30 points)

We will implement Gaussian process regression using the kernel function in Bishop Eqn. 6.63.

1.1 k_n_m(xn, xm, thetas) (10 points)

To start, implement function "k_n_m(xn, xm, thetas)" that takes scalars $\mathbf{x}_n$ and $\mathbf{x}_m$, and a vector of 4 thetas, and computes the kernel function Bishop Eqn. 6.63 (10 points).

In [ ]:

1.2 computeK(X1, X2, thetas) (5 points)

Eqn 6.60 is the marginal distribution of mean ouput of N data vectors: $p(\mathbf{y}) = \mathcal{N}(\mathbf{0}, \mathbf{K})$. Notice that the expected mean function is 0 at all locations, and that the covariance is a N by N kernel matrix $\mathbf{K}$. Write a function "computeK(X1, X2, thetas)" that computes the kernel matrix. Hint: use k_n_m as part of an innner loop (of course, there are more efficient ways of computing the kernel function making better use of vectorization, but that is not necessary) (5 points).

In [ ]:

1.3 Plot function samples (15 points)

Now sample mean functions at the x*test locations for the theta values in Bishop Figure 6.5, make a figure with a 2 by 3 subplot and make sure the title reflects the theta values (make sure everything is legible). In other words, sample $\yi \sim \mathcal{N}(\zero,* \K{\thetav}). $Makeuseofnumpy.random.multivariate_normal().Onyourplotsincludetheexpectedvalueof\$ is the variance of the model uncertainty) (15 points).

```
In [ ]:
```

2. Predictive distribution (35 points)

So far we have sampled mean functions from the prior. We can draw actual data $\mathbf{t}$ two ways. The first way is generatively, by first sampling $\mathbf{y}|\mathbf{K}$, then sampling $\mathbf{t}|\mathbf{y},\beta$ (Eqns 6.60 followed by 6.59). The second way is to integrate over $\mathbf{y}$ (the mean draw) and directly sample $\mathbf{t}|\mathbf{K},\beta$ using Eqn 6.61. This is the generative process for $\mathbf{t}$. Note that we have not specified a distribution over inputs $\mathbf{x}$; this is because Gaussian processes are conditional models. Because of this we are free to generate locations $\mathbf{x}$ when playing around with the GP; obviously a dataset will give us input-output pairs.

Once we have data, we are interested in the predictive distribution (note: the prior is the predictive distribution when there is no data). Consider the joint distribution for $N+1$ targets, given by Eqn 6.64. Its covariance matrix is composed of block components $\mathbf{C}_N$, $\mathbf{k}$, and c. The covariance matrix $\mathbf{C}_N$ for $\mathbf{t}_N$ is $\mathbf{C}_N = \mathbf{K}_N + \mathbf{I}_N/\beta$. We have just made explicit the size N of the matrix; N is the number of training points. The kernel vector $\mathbf{k}$ is a N by 1 vector of kernel function evaluations between the training input data and the test input vector. The scalar c is a kernel evaluation at the test input.

2.1 gp_predictive_distribution(...) (10 points)

Write a function "gp_predictive_distribution(x_train, t_train, x_test, theta, beta, C = None)" that computes Eqns 6.66 and 6.67, except allow for an arbitrary number of test points (not just one) and now the kernel matrix is for training data. By having C as an optional parameter, we can avoid computing it more than once (for this problem it is unimportant, but for real problems this is an issue). The function should compute $\mathbf{C}$, $\mathbf{k}$ and c, and return the mean and noise functions. Do not forget: the computeK function computes $\mathbf{K}$, not $\mathbf{C}$! (10 points)

```
In [ ]:
```

2.2 gp_log_likelihood(...) (10 points)

Later, to learn the hyperparameters, we will need to compute the log-likelihood of the of the training data. Implicitly, this is conditioned on the value setting for θ. Write a function "gp_log_likelihood(x_train, t_train, theta, C = None, invC = None, beta = None)", where C and invC can be stored and reused. (10 points) Note: you need to add beta if you want to calculate C.

```
In [ ]:
```

2.3 Plotting (10 points)

Repeat the 6 plots above, but this time conditioned on the training points. Use the sinuosoidal data generator to create 2 training points where x is sampled uniformly between -1 and 1. For these plots, feel free to use the provided function "gp_plot". Make sure you put the parameters in the title and this time also the log-likelihood. (10 points) Try to understand the two types of uncertainty! If you do not use "gp_plot", please add a fill between for the model and target noise.

