Linear regression to Gaussian Process regression to I-SSGPR

Kazuya Otani

May 4, 2018

Derivation and notes for "Real-time model learning using Incremental Sparse Spectrum Gaussian Process Regression", Arjan Gijsberts and Giorgio Metta, 2012.
Mostly just following the explanations in the paper, with additional calculation steps and commentary in between equations.

To-do

- automatic relevance detection
- Read rahimi and recht paper
- Read Locally Weighted Projection Regression paper
- Understand difference between this and other local GP. How do they compare in terms of complexity in updates, inference? What approximations do they make, and how do their expected errors compare?
- Finish some derivations

1 Linear regression

Given examples (x, y)
Compute a linear model $\hat{Y} = Xw + b$ (reduced to $\hat{Y} = Xw$ by augmenting x with constant term at end)...
That minimizes $||Y - \hat{Y}||_2^2 = ||Y - Xw||_2^2$. This amounts to Maximum Likelihood Estimation, where we have a Gaussian likelihood over y:

$$p(Y|X, w; \sigma^2) \propto \exp(-\frac{1}{2\sigma^2}||Y - Xw||^2)$$

$$\nabla_w p(Y|X, w; \sigma^2) = -\frac{1}{\sigma^2}X^T(Y - Xw) = 0$$

In the general case, X is an $(m \times n)$ matrix with m examples of n-dimensional inputs, and Y is a vector of corresponding outputs.

$$\begin{aligned} &||y - Xw||_2^2 \\ &= (y - Xw)^T(y - Xw) \\ &= y^T y - 2(Xw)^T y + (Xw)^T(Xw) \end{aligned}$$

Taking the derivative of this with respect to w, we get:

$$\begin{aligned} -X^T Y + (X^T X)w &= 0 \\ A &= (X^T X)^{-1} X^T Y \end{aligned}$$

$(X^T X)^{-1} X^T$ is the Moore-Penrose pseudo-inverse.

2 Bayesian Linear Regression

On top of linear regression, make two assumptions:

- Training examples (x, y) come from a noisy underlying distribution $y = Xw + \epsilon$, where $\epsilon \sim N(0, \sigma_n^2)$
- Assume linear weights comes from a zero-mean Gaussian distribution $w \sim N(0, \Sigma_p)$ (prior)

This means $p(y|w, x) = N(x^T w, \sigma_n^2 I)$
Computing the posterior $w \sim N(\mu_n, \Sigma_n)$ for this, we get:

$$\mu_n = (X^T X + \sigma_n^2 \Sigma_p^{-1})^{-1} X^T Y$$
$$\Sigma_n = \sigma_n^2 (X^T X + \sigma_n^2 \Sigma_p^{-1})^{-1}$$

This comes from Maximum A Posteori Estimation, with a Gaussian likelihood on w:

$$p_{BLR} = p_{LR} \cdot p_{\text{prior}} \propto \exp(\text{LR terms}) \exp(\text{prior terms}) = \exp(\text{LR terms} + \text{prior terms})$$

$$\begin{aligned} p(w|Y; X, \sigma^2) &\propto \exp\left(-\frac{1}{2\sigma^2}||y - Xw||^2 - \frac{1}{2} w^T \Lambda w\right) \\ &= \exp\left(-\frac{1}{2\sigma^2}(y^T y - 2y^T X w + w^T X^T X w + \sigma^2 w^T \Lambda w)\right) \\ &= \exp\left(-\frac{1}{2\sigma^2}(w^T (X^T X + \sigma^2 \Lambda) w - 2y^T X w + y^T)\right) \end{aligned}$$

TODO: Finish this derivation
Reference link
Note that this is equivalent to regularized linear regression. Regularization aka weight decay is the result of placing a zero-mean prior on the model weights!

3 Non-linear mappings

To extend this framework to non-linear features, replace X with $\Phi(X)$. For a prediction of y at x:

$$p(y|x, X, Y) = N(\phi(x)^T A^{-1} \phi(X)^T Y, \sigma_n^2 (1 + \phi(x)^T A^{-1} \phi(x)))$$
$$A = \phi(X)^T \phi(X) + \sigma_n^2 \Sigma_p^{-1}$$

Here, $A \in R^{n \times n}$. The most expensive step in inference is inverting A, which is $O(n^3)$.

4 Kernel trick

The kernel trick makes use of the fact that the non-linear mapping $\phi(X)$ only appears in the inference equations as inner products: e.g. $\phi(x)^T A^{-1} \phi(x)$ and $\phi(X)^T \phi(X)$.
We use a kernel / covariance function that can compute an inner product in closed form.

$$k(x_i, x_j) = (\Sigma^{1/2} \phi(x_i))^T (\Sigma^{1/2} \phi(x_i)) = \phi(x_i)^T \Sigma \phi(x_j)$$

