Advanced Statistical Inference
Gaussian Process Regression

1 Aims

- To sample from a Gaussian process prior distribution.
- To implement Gaussian process inference for regression.
- To use the above to observe samples from a Gaussian process posterior distribution.
- To evaluate how different hyperparameter settings impact model quality.

2 Introduction

Gaussian processes (henceforth GPs) achieve greater flexibility over parametric models by imposing a preference bias as opposed to restrictive constraints. Although the parameterisation of GPs allows one to access a certain (infinite) set of functions, preference can be expressed using a *prior* over functions. This allows greater freedom in representing data dependencies, thus enabling the construction of better-suited models. In this lab, we shall cover the basic concepts of GP regression. For the sake of clarity, we shall focus on univariate data, which enables better visualisation of the GP. Nonetheless, the code implemented within this lab can very easily be extended to handle multi-dimensional inputs.

3 Sampling from GP Prior

We shall consider a one-dimensional regression problem, whereby the inputs `x` are transformed by a function

```
f(x) = sin(exp(0.03 * x))
```

Generate 200 random points, `x`, in the range [-20, 80], and compute their corresponding function values, `t`. The target function can then be plotted accordingly.

Recall that since GPs are non-parametric, we define a *prior* distribution over functions (models), specified as a multivariate Gaussian distribution

$$p(f) = \mathcal{N}(\mu, \Sigma).$$

Without loss of generality, we shall assume a zero-mean GP prior, i.e. $\mu = \mathbf{0}$. The covariance matrix of the distribution, Σ, may then be computed by evaluating the covariance between the input points. For this tutorial, we shall consider the widely used squared-exponential (RBF) covariance (also referred to as the kernel function), which is defined between two points as:

$$k(x, x') = \sigma_f^2 \mathbf{exp}\left(-\frac{(x - x')^2}{2l^2}\right).$$

This kernel is parameterised by a lengthscale parameter l, and variance σ_f^2. Given that the true function may be assumed to be corrupted with noise, we can also add a noise parameter, σ_n^2, to the diagonal entries of the resulting kernel matrix, K, such that

$$K = K + \sigma_n^2 I.$$

Download the file *compute_kernel.m* and complete the implementation for computing RBF covariance between two sets of input points. *Hint: The 'pdist2' function in Matlab can be used for evaluating the pairwise Euclidean distance between two sets of points.*

Assuming a zero-mean prior, and using the kernel matrix constructed with *compute_kernel* for input points `x`, we can sample from the prior distribution using the `mvnrnd` function as follows:

```
mu = zeros(length(x), 1)
K = compute_kernel(....)
r = mvnrnd(mu, K)
```

On the same figure where we plotted the true function, draw four samples from the prior distribution and plot them accordingly. In order to better understand the role of the hyperparameters, observe how altering them impacts the shape of the prior samples.

4 GP Inference

Recall that the prior represents our prior beliefs before observing the function values of any data points. Suppose we can now observe 3 points at random from the input data; we would expect that with this additional knowledge, the functions drawn from the updated GP distribution would be constrained to pass through these points (or at least close if corrupted with noise). The combination of the prior and the data leads to the posterior distribution over functions.

Assign 3 points at random from `x` (and their corresponding function values) to `obs_x` and `obs_t` respectively. We shall also package the kernel parameters as a list, `params = [lengthscale, variance, noise]`. You are encouraged to use the following initial configuration:

```
lengthscale = 10
variance = 2
noise = 1e-6
```

Download the file *gp_inference.m* and complete the provided implementation for evaluating the posterior GP mean and variance using the equations given in the lecture.

Note: As we have encountered in previous labs, matrix inversions can be both numerically troublesome and slow to compute. In this lab, we shall avoid computing direct matrix inversions by instead considering Cholesky decompositions for solving linear systems. The steps for using Cholesky within GP inference are outlined as comments within the provided code. Even so, you are encouraged to read more about Cholesky decompositions for GPs by consulting Appendix A.4 of Gaussian Processes for Machine Learning (Rasmussen and Williams, 2005) - available online!

5 Sampling from GP Posterior

Now that you have computed the posterior mean and variance, create a new figure once again showing the true function. To this figure, add the posterior mean and 95% confidence interval; this can be achieved using the following code:

```
[post_m, post_V, nll] = gp_inference( ... );
post_stdv = sqrt(diag(post_V));
figure
hold on
color = [0.6 0.5 0.4];
fill([x;x(end:-1:1)], [post_m;post_m(end:-1:1)] +
  1.96*[post_stdv; -post_stdv(end:-1:1)], color);
plot(x,post_m,'g-')
plot(x,t,'r-')
```

Note that we should also add the noise variance to the predictive variance of the posterior. Fix this accordingly.

Additionally, as we did with the prior distribution, sample four functions from the posterior and plot them on the same figure. Comment on what you observe. You should also view how the posterior mean and variance improve when more observations are included in the GP inference procedure.

As a measure of model quality, you should also compute the log marginal likelihood of the model. To this end, complete the code provided in *gp_inference* to include the `nll` term. *Hint: Refer to Algorithm 2.1 in Chapter 2 of the book referenced above for details on how to compute this term efficiently.*

Bonus: Parameter Optimisation

Optimise the hyperparameters of the model by minimising the negative log-likelihood of the model. For a complete solution, you should include the derivatives of the objective function with respect to the parameters being optimised.

Note that the parameters l, σ_f^2, and σ_n^2 are supposed to be positive. It is possible that the optimisation algorithm attempts to evaluate the log-likelihood in regions of the parameter space where one or more of these parameters are negative, leading to numerical issues. An easy way around this is to optimise a transformed version of covariance parameters using the logarithm transformation. In particular, define $\psi_l = \log(l)$, $\psi_f = \log(\sigma_f^2)$, and $\psi_n = \log(\sigma_n^2)$, and optimise with respect to the ψ parameters. The optimisation problem in the transformed space is now unbounded, and the gradient of the log-likelihood should be computed with respect to the ψ parameters.