In [5]:
```
def gp_plot( x_test, y_test, mu_test, var_test, x_train, t_train, theta, beta
):
    # x_test:   the test data
    # y_test:   the true function at x_test
    # mu_test:  predictive mean at x_test
    # var_test: predictive covariance at x_test
    # t_train:  the training values
    # theta:    the kernel parameters
    # beta:     the precision (known)

    # the reason for the manipulation is to allow plots separating model and d
ata stddevs.
    std_total = np.sqrt(np.diag(var_test))         # includes all uncertainty,
 model and target noise
    std_model = np.sqrt( std_total**2 - 1.0/beta ) # remove data noise to get
model uncertainty in stddev
    std_combo = std_model + np.sqrt( 1.0/beta )    # add stddev (note: not the
 same as full)

    pp.plot( x_test, y_test, 'b', lw=3)
    pp.plot( x_test, mu_test, 'k--', lw=2 )
    pp.fill_between( x_test, mu_test+2*std_combo,mu_test-2*std_combo, color='k
', alpha=0.25 )
    pp.fill_between( x_test, mu_test+2*std_model,mu_test-2*std_model, color='r
', alpha=0.25 )
    pp.plot( x_train, t_train, 'ro', ms=10 )
```

In [ ]:

2.4 More ploting (5 points)

Repeat the 6 plots above, but this time conditioned a new set of 10 training points. (5 points)

In [ ]:

3. Learning the hyperparameters (45 points)

Learning the values of the parameter θ can be very tricky for Gaussian processes in general, but when the data is univariate like ours, we can visualize the fit and see how plausible it looks.

3.1 Derivatives (5 points)

Maximum likelihood or MAP learning is the most common way of setting the parameters, though a fully Bayesian approach is possible too. We will look at ML today. For this, we start with the dervivative of the log-likelihood with respect to the parameters θ; this is Eqn 6.70. This, in turn, requires the derivative of the kernel matrix $\mathbf{C}_N$ wrt θ. This is the matrix of element-wise derivatives of the kernel function. Write the derivatives for θ_0 to θ_3 for our kernel function (5 points).

[***answer here***]

3.2 Questions (5 points)

Which parameters in θ are constrained, that is, where not all positive/ negative values are valid? (5 points)

[***answer here***]

3.3 More derivatives (5 points)

For parameters that are constrained to be positive, the usual approach is to use the exponential of the free-parameter in the kernel function, but perform gradient ascent on the unconstrained values. Consider the case $\theta_i = \exp(\phi_i)$, where ϕ_i is unconstrained. Write the derivative for ϕ_i in terms of the derivatives you already computed (5 points). Hint: use the chain rule and do not repeat the full derivation.

[***answer here***]

3.4 Grid search (10 points)

Grid-search: for the same training set you have above, perform a small grid search over θ (try at least 20 combinations). Have your grid-search loop or function print out rows of log-likelihood + θ sorted by best to worst. Use the log-likelihood to select the best θ and the worst. Plots both the same way as the subplots above (ie a 1 by 2 subplot of best and worst). (10 points)

[***answer here***]

3.5 Questions (10 points)

Selecting kernel functions can be somewhat of an art. There are charateristics of kernel functions that are useful for some data sets, but not others. Complicating the matter is the ability to combine kernels with different characteristics (long term trends + seasonal fluctuations). Describe the charactistics of the kernel function we are using in terms of (signal, scale, offsets, etc). You may want to play around with θ and see what each parameter does/affects/etc. (5 points) Describe why the best parameters work well for the training data and explain why the bad parameter settings perform poorly (in terms of the first part of the question). (5 points)

[***answer here***]

3.6 Bonus: Implementation (20 points)

Implement gradient-ascent (or descent if you wish) using the combination of a) the log-likelihood objective function and b) the gradients you calculated above. Run on the training data above and show the log-likehood curve as it learns and a plot of the final model. Feel free to use available software (eg search for "minimize.py" which uses conjugate gradient descent, or something in scipy). NB: log-likelihood should be monotonically increasing. You are encouraged to also search and use "checkgrad". (20 points)

```
In [ ]:
```