TODO: Elaborate on kernel trick
Applying this trick to Bayesian Linear Regression leads Gaussian Process Regression.
TODO: Derive inference equations

$$p(y|x, X, Y) = N(k^T G^{-1} Y, k(x, x) - k^T G^{-1} k + \sigma_n^2)$$

$$K = [k(x_i, x_j)]_{i,j=1}^m, k = [k(x, x_i)]_{i=1}^m, G = K + \sigma_n^2 I$$

Here, $G \in R^{m \times m}$, and the most expensive step is in inverting G, which is $O(m^3)$. Notice that when $m >> n$, Gaussian Process Regression with the kernel trick is actually more expensive than directly projecting using $\phi(X)$! However, this assumes the projection is finite-dimensional, which some popular kernels (e.g. RBF kernel, aka squared exponential) are not. Because of this, we can't just directly compute $\phi(X)$.
The RBF / squared exponential function is infinite-dimensional because in order to write it in a form $\phi(X)^T\phi(X)$ (inner product between two vectors, i.e. linear), we would have to take an infinitely long Taylor series. Details here.

5 Incremental updates

Assume we have a finite-dimensional kernel function. In this case, if $m >> n$ (i.e. number of examples is much larger than dimensionality of training examples), it may be better to ditch the kernel function and compute the projections $\phi(X)$ directly. Here, we will derive an incremental method for computing the A, b matrices
Recall:

$$p(y|x, X, Y) = N(\phi(x)^T A^{-1}\phi(X)^T Y, \sigma_n^2(1 + \phi(x)^T A^{-1}\phi(x)))$$
$$A = \phi(X)^T\phi(X) + \sigma_n^2\Sigma_p^{-1}$$

Here, σ_n^2 is the assumed noise in y, and Σ_p is the covariance of the prior over our linear weights. When a new sample (x_t, y_t) comes in, we can decompose the update as follows:

$$\phi(X_t) = \begin{bmatrix}\phi(X_{t-1})^T & \phi(x_t)\end{bmatrix}$$
$$Y_t = \begin{bmatrix}Y_{t-1}^T & y_t\end{bmatrix}$$

As shown above, to compute the posterior distribution we only need two values that are based on the training data:

$$A = \phi(X)^T\phi(X) + \sigma_n^2\Sigma_p^{-1}, b = \phi(X)^T Y$$
$$p(y|x, X, Y) = N(\phi(x)^T A^{-1} b, \sigma_n^2(1 + \phi(x)^T A^{-1}\phi(x)))$$

To update A, b incrementally:

$$\begin{aligned} A_t &= \begin{bmatrix}\phi(X_{t-1}) & \phi(x_t)\end{bmatrix}\begin{bmatrix}\phi(X_{t-1}\\ \phi(x_t)\end{bmatrix} + \sigma_n^2\Sigma_p^{-1} \\ &= \phi(X_{t-1})^T\phi(X_{t-1}) + \phi(x_t)^T\phi(x_t) + \sigma_n^2\Sigma_p^{-1} \\ &= A_{t-1} + \phi(x_t)^T\phi(x_t)\end{aligned}$$

$$\begin{aligned} b_t &= \phi(X_t)^T Y_t \\ &= \begin{bmatrix}\phi(X_{t-1}) & \phi(x_t))\end{bmatrix}\begin{bmatrix}Y_{t-1}\\ y_t\end{bmatrix} \\ &- \phi(X_{t-1})^T Y_{t-1} + \phi(x_t)^T y_t \\ &= b_{t-1} + \phi(x_t)^t y_t\end{aligned}$$

Thus, the updates to both A and b are rank-one updates. Set $A_0 = \sigma_n^2\Sigma_p^{-1}, b_0 = 0$.
However, this doesn't make training any more efficient because the problem of computing A^{-1} with $O(n^3)$ complexity remains. The solution to this is to update the Cholesky factor of A instead, since this can be used to compute an "inverse" without actually inverting the matrix.

"Inverting" a matrix with Cholesky factorization

Most of the time when we invert a matrix, it's for the purpose of solving a linear equation, i.e. multiplying that inverse with a vector.

$$Ax = b \rightarrow x = A^{-1}b$$

Cholesky factorization gives us a positive definite matrix as an inner product of an upper triangular (in this case; lower-triangular also used) matrix $R \in (n \times n)$:

$$A = R^T R$$

The trick for using Cholesky factorization to "invert" matrices is to notice that $Rx = b$ is actually solvable without explicitly inverting the matrix - since R is upper triangular, the bottom row defines a simple equation $x_j = b_j / R_{jj}$. This can then be substituted into the second row from the bottom with substitution. So solving $Rx = b$, then $R^T z = b$ is equivalent to computing $(R^T R)^{-1} b$. Given this factorization, we can multiply the inverse using forward and backward substitution. This makes the "inverse" $O(n^2)$.

$$\begin{aligned} A = R^T R \rightarrow A^{-1} &= R^{-1} R^{-T} \\ A^{-1} v &\rightarrow R^{-1} R^{-T} v \end{aligned}$$

Incremental Cholesky updates

$$\begin{aligned} A &= R^T R \\ A_t &= R_t^T R_t \\ &= A_{t-1} + \phi(x_t)^T \phi(x_t) \\ &= R_{t-1}^T R_{t-1} + \phi(x_t)^T \phi(x_t) R_t^T R_t && = \tilde{R}_t^T \tilde{R}_t \\ \tilde{R}_t &= \begin{bmatrix} R_{t-1} \phi(x_t) \end{bmatrix} \end{aligned}$$

