Gaussian Processes (Contd.), Intro to Latent Variable Models

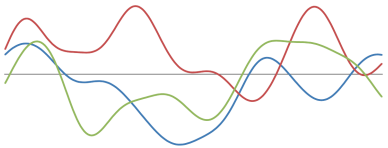
Piyush Rai

Topics in Probabilistic Modeling and Inference (CS698X)

Feb 6, 2018

Gaussian Process

- Denoted as $\mathcal{GP}(\mu, \kappa)$. It is a distribution over functions
  - Defined by mean function μ and covariance function κ



- Mean function μ models the "average" function f drawn from $\mathcal{GP}(\mu, \kappa)$

$$\mu(\boldsymbol{x}) = \mathbb{E}_{f \sim \mathcal{GP}(\mu,\kappa)}[f(\boldsymbol{x})]$$

- Cov. function κ models "shape/smoothness" of these functions
  - $\kappa(.,.)$ is a function that computes similarity between two inputs (just like a kernel function)
  - Note: $\kappa(.,.)$ needs to be positive definite (just like kernel functions)
- An appealing aspect: Can learn "hyperparameters" μ and κ (i.e., can learn the kernel)

Gaussian Process

- A random function f drawn from $\mathcal{GP}(\mu, \kappa)$, has the following "finite dimensional marginal"

$$\begin{bmatrix} f(\boldsymbol{x}_1) \\ f(\boldsymbol{x}_2) \\ \vdots \\ f(\boldsymbol{x}_N) \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu(\boldsymbol{x}_1) \\ \mu(\boldsymbol{x}_2) \\ \vdots \\ \mu(\boldsymbol{x}_N) \end{bmatrix}, \begin{bmatrix} \kappa(\boldsymbol{x}_1, \boldsymbol{x}_1) \dots \kappa(\boldsymbol{x}_1, \boldsymbol{x}_N) \\ \kappa(\boldsymbol{x}_2, \boldsymbol{x}_1) \dots \kappa(\boldsymbol{x}_2, \boldsymbol{x}_N) \\ \vdots \quad \ddots \quad \vdots \\ \kappa(\boldsymbol{x}_N, \boldsymbol{x}_1) \dots \kappa(\boldsymbol{x}_N, \boldsymbol{x}_N) \end{bmatrix} \right)$$

- Note that, unlike traditional parametric definitions $y_n = \boldsymbol{w}^\top \boldsymbol{x}_n + \epsilon_n$, here the "function" f is defined in terms of the joint marginal distribution of a set of outputs given the corresponding inputs
  - Weight-space view vs function-space view of input to output mapping
- We can also write the above more compactly as $\mathbf{f} \sim \mathcal{N}(\boldsymbol{\mu}, \mathbf{K})$ where

$$\mathbf{f} = \begin{bmatrix} f(\boldsymbol{x}_1) \\ f(\boldsymbol{x}_2) \\ \vdots \\ f(\boldsymbol{x}_N) \end{bmatrix}, \boldsymbol{\mu} = \begin{bmatrix} \mu(\boldsymbol{x}_1) \\ \mu(\boldsymbol{x}_2) \\ \vdots \\ \mu(\boldsymbol{x}_N) \end{bmatrix}, \mathbf{K} = \begin{bmatrix} \kappa(\boldsymbol{x}_1, \boldsymbol{x}_1) \dots \kappa(\boldsymbol{x}_1, \boldsymbol{x}_N) \\ \kappa(\boldsymbol{x}_2, \boldsymbol{x}_1) \dots \kappa(\boldsymbol{x}_2, \boldsymbol{x}_N) \\ \vdots \quad \ddots \quad \vdots \\ \kappa(\boldsymbol{x}_N, \boldsymbol{x}_1) \dots \kappa(\boldsymbol{x}_N, \boldsymbol{x}_N) \end{bmatrix}$$

  Note: $\mathbf{K}$ is also called the kernel matrix. $K_{nm} = \kappa(\boldsymbol{x}_n, \boldsymbol{x}_m)$
- Therefore the finite-dim version of the GP $p(f) = \mathcal{GP}(\mu, \kappa)$ is $p(\mathbf{f}) = \mathcal{N}(\boldsymbol{\mu}, \mathbf{K})$

Using GP as a Prior Distribution in Regression/Classification

- Consider a problem with data of the form $(\mathbf{X}, \boldsymbol{y}) = \{\boldsymbol{x}_n, y_n\}_{n=1}^N$ using a discriminative model
- Typically the probability distribution of y_n given $\boldsymbol{x}_n$ depends on a "score" $f_n = f(\boldsymbol{x}_n)$, e.g.,

$$\begin{aligned} \text{Regression} \quad & p(y_n|\boldsymbol{x}_n) = p(y_n|f_n) &&= \mathcal{N}(y_n|f_n, \sigma^2) \\ \text{Classification} \quad & p(y_n|\boldsymbol{x}_n) = p(y_n|f_n) &&= \text{Bernoulli}(y_n|\sigma(f_n)) \end{aligned}$$

