Scaling Multidimensional Gaussian Processes using Projected Additive Approximations

Elad Gilboa GILBOAE@ESE.WUSTL.EDU
Washington University, St. Louis, USA

Yunus Saatçi YUNUS.SAATCI@GMAIL.COM
University of Cambridge, Cambridge, UK

John P. Cunningham CUNNINGHAM@WUSTL.EDU
Washington University, St. Louis, USA

Abstract

Exact Gaussian Process (GP) regression has $\mathcal{O}(N^3)$ runtime for data size N, making it intractable for large N. Advances in GP scaling have not been extended to the multidimensional input setting, despite the preponderance of multidimensional applications. This paper introduces and tests a novel method of projected additive approximation to multidimensional GPs. We illustrate the power of this method on several datasets, achieving performance close to the naive Full GP at orders of magnitude less cost.

1. Introduction

Gaussian Processes (GP) have become a popular tool for nonparametric Bayesian regression. Naive GP regression has $\mathcal{O}(N^3)$ runtime (due to matrix inversions and determinants) and $\mathcal{O}(N^2)$ memory complexity, where N is the number of observations. At ten thousand or more, this problem is for all practical purposes intractable, given current hardware.

A significant amount of research has gone into sparse approximations, reducing run-time complexity to $\mathcal{O}(M^2N)$ for some $M \ll N$. For an excellent review of sparse GP approximations, see (Quiñonero-Candela & Rasmussen, 2005). All sparse approximation methods are based on the assumption of conditional independence of the training and test sets, given an active set of inducing inputs. As emphasized in (Quiñonero-Candela & Rasmussen, 2005), the results of these algorithms can depend strongly on the properties of the data. Since different assumptions fit different datasets, and since sparsity has by no means solved all efficiency issues for GPs, it is imperative to explore alternative avenues for attaining scalability.

The central aim of this paper is to introduce a new algorithm, based on the classical projection pursuit method, for *structured* GPs of multidimensional inputs. We say a GP is *structured* if its marginals $p(\mathbf{f}|\mathbf{X},\theta)$ contain exploitable structure that enables reduction in computational complexity (where $\mathbf{X} = \{\mathbf{x}_n\}_{n=1}^N, \mathbf{x}_n \in \mathbb{R}^d$ are the training inputs, θ is the vector of the hyperparameters, and $\mathbf{f}$ is the unknown function). While these structured GP methods are known in the case of scalar inputs, here we explore the nontrivial extensions required for multidimensional input spaces.

1.1. Gaussian Process Regression

In brief, GP regression is a Bayesian method for nonparametric regression, where a prior distribution over continuous functions is specified via a Gaussian process. The use of GP in machine learning is well described in (Rasmussen & Williams, 2006).

A GP is a distribution on f over an input space X such that any finite selection of input locations $\mathbf{x}_1, \ldots, \mathbf{x}_N \in X$ gives rise to a multivariate Gaussian density over the associated targets, i.e.,

$$p(f(\mathbf{x}_1), \ldots, f(\mathbf{x}_N)) = \mathcal{N}(\mathbf{m}_N, \mathbf{K}_N), \qquad (1)$$

where $\mathbf{m}_N = m(\mathbf{x}_1, \ldots, \mathbf{x}_N)$ is the mean vector and $\mathbf{K}_N = \{k(\mathbf{x}_i, \mathbf{x}_j)\}_{i,j}$ is the covariance matrix for mean function m and covariance function k. In this paper we are specifically interested in the computational burden

Proceedings of the 30th International Conference on Machine Learning, Atlanta, Georgia, USA, 2013. JMLR: W&CP volume 28.

of the basic equations for GP regression, which involve two steps. First, for given data $\mathbf{y} \in \mathbb{R}^N$ (making the standard assumption of zero-mean data, without loss of generality), we calculate the predictive mean and covariance at M unseen inputs as:

$$\boldsymbol{\mu}_\star = \mathbf{K}_{MN}\left(\mathbf{K}_N + \sigma_n^2\mathbf{I}_N\right)^{-1}\mathbf{y}, \quad (2)$$

$$\boldsymbol{\Sigma}_\star = \mathbf{K}_M - \mathbf{K}_{MN}\left(\mathbf{K}_N + \sigma_n^2\mathbf{I}_N\right)^{-1}\mathbf{K}_{NM}, \quad (3)$$

