Quick Start to Gaussian Process Regression

A quick guide to understanding Gaussian process regression (GPR) and using scikit-learn's GPR package

Hilarie Sit Jun 19, 2019 · 6 min read ★

Gaussian process regression (GPR) is a nonparametric, Bayesian approach to regression that is making waves in the area of machine learning. GPR has several benefits, working well on small datasets and having the ability to provide uncertainty measurements on the predictions.

for every parameter in a function, the Bayesian approach infers a probability distribution over all possible values. Let's assume a linear function: $y=wx+\epsilon$. How the Bayesian approach works is by specifying a ***prior distribution***, $p(w)$, on the parameter, w, and relocating probabilities based on evidence (*i.e.* observed data) using Bayes' Rule:

$$p(w|y,X) = \frac{p(y|X,w)p(w)}{p(y|X)}$$

$$\text{posterior} = \frac{\text{likelihood} \times \text{prior}}{\text{marginal likelihood}}$$

Calculating posterior distribution with Bayes' Rule [1]

The updated distribution $p(w|y, X)$, called the ***posterior distribution***, thus incorporates information from both the prior distribution and the dataset. To get predictions at unseen points of interest, x^*, the ***predictive distribution*** can be calculated by weighting all possible predictions by their calculated posterior distribution [1]:

$$p(f^*|x^*,y,X) = \int_w p(f^*|x^*,w)p(w|y,X)dw$$

Calculating predictive distribution, f* is prediction label, x* is test observation [1]

The prior and likelihood is usually assumed to be Gaussian for the integration to be tractable. Using that assumption and solving for the predictive distribution, we get a Gaussian distribution, from which we can obtain a point prediction using its mean and an uncertainty quantification using its variance.

Gaussian process regression is nonparametric (*i.e.* not limited by a functional form), so rather than calculating the probability distribution of parameters of a specific function, GPR calculates the probability distribution over all admissible functions that

distribution on our points of interest.

There are several libraries for efficient implementation of Gaussian process regression (*e.g.* scikit-learn, Gpytorch, GPy), but for simplicity, this guide will use scikit-learn's Gaussian process package [2].

```
import sklearn.gaussian_process as gp
```

In GPR, we first assume a ***Gaussian process prior***, which can be specified using a mean function, $m(x)$, and covariance function, $k(x, x')$:

$$f(x) \sim GP(m(x), k(x, x'))$$

Labels drawn from Gaussian process with mean function, m, and covariance function, k [1]

More specifically, a Gaussian process is like an infinite-dimensional multivariate Gaussian distribution, where any collection of the labels of the dataset are joint Gaussian distributed. Within this GP prior, we can incorporate prior knowledge about the space of functions through the selection of the mean and covariance functions. We can also easily incorporate independently, identically distributed *(i.i.d)* Gaussian noise, $\epsilon \sim N(0, \sigma^2)$, to the labels by summing the label distribution and noise distribution:

$$y \sim GP(m(x), k(x, x') + \delta_{ij}\sigma_n^2)$$

Labels with noise: $y = f(x) + \epsilon$, $\epsilon \sim N(0, \sigma^2)$ [1]

Dataset

The dataset consists of observations, *X,* and their labels, *y,* split into "training" and "testing" subsets:

```
# X_tr <-- training observations [# points, # features]
# y_tr <-- training labels [# points]
# X_te <-- test observations [# points, # features]
# y_te <-- test labels [# points]
```

way [1]:

$$\begin{bmatrix} y \\ f_* \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu \\ \mu_* \end{bmatrix}, \begin{bmatrix} K(X,X) + \sigma_n^2 I & K(X,X_*) \\ K(X_*,X) & K(X_*,X_*) \end{bmatrix} \right)$$

GP prior rewritten: multivariate distribution of training and testing points

Here, K is the covariance kernel matrix where its entries correspond to the covariance function evaluated at observations. Written in this way, we can take the training subset to perform model selection.

Model Selection

The form of the mean function and covariance kernel function in the GP prior is chosen and tuned during ***model selection***. The mean function is typically constant, either zero or the mean of the training dataset. There are many options for the covariance kernel function: it can have many forms as long as it follows the properties of a kernel (*i.e.* semi-positive definite and symmetric). Some common kernel functions include constant, linear, square exponential and Matern kernel, as well as a composition of multiple kernels.

A popular kernel is the composition of the constant kernel with the radial basis function (RBF) kernel, which encodes for smoothness of functions (*i.e.* similarity of inputs in space corresponds to the similarity of outputs):

$$k(x, x') = \sigma_f^2 \, exp(-\frac{1}{2\ell^2} ||x - x'||^2)$$

Constant times RBF kernel function

This kernel has two hyperparameters: signal variance, σ^2, and lengthscale, l. In scikit-learn, we can chose from a variety of kernels and specify the initial value and bounds on their hyperparameters.

```
kernel = gp.kernels.ConstantKernel(1.0, (1e-1, 1e3)) *
gp.kernels.RBF(10.0, (1e-3, 1e3))
```

normalize_y refers to the constant mean function — either zero if False or the training data mean if True.

```
model = gp.GaussianProcessRegressor(kernel=kernel,
n_restarts_optimizer=10, alpha=0.1, normalize_y=True)
```

A popular approach to tune the hyperparameters of the covariance kernel function is to maximize the log marginal likelihood of the training data. A gradient-based optimizer is typically used for efficiency; if unspecified above, the default optimizer is *'fmin_l_bfgs_b'*. Because the log marginal likelihood is not necessarily convex, multiple restarts of the optimizer with different initializations is used (*n_restarts_optimizer)*.

```
model.fit(X_tr, y_tr)
params = model.kernel_.get_params()
```

The tuned hyperparameters of the kernel function can be obtained, if desired, by calling *model.kernel_.get_params()*.