- Suppose the vector of scores is defined as $\boldsymbol{f} = [f(\boldsymbol{x}_1), \ldots, f(\boldsymbol{x}_N)]$
- Can use GP as a prior on the function f, or equivalently on the score vector $\boldsymbol{f}$

$$p(\mathbf{f}) = \mathcal{N}(\mathbf{0}, \mathbf{K}) \qquad \text{(assuming zero mean function)}$$

- Can now combine this prior with the likelihood $p(\boldsymbol{y}|\boldsymbol{f})$ to compute
  - Posterior over the function f, which is given in the form $p(\boldsymbol{f}|\boldsymbol{y}) \propto p(\boldsymbol{f})p(\boldsymbol{y}|\boldsymbol{f})$
  - Posterior predictive $p(y_*|\boldsymbol{y}) = \int p(y_*|f_*)p(f_*|\boldsymbol{y})df_*$ where $p(f_*|\boldsymbol{y}) = \int \underbrace{p(f_*|\boldsymbol{f})}_{\substack{\text{Always} \\ \text{Gaussian}}} p(\boldsymbol{f}|\boldsymbol{y})d\boldsymbol{f}$

Scalability Aspects of GP

- Computational costs in some steps of GP based models scale in the size of training data
  - E.g., test time prediction in GP regression takes $O(N)$ time

$$
\begin{aligned}
p(y_*|\boldsymbol{y}) &= \mathcal{N}(y_*|\mu_*, \sigma_*^2) \\
\mu_* &= \mathbf{k}_*{}^\top \mathbf{C}_N^{-1} \boldsymbol{y} \qquad (O(N) \text{ cost assuming } \mathbf{C}_N^{-1} \text{ is pre-computed}) \\
\sigma_*^2 &= k(\boldsymbol{x}_*, \boldsymbol{x}_*) + \sigma^2 - \mathbf{k}_*{}^\top \mathbf{C}_N^{-1} \mathbf{k}_*
\end{aligned}
$$

- GP models often require matrix inversions - takes $O(N^3)$ time. Storage also requires $O(N^2)$ space
- A lot of work on speeding up GPs. For some recent advances, you may check out
  - "Thoughts on Massively Scalable Gaussian Processes"
  - "Kernel Interpolation for Scalable Structured Gaussian Processes (KISS-GP)"
  - .. and references therein
- Note that nearest neighbor methods and kernel methods also face similar issues w.r.t. scalability
  - Many tricks to speed up kernel methods can be used for speeding up GPs too

GP: A few comments

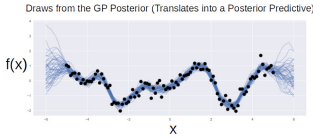
- GP is a nonparametric model. Why called "nonparametric"?
  - Complexity (representation size) of the function f grows in the size of training data
  - To see this, note the form of the GP predictions, e.g., predictive mean in GP regression

$$\mu_* = f(\boldsymbol{x}_*) = \mathbf{k}_*{}^\top \mathbf{C}_N^{-1} \boldsymbol{y} = \mathbf{k}_*{}^\top \boldsymbol{\alpha} = \sum_{n=1}^{N} \alpha_n k(\boldsymbol{x}_*, \boldsymbol{x}_n)$$

  - It implies that $f(.) = \sum_{n=1}^{N} \alpha_n k(., \boldsymbol{x}_n)$, which means f is written in terms of all training examples
  - Thus the representation size of f depends on the number of training examples
- In contrast, a parametric model has a size that doesn't grow with training data
  - E.g., a linear model learns a fixed-sized weight vector $\boldsymbol{w} \in \mathbb{R}^D$ (D parameters, size independent of N)
- Nonparametric models therefore are more flexible since their complexity is not limited beforehand
  - Note: Methods such as nearest neighbors and kernel SVMs are also nonparametric (but not Bayesian)
- GPs equivalent to infinitely-wide single hidden-layer neural net (under some technical conditions)

GP: A few other comments

- Can be thought of as Bayesian analogues of kernel methods
  - Can get estimate in the uncertainty in the function and its predictions



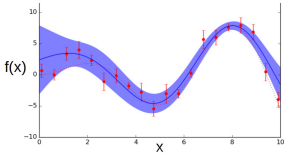
  - Can learn the kernel (by learning the hyperparameters of the kernels)
- Not limited to supervised learning problems
  - The function f could even be a mapping of an unknown quantity to an observed quantity

$$\boldsymbol{x}_n = f(\boldsymbol{z}_n) + \text{“noise”}$$

  where $\boldsymbol{z}_n$ is a latent representation of $\boldsymbol{x}_n$ (“GP latent variable models” for nonlin. dim. red.)
- Many mature implementations of GP exist. You may check out
  - GPML (MATLAB), GPsuff (MATLAB/Octave), GPy (Python), GPyTorch (PyTorch)

GPs are very versatile!