We want to transform:

$$R_t^T R_t = \begin{bmatrix} R_{t-1} & \phi(x_t) \end{bmatrix}^T \begin{bmatrix} R_{t-1} \\ \phi(x_t) \end{bmatrix} \rightarrow \begin{bmatrix} R_t & 0 \end{bmatrix}^T \begin{bmatrix} R_t \\ 0 \end{bmatrix}$$

We can do this by applying an orthogonal matrix/transformation, like the ones used for QR factorization: Givens rotations or Householder reflections. For orthogonal matrices: $Q^{-1} = Q^T, Q^T Q = I$.

$$\begin{aligned} R^T R &= (Q \begin{bmatrix} R_t & 0 \end{bmatrix})^T (Q \begin{bmatrix} R_t \\ 0 \end{bmatrix}) \\ &= \begin{bmatrix} R_t & 0 \end{bmatrix} Q^T Q_t \begin{bmatrix} R_t \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} R_t & 0 \end{bmatrix} \begin{bmatrix} R_t \\ 0 \end{bmatrix} \end{aligned}$$

Givens rotations

Givens rotations are a method used to zero out subdiagonal elements, commonly employed in QR factorization. Main idea is that it multiplies two rows of a matrix by a rotation matrix (embedded inside an identity matrix), to zero out a single element in a column. Note that although Givens rotations are usually used to zero out subdiagonals, in this use case they are used to zero out an entire row, one element at a time. Reference link

The main intuitive difference between Givens rotations and Householder reflections (another method commonly used for QR) is that Givens rotations zero out one element at a time while affecting only two rows at a time, while Householder reflections zero out entire subdiagonals on a column while affecting the entire matrix. On a dense matrix, Householder reflections are more efficient (by

about 50%). Givens rotations can be preferable for their parallelizability, or for sparse matrices (which have fewer elements to zero out).
TODO: Figure out how exactly to zero out entire row with Givens rotations. My current implementation just uses built-in QR factorization.

Results

By incrementally updating the Cholesky factor of A, we get the most expensive step down to complexity $O(n^2)$, where n is dimensionality of non-linear mapping. This means we can compute an exact Gaussian Process Regression in $O(mn^2)$! But this is assuming that we have a finite-dimensional kernel, which is not the case for the most popular kernel function: RBF / squared exponential:

$$k(x_i, x_j) = \sigma_f^2 \exp(-\frac{1}{2}(x_i - x_j)^T M(x_i - x_j))$$

where M is a diagonal matrix with $M_{ii} = l_i^{-2}$ (length scales). We can deal with this with a finite-dimensional approximation.

6 Approximating RBF kernel

"The feature mapping for the popular RBF kernel is infinite-dimensional and thus cannot be explicitly computed. Rahimi and Recht (2008a), however, demonstrate that the RBF and other shift invariant kernels $k(x_i, x_j) = k(x_i - x_j)$ can be approximated to an arbitrary precision using a finite dimensional random feature mapping."
This random feature mapping is done in the spectral (frequency) domain:

$$k(x_i - x_j) = E_\omega \left[z_\omega(x_i) \quad z_\omega(x_j) \right]$$
$$z_\omega(x) = \left[\cos(\omega^T x) \quad \sin(\omega^T x) \right]$$

ω is drawn according to some probability distribution μ that is derived with the inverse Fourier transform (not sure how this works)
We can get an approximation by averaging over D projections:

$$\phi(x) = \frac{1}{\sqrt{D}} \left[z_{\omega_1}(x) \quad \dots z_{\omega_D}(x) \right]$$

The random feature mapping $\phi(X)$ that corresponds to the RBF kernel is shown in the paper. Using this mapping, we can perform incremental updates with complexity $O(D^2)$, where D is a parameter that trades off between approximation accuracy and computation speed.
TODO: Put phi equation for rbf here. How are length scales computed?

7 Comparison to other GP extensions

Two methods cited at bottom of Black-DROPS paper:

- "Patchwork Kriging for Large-scale Gaussian Process Regression"

  Partition input domain into multiple regions, with different local GP fitted to each region. Enforce continuity between regions with pseudo-observations at borders.

  Complexity is $O(KM^3)$, for updating K local GPs of dimensions M each.

  Need to read paper. For inference/training, it seems like each sample updates every local GP. For prediction, do we only use the local GP?

- "Distributed Gaussian Processes", Deisenroth and Ng

  Not sure what they're doing here; looks complicated. Basic idea seems to be decomposition/parallelization of GP solve. Do they make approximations?

Note that both of these methods seem to be focused on batch training, instead of incremental. It seems intuitive that incremental updates would be more efficient, if a formulation is possible.
EPC uses Locally Weighted Projection Regression - why? They note that LWPR is $O(dim(z))$, where z is input dimensionality??
The I-SSGPR paper shows that it is both faster and learns faster/better than LWPR.
"Although LWPR can more accurately represent accumulated experiences via its local models, we also consider ISSGPR due to its potential for faster model adaptation."