Inference

To calculate the predictive posterior distribution, the data and the test observation is conditioned out of the posterior distribution. Again, because we chose a Gaussian process prior, calculating the predictive distribution is tractable, and leads to normal distribution that can be completely described by the mean and covariance [1]:

$$f^*|X,y,X^* \sim \mathcal{N}(\bar{f}^*, \Sigma^*)$$

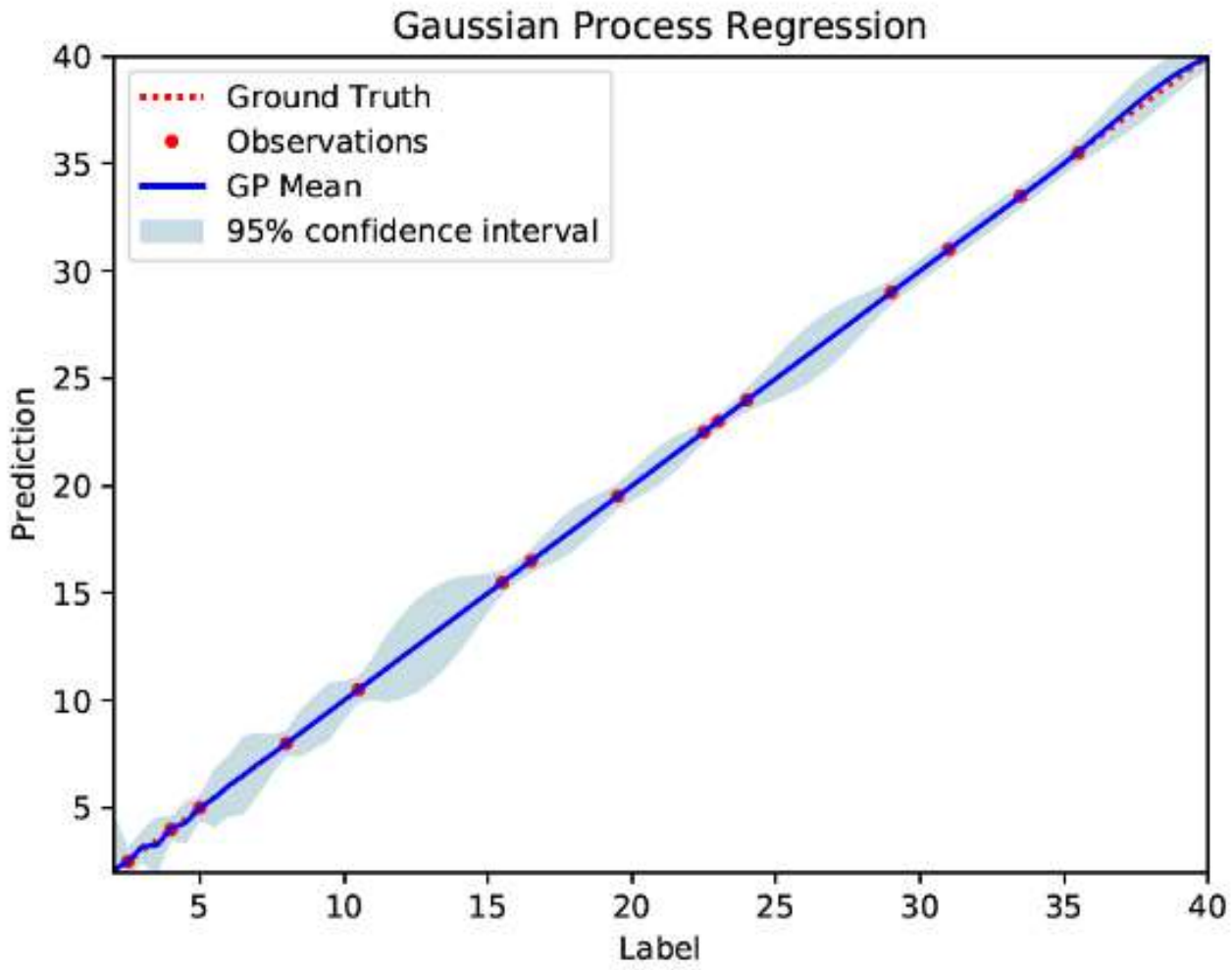
$$\bar{f}^* = \mu^* + K(X^*,X)[K(X,X)+\sigma_n^2 I]^{-1}(y-\mu)$$

$$\Sigma^* = K(X^*,X^*) - K(X^*,X)[K(X,X)+\sigma_n^2 I]^{-1}K(X,X^*)$$

Predictive posterior distribution for GPR [1]

The predictions are the means f_bar*, and variances can be obtained from the diagonal of the covariance matrix Σ*. Notice that calculation of the mean and variance requires

```
y_pred, std = model.predict(X_te, return_std=True)
```



Note that the standard deviation is returned, but the whole covariance matrix can be returned if *return_cov=True*. The 95% confidence interval can then be calculated: 1.96 times the standard deviation for a Gaussian.

To measure the performance of the regression model on the test observations, we can calculate the mean squared error (MSE) on the predictions.

```
MSE = ((y_pred-y_te)**2).mean()
```

References:

[2] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., et. al., Scikit-learn: Machine learning in python (2011), Journal of Machine Learning Research