Hyperparameter Searching

We can judge how good a set of hyperparameters is by the probability they give to the training data, perhaps combined with a prior density for the hyperparameters. But how can we search for the best hyperparameter values, or average over them?

- If there are only a few hyperparameters, we might define a grid of values for each (eg, $\rho = 0.01, 0.03, 0.1, 0.3, 1, 3, 10$), and then try all possible combinations of values on these grids. This is infeasible for more than about three hyperparameters
- We can use various numerical optimization methods to find hyperparameters that at least locally maximize the probability of the data. It is possible to compute derivatives of the data probability with respect to the hyperparameters, which often helps with optimization.
- We can use Markov chain Monte Carlo methods. We define a Markov chain that moves around the space of hyperparameters, spending an amount of time in each region that is proportional to the probability of the data given the hyperparameters, times the prior for the hyperparameters. We then average predictions done using these possible hyperparameter values.

Interesting Things to Do With Hyperparameters

Hyperparameters are good for more than just controlling overfitting:

- Hyperparameters multiplying linear and exponential covariance terms can control how close the function is to being linear. Eg, in

$$C(x_{i_1}, x_{i_2}) \;=\; \sigma_0^2 \;+\; \sigma_1 x_{i_1} x_{i_2} \;+\; \eta^2 \exp(-(x_{i_1} - x_{i_2})^2) \;+\; \delta_{i_1,i_2} \sigma_\epsilon^2$$

  the values of σ_1 and η control how linear/non-linear the function is.

- Hyperparameters multiplying sums and products of exponential covariances can control whether or not an additive model is used.

- Using a different scale hyperparameter for each input variable lets the model discover how relevant each input is to predicting the response. Eg,

$$C(x_{i_1}, x_{i_2}) \;=\; \eta^2 \exp\left(-\sum_{j=1}^{p} ((x_{i_1,j} - x_{i_2,j}) \,/\, \rho_j)^2\right)$$

One can do similar things with spline models fit using penalties, setting multiple penalty factors by cross-validation.

Spline and Gaussian Process Methods for Classification

Both spline and Gaussian process models apply naturally to regression problems — where the response is real-valued. The response, Y, in a case with inputs X is modeled as

$$Y \;=\; f(X) \,+\, \epsilon$$

where f is the function modeled by the spline or the Gaussian process, and ϵ is the noise for this case, assumed to be Gaussian.

For binary classification problems, we can use a logistic regression approach, in which

$$P(G=1\,|\,X) \;=\; [1 \,+\, \exp(-f(X))]^{-1}$$

For classification with K classes, we use K functions, $f_k(x)$, and model class probabilities as

$$P(G=k\,|\,X) \;=\; \frac{\exp(f_k(X))}{\sum_{\ell=1}^{K} \exp(f_\ell(X))}$$

Spline Classification Using Maximum Penalized Likelihood

To fit spline models for classification, we minimize minus the log likelihood, rather than minimize the RSS. For smoothing splines, we add a penalty relating to the derivatives of f to minus the log likelihood.

Using a natural cubic smoothing spline for binary classification, with classes $Y = 0$ and $Y = 1$, we find the function f that minimizes

$$\sum_{i=1}^{N} \Big[y_i \log(1 + \exp(-f(x_i))) \;+\; (1-y_i) \log(1 + \exp(f(x_i))) \Big] \;+\; \lambda \int \Big[f''(t) \Big]^2 dt$$

If we have $K > 2$ classes, we need K functions, $f_k(x)$, which are all involved in the minus log likelihood term. To this we add K penalty terms, one for each f_k.

As before, the f that minimizes this is a natural cubic spline. So we can find f by minimizing over the coefficients of a set of basis functions for natural cubic splines.

Unlike minimization of a penalized RSS, there is no formula for the coefficients that minimize the penalized minus log likelihood. We have to use iterative numerical methods.

Gaussian Process Classification (The Easy Way)

Gaussian process classification uses "latent variables" associated with each case.

For binary classification, with $y_i = 0$ or $y_i = 1$, we introduce a latent variable z_i for each case. The Gaussian process defines a multivariate Gaussian distribution for $z_1, \ldots, z_N$, as determined by the covariance function. The class variables are then modeled by $P(y_i = 1 \,|\, z_i) \;=\; [1 + \exp(-z_i)]^{-1}$.

We could now find values for $z_1, \ldots, z_N$ by minimizing minus the log likelihood plus minus the log of the Gaussian prior density, which (omitting constants) is:

$$\sum_{i=1}^{N} \Big[y_i \log(1 + \exp(-z_i)) + (1 - y_i) \log(1 + \exp(z_i)) \Big] \;+\; \log(\det(\mathbf{C}))/2 \;+\; \mathbf{z}^T \mathbf{C}^{-1} \mathbf{z}/2$$

As for splines, this minimization has to be done by iterative numerical methods.

We predict the class in a test case by predicting the latent value for the test case (with the same prediction formulas as for regression), then converting this latent variable to a class probability.

This isn't too hard, but it's not exactly what the Bayesian framework says to do — we ought to *average* over possible values for the latent variables.

Gaussian Process Classification (The Hard Way)

Doing Gaussian process classification completely correctly is harder.

We'd like to predict for a test case as follows:

$$P(y_* = 1 \mid y_1, \ldots, y_N)$$

$$= \int P(y_* = 1 \mid z_*)\, P(z_* \mid z_1, \ldots, z_N)\, P(z_1, \ldots, z_N \mid y_1, \ldots, y_N)\, d(z_1, \ldots, z_N, z_*)$$

It's not possible to do this integral over values of latent variables analytically. We have to use Monte Carlo methods based on Markov chain sampling, which can be combined with sampling to average over values of hyperparameters in the covariance function.