- GPs enable us to learn nonlinear functions while also capturing the uncertainty

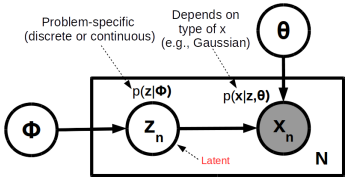


- Uncertainty can tell us where to acquire more training data to improve the function's estimate
  - Especially useful if we can't get too many training examples (e.g., expensive inputs and/or labels)
- This is very useful in a wide range of applications involving sequential decision-making
  - Active Learning: Learning a function by gathering the most informative training examples
  - Bayesian Optimization: Optimizing an expensive to evaluate functions (and maybe we don't even know it form) – boils down to simultaneous function learning and optimization
- We will look at some of these later during the semester

Latent Variable Models

Modeling Data via Latent Variables

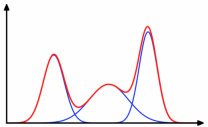
- Assume each observation $\boldsymbol{x}_n$ to be associated with a latent variable $\boldsymbol{z}_n$



- $\boldsymbol{z}_n$ is a latent representation or "encoding" of $\boldsymbol{x}_n$ (often reveals the latent structure of data)
  - $\boldsymbol{z}_n \in \{1, \ldots, K\}$ denotes the cluster $\boldsymbol{x}_n$ belongs to
  - $\boldsymbol{z}_n \in \mathbb{R}^K$ denotes a low-dimensional latent representation or latent "code" for $\boldsymbol{x}_n$
- Sometimes $\mathbf{Z} = \{\boldsymbol{z}_1, \ldots, \boldsymbol{z}_N\}$ called "local" variables; $\Theta = (\theta, \phi)$ called "global" variables
  - The goal is to learn these unknows given the observed data $\mathbf{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N\}$, i.e., $p(\mathbf{Z}, \Theta | \mathbf{X})$

Motivating Example 1: Mixture Model

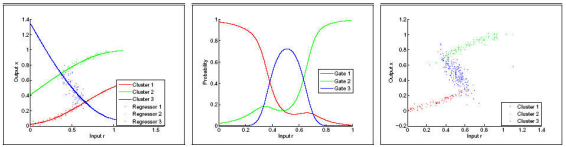
- Assume data $\{\boldsymbol{x}_n\}_{n=1}^N$ was generated from a mixture of K distributions $p(\boldsymbol{x}|\theta_1), \ldots, p(\boldsymbol{x}|\theta_K)$



- Don't know which component generated each $\boldsymbol{x}_n$ (o/w it is simply generative classification)
- Assume a latent random variable $\boldsymbol{z}_n \in \{1, \ldots, K\}$ denotes which component generated $\boldsymbol{x}_n$
- Here is a simple generative story for each $\boldsymbol{x}_n$, $n = 1, 2, \ldots, N$
  - First choose a mixture component $\boldsymbol{z}_n \in \{1, 2, \ldots, K\}$ as $\boldsymbol{z}_n \sim \text{multinoulli}(\boldsymbol{z}|\pi)$
  - Now generate $\boldsymbol{x}_n$ from that mixture component as $\boldsymbol{x}_n \sim p(\boldsymbol{x}|\theta_{\boldsymbol{z}_n})$
- Goal: Given data $\{\boldsymbol{x}_n\}_{n=1}^N$, learn the K distributions $(\theta_1, \ldots, \theta_K)$ and latent variables $\boldsymbol{z}_1 \ldots, \boldsymbol{z}_N$
- If each $p(\boldsymbol{x}|\theta_k)$ is a Gaussian $\Rightarrow$ Gaussian Mixture Model (used for probabilistic or "soft" clustering)

Motivating Example 2: Mixture of Experts

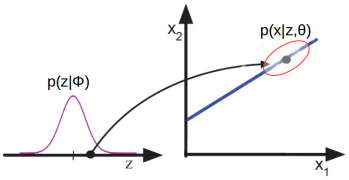
- Assume N inputs $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$ whose responses are $y_n, \ldots, y_N$
- Assume responses $y_n, \ldots, y_N$ are generated by a mixture of K linear regression models (experts)



