slayR

Table of Contents

Summary

The goal of this project is to collect together a number of R convenience functions that I have written to facilitate statistical analysis and machine learning. My current focus is global optimization for hyperparameter tuning and model selection. Ancillary topics include construction of novel kernel functions for data analysis and hyperparameter selection. I think the functions are self-documenting, but this document collects notes about things which may be unclear or counterintuitive.

Overview

Bayesian Optimization Notes

Posterior predictions computation

The expected improvement metric is computed using posterior predictions from the Gaussian process models. This happens in the generated quantities block of the Stan program using Stan's convenience functions to call lower-level linear algebra functions in a C++ library (more details in the Stan documentation).

These posterior predictions for the Gaussian process have the following forms:

$$\hat{y} = \mu + K(\tilde{x}, x)[K(x, x) + \sigma^2 I]^{-1}(y_{\text{old}} - \mu)$$
$$cov(\hat{y}) = K(\tilde{x}, \tilde{x}) - K(\tilde{x}, x)^T[K(x, x) + \sigma^2 I]^{-1}K(x, \tilde{x})$$

Where $K(a, b)$ is the kernel function computed for the vector-valued inputs a and b, and μ is the *constant* mean function. Suppose that there are m points a and n points b. We will only consider Mercer kernels, so the kernel functions are symmetric. Furthermore, Mercer kernels are positive semi-definite (PSD) which is defined as $zKz^T \geq 0 \forall z \in \mathbb{R}$, and a theorem from linear algebra provides that all eigenvalues of a symmetric, PSD matrix are nonnegative. Thus these equations show that the posterior covariance is always smaller than the prior covariance.

While these equations are all well and good for the linear algebra exposition of the Gaussian process, they aren't great from a scientific computing perspective. In particular, we want to avoid inverting a matrix whenever possible. So we can streamline this notation some using the following conventions:

$$K(x, x) + \sigma^2 I = K$$
$$K(x, \tilde{x}) = B$$

This produces somewhat nicer equations:

$$\hat{y} = \mu + B^T K^{-1}(y_{\text{old}} - \mu)$$
$$\mathbb{V}(\hat{y}) = K(\tilde{x}, \tilde{x}) - B^T K^{-1} B$$

Now, in stan, it's actually preferred to work with the Cholesky decomposition $K = LL^T$ since the multivariate normal distribution will compute the Cholesky by default and it's cheaper to do the computation once and store it than it is to re-compute it every time. Therefore, we can write

$$\hat{y} = \mu + B^T (LL^T)^{-1}(y_{\text{old}} - \mu)$$
$$\mathbb{V}(\hat{y}) = K(\tilde{x}, \tilde{x}) - B^T (LL^T)^{-1} B$$

This produces some interesting factorization opportunities in the covariance equation. Specifically, we can distribute the inversion and examine the resulting product:

$$\mathbb{V}(\hat{y}) = K(\tilde{x}, \tilde{x}) - B^T L^{-T} L^{-1} B$$

This exposes a helpful symmetry: if we denote $L^{-1}B = X$, then $B^T L^{-T} L^{-1} B = X^T X$. So writing out the whole expression for the covariance,

$$\mathbb{V}(\hat{y}) = K(\tilde{x}, \tilde{x}) - X^T X$$

This would be sufficient for a mathematical exposition, but since we want to implement this in code, we have to be careful about computing the matrix inverse. As previously noted, computing the inverse directly is one of the worst ways to compute $\mathbb{V}(\hat{y})$. But we can exploit the triangular shape of L, and observe that we are interested in solving this linear system for X: $LX = B$. So we are computing

$$X = L^{-1}B$$

where L and B are known. Since L is triangular, this can be done cheaply using row operations. Once X is computed, then the whole expression $B^T L^{-T} L^{-1} B$ can be computed by exploiting the symmetry in terms of X: $K = X^T X$. Computing $B^T K^{-1} = A$ is a bit trickier, since it does not have the symmetry property and appears to require an explicit inversion. However, a second look at the underlying mechanics allows us to untangle the problem and exploit the shape matrix properties here as well.

$$B^T K^{-1} = A$$
$$B^T L^{-T} L^{-1} =$$
$$(L^{-1}B)^T L^{-1} =$$

$$X^T L^{-1} = A$$

This is the same form of linear system that we solved before, except that some of the elements are transposed. We know X from the previous step, and L is provided directly by the estimation procedure. So an appropriate call to a triangular solver will provide the output A^T. Our final results then are

$$\hat{y} = \mu + A(y_{\text{old}} - \mu)$$
$$\mathbb{V}(\hat{y}) = K(\tilde{x}, \tilde{x}) - X^T X$$

The Matérn Kernel

In general, the Matérn kernel uses the modified Bessel function. But for some parameter choices, it has a convenient form which will directly control the roughness of the resulting function. We will only consider these special cases here: for the specific choice of $\nu = p + \frac{1}{2} \forall p \in \{\mathbb{N} \cup \{0\}\}$, the isotropic Matérn kernel has the form of the product of an exponential function and a polynomial:

$$k_\nu(r) = \underbrace{\exp\left(\frac{-r\sqrt{2\nu}}{\ell}\right)}_{\text{varies in } r} \cdot \underbrace{\frac{\Gamma(p+1)}{\Gamma(2p+1)}}_{q} \cdot \underbrace{\sum_{k=0}^{p} \frac{(p+k)!}{k!\,(p-k)!}\left(\frac{r\sqrt{8\nu}}{\ell}\right)^{p-k}}_{\text{varies in } r \text{ and } i}$$

Where $r = \|x - x'\|_2 = \sqrt{\sum_{i=1}^{d}(x_i - x_i')^2}$, where it's understood that x is a *vector*. Importantly, the degree of the polynomial is fixed by the choice of p.

But it's probably more realistic to consider an anisotropic kernel in practice. So we can write

$$k_\nu(x, x') = \exp\left(-\underbrace{\sqrt{2\nu(x-x')^T\Lambda(x-x')}}_{d}\right) \cdot q \cdot \sum_{k=0}^{p} \frac{(p+k)!}{k!\,(p-k)!}\left(2\underbrace{\sqrt{2\nu(x-x')^T\Lambda(x-x')}}_{d}\right)^{p-k}$$
$$= \exp(\ln q - d +) \cdot \sum_{k=0}^{p} \exp[\ln\Gamma(p+k+1) - \ln\Gamma(k+1) - \ln\Gamma(p-k+1)] \cdot (2d)^{p-k}$$

Converting the contents of the summation to exponents of sums of logs is necessary because the Stan software only implements $\Gamma(n)$ through the logarithm, i.e. only $\ln\Gamma(n)$ is available, much to my annoyance. Denote $\Lambda = \begin{bmatrix} \rho_1^2 & 0 & \cdots & 0 \\ 0 & \rho_2^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & \rho_n^2 \end{bmatrix}$ a diagonal matrix and ρ_i is the scaling in the i^{th} direction. Note that $d^2 = 2\nu(x-x')^T\Lambda(x-x')$, which is a scalar times a quadratic form, and quadratic forms have convenience functions in Stan. Rearrangement produces the equivalent expression

$$d^2 = (x-x')^T(2\nu\Lambda)(x-x')$$

Which is what is presently implemented.* (But the implementation in Stan *actually* looks like $(x-x')\Lambda(x-x')^T$ because Stan casts $x - x'$ as a *row* vector because it's extracted as a row from a

matrix. The actual results and arithmetic and convenience function usage is the same, but the notation is slightly different.)

The default implementation of the Matérn kernel is $\nu = \frac{5}{2}$, but users may specify an alternative through the argument setting p.

Are there any non-obvious constraints on LCB_delta? (Can it blows some bounds?)