- Each linear regression model handles one region of the input space (regions may have overlap)
- The resulting $\boldsymbol{x}$ to y mapping learned by such a mixture model is effectively a nonlinear model
- The generative story for each y_n conditioned on $\boldsymbol{x}_n$
  - First choose an expert $\boldsymbol{z}_n \in \{1, 2, \ldots, K\}$ as $\boldsymbol{z}_n \sim \text{multinoulli}(\boldsymbol{z}|\pi(\boldsymbol{x}_n))$ (note: π depends on $\boldsymbol{x}_n$ too)
  - Now generate y_n using the regression model of component K as $y_n|\boldsymbol{x}_n \sim p(y_n|\boldsymbol{x}_n, \boldsymbol{w}_{\boldsymbol{z}_n})$

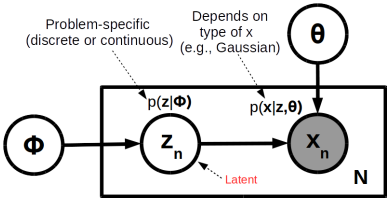
Motivating Example 3: Latent Factor Model

- Assume data $\boldsymbol{x}_n \in \mathbb{R}^D$ generated from a low-dimensional latent factor $\boldsymbol{z}_n \in \mathbb{R}^K$



- The $\boldsymbol{z}$ to $\boldsymbol{x}$ map can be a linear/nonlinear transformation
- Consider the following generative story for each $\boldsymbol{x}_n$, $n = 1, 2, \ldots, N$
  - First generate $\boldsymbol{z}_n$ from a K-dim distr. as $\boldsymbol{z}_n \sim p(\boldsymbol{z}|\phi)$
  - Now generate $\boldsymbol{x}_n$ from a D-dim distr. as $\boldsymbol{x}_n \sim p(\boldsymbol{x}|\boldsymbol{z}_n, \theta)$
- If $p(\boldsymbol{z}|\phi)$ and $p(\boldsymbol{x}|\boldsymbol{z}, \theta)$ are Gaussians and $\boldsymbol{z}$ to $\boldsymbol{x}$ map linear $\Rightarrow$ factor analysis or probabilistic PCA

Learning Parameters of Latent Variable Models



- Suppose we want to learn the "global" parameters $\Theta = (\theta, \phi)$ of this LVM
- While full posterior inference can be tried for Θ, let's keep it simple for now and do MLE/MAP
- The MLE would be for the above LVM would be

$$\hat{\Theta} = \arg\max_{\Theta} \sum_{n=1}^{N} \log p(\boldsymbol{x}_n|\Theta)$$

- It turns out that the above problem is a hard problem in general!

Why is MLE/MAP Hard for LVMs?

- First of all, note that when working with exp. family distributions, MLE turns out to be very easy
  - Reason: Usually log of an exp-fam distribution has simple algebraic form, easy to do MLE
- In the MLE/MAP for LVM, the goal is to solve $\hat{\Theta} = \arg\max_{\Theta} \sum_{n=1}^{N} \log p(\boldsymbol{x}_n|\Theta)$
- To do so, we first need $p(\boldsymbol{x}_n|\Theta)$ which is a marginal

$$\text{Discrete } \boldsymbol{z}_n: \quad p(\boldsymbol{x}_n|\Theta) = \sum p(\boldsymbol{x}_n, \boldsymbol{z}_n|\Theta) = \sum_{k=1}^{K} p(\boldsymbol{x}_n|\boldsymbol{z}_n = k, \Theta) p(\boldsymbol{z}_n = k|\Theta)$$

$$\text{Continuous } \boldsymbol{z}_n: \quad p(\boldsymbol{x}_n|\Theta) = \int p(\boldsymbol{x}_n, \boldsymbol{z}_n|\Theta) d\boldsymbol{z}_n = \int p(\boldsymbol{x}_n|\boldsymbol{z}_n, \Theta) p(\boldsymbol{z}_n|\Theta) d\boldsymbol{z}_n$$

- Causes problem! Even if $p(\boldsymbol{x}_n|\boldsymbol{z}_n, \Theta)$ and $p(\boldsymbol{z}_n|\Theta)$ (and thus the joint $p(\boldsymbol{x}_n, \boldsymbol{z}_n|\Theta)$) are exp-fam distributions, marginal $p(\boldsymbol{x}_n|\Theta)$ isn't in general an exp-fam distribution (Bar-Lev et al, 1994)
  - Consequently, $\log p(\boldsymbol{x}_n|\Theta)$ doesn't usually have a simple algebraic form amenable for MLE
- On the other hand, if someone gave us the "good guess" $\hat{\boldsymbol{z}}_n$ of $\boldsymbol{z}_n$, MLE becomes much simpler (that's the underlying idea behind the Expectation Maximization algorithm)