The lower confidence bound acquisition function has a cooling schedule which provably places an upper bound on the regret of the optimization procedures. For LCB, the plan is to choose the point $\tilde{x}$ that minimizes

$$\mu(\tilde{x}) - \sqrt{\beta}\,\sigma(\tilde{x})$$

That is, for some interval of width $\sqrt{\beta}$ about the posterior predictive mean $\mu(\tilde{x})$, what value of $\tilde{x}$ has the smallest left endpoint?

It can be shown that for $\delta \in (0,1)$ and $\beta_t = 2\log\left(\frac{Dt^2\pi^2}{6\delta}\right)$, applying the LCB algorithm with β_t for a sample f of a GP with mean function zero and a covariance function $K(x,x')$, there is a regret bound of $\mathcal{O}^*\left(\sqrt{T\gamma_T \log(D)}\right)$ with high probability.[1] ($D \geq 1$ is the dimensionality of the problem and $t \geq 1$ is the iteration number.) Specifically,

$$\mathbb{P}\left(R_T \leq \sqrt{\frac{8}{\log(1+\sigma^{-2})} T\beta_T\gamma_T}\ \forall T \geq 1\right) \geq 1-\delta.$$

Claim: for all δ in the unit interval, β_t is always positive.

Proof: Observe that we can rearrange β_t to

$$\beta_t = 2\log(Dt^2\pi^2) - 2\log(6\delta)$$

The first term is *smallest* for $t = 1$ and $D = 1$. The second term is *largest* for $\delta = \lim_{\epsilon\to 0} 1-\epsilon$. Thus, the first term reduces to $\log(\pi^4)$ and the left term reduces to $\log(6^2)$. Critically, in the extreme case when the second term is as large as it can ever be and the first term is as small as it can ever be, it is clear that $\beta_t > 0$ because $\pi^4 > 6^2$ which is apparent by inspection. In all other cases, the first term will be larger or the second term will be smaller, so the sign of β_t remains positive. Proof complete.

[1] Niranjan Srinivas, Andreas Krause, Sham M. Kakade and Matthias Seeger, "Gaussian Process Optimization in the Bandit Setting: No Regret and Experimental Design"2010.

Cross-Validation function notes

SVM module

Kernel Factorization

Consider a Mercer kernel: a symmetric function whose $n \times n$ Gram matrix K is positive semidefinite for all inputs $X_1, X_2, \ldots, X_n$ which are elements of a specified set $\mathcal{X}$. K has eigendecomposition

$$K = PDP^{-1}$$

Where D is a diagonal matrix of eigenvalues to K and P is a matrix of corresponding eigenvalues. Because a K is PSD, we know that $D_{ii} > 0 \forall i = \{1,2,3, \ldots, n\}$. Moreover, K is symmetric, which implies that the spectral theorem applies and its eigenvectors form a basis for $\mathbb{R}^n$, so we can write

$$\begin{aligned} K &= PDP^T \\ &= PD^{\frac{1}{2}}D^{\frac{1}{2}}P^T \\ &= Q^TQ \end{aligned}$$

For $Q = D^{\frac{1}{2}}P^T$. The key detail here is that K is the Gram matrix for the inner product kernel function on Q, implying that we can implement an arbitrary kernel function in software through this factorization. To my knowledge, every SVM implementation has a linear kernel option, so this view into the problem will give us a nice platform to do more interesting stuff.

needsFactorization

This is a hard-coded part of the return for all kernels implemented in the SVM module. Some kernels (diffusion graph kernel, for example) will compute an eigendecomposition as a component of the kernel computation. In these cases, it's easier and more precise to simply compute the decomposition once, rather than decompose and multiply only to decompose again immediately at the next step. Most kernels, like the RBF, will just compute the matrix elementwise, so needsFactorization is TRUE for nearly all kernels. But for the ones that inherently involve a spectral decomposition,