For model selection, since the covariance function $k(\cdot,\cdot;\theta)$ is parameterized by hyperparameters such as amplitude and lengthscale (which we group into θ), we must consider the log marginal likelihood $Z(\theta)$:

$$\log Z(\theta) = -\frac{1}{2}\left[\mathbf{y}^\top(\mathbf{K}_N + \sigma_n^2\mathbf{I}_N)^{-1}\mathbf{y} + N\log(2\pi) + \log|\mathbf{K}_N + \sigma_n^2\mathbf{I}_N|\right]. \quad (4)$$

Here we use this marginal likelihood to optimize over the hyperparameters in the usual way (Rasmussen & Williams, 2006). The runtime of GP regression and hyperparameter learning is $\mathcal{O}(N^3)$ due to the term $\left(\mathbf{K}_N + \sigma_n^2\mathbf{I}_N\right)^{-1}$, which is present in all equations.

1.2. Gauss-Markov Processes

We briefly review the use of Gauss-Markov Processes for efficient GP regression on scalar inputs, as a starting point for the multidimensional extensions in Section 2. Although Gauss-Markov Processes are well studied, their use for exact and efficient GP regression is underappreciated. A GP with a kernel corresponding to a state-space model can be viewed as a Gauss-Markov Process, enabling linear runtime. Gauss-Markov Processes can be viewed as the solution of an order-m linear, stationary stochastic differential equation (SDE), given by:

$$\frac{d\mathbf{z}(x)}{dx} = \mathbf{A}\mathbf{z}(x) + \mathbf{L}w(x), \quad (5)$$

where

$$\mathbf{z}(x) = \left[f(x), \frac{df(x)}{dx}, \ldots, \frac{d^{m-1}f(x)}{dx^{m-1}}\right]^\top, \quad (6)$$

and where $\mathbf{L} = [0, 0, \ldots, 1]$, $\mathbf{A}$ is the coefficient matrix, and $w(x)$ is a zero-mean white noise process. Eq. (**??**) shows that, given knowledge of $f(x)$ and its first m derivatives, we have Markov structure in the graph underlying GP inference, which will enable all efficiency gains in this section.

Earlier work (Hartikainen & Särkkä, 2010; Saatci, 2011), derived the SDEs corresponding to several commonly used covariance functions including the Matérn family and spline kernels, and good approximate SDEs corresponding to the exponentiated-quadratic kernel. Once the SDE is known, the Kalman filtering and Rauch-Tung-Striebel (RTS) smoothing algorithms (which correspond to belief propagation) can be used to perform GP regression in $\mathcal{O}(N)$ time and memory, a noteworthy leap in efficiency[1].

2. GP Regression for Multidimensional State-Space Models

For the purposes of extending one-dimensional Gauss-Markov Processes (Sec. 1.2) to multiple dimensions, we will initially consider the simplifying assumption of additivity, for which we will present two efficient algorithms. However, since the assumption of additivity is usually too strong for real problems, in Section 2.2 we will extend the modeling power by considering an additive model in a feature space.

2.1. Additive GP regression

Additive GP regression can be described using the following generative model:

$$y_i = \sum_{d=1}^{D}\mathbf{f}_d(\mathbf{X}_{i,d}) + \epsilon \qquad i = 1,\ldots,N, \quad (7)$$

$$\mathbf{f}_d(\cdot) \sim \mathcal{GP}\left(\mathbf{0}, k_d(\mathbf{x}_d, \mathbf{x}'_d; \theta_d)\right) \qquad d = 1,\ldots,D,$$

$$\epsilon \sim \mathcal{N}(0, \sigma_n^2),$$

where $\mathbf{X}_{i,d}$ is the d-th component of input i, $k_d(\cdot,\cdot)$ is the kernel of the scalar GP along dimension d, θ_d represent the dimension-specific hyperparameters, and σ_n^2 is the (global) noise hyperparameter (Duvenaud et al., 2011). The resulting model regresses a sum of D Gauss-Markov Processes (which are independent a priori), where $D > 1$ is the dimensionality of the input space.

As described in (Hastie et al., 2009), a nonparametric regression technique (such as the spline smoother) which allows a scalable fit over a scalar input space can be used to fit an additive model over a D-dimensional space with the same overall asymptotic complexity, by means of the backfitting algorithm. Surprisingly, the application of backfitting (Algorithm 1) can be proved to converge to the exact posterior mean. The easiest way to see this is by viewing (Algorithm 1) as a Gauss-Seidel iteration. As a reminder, Gauss-Seidel is an iterative technique to solve linear systems, in this case

[1] Note that the Gauss-Markov Process framework requires sorted input points. Else, a preprocessing step of $\mathcal{O}(N\log N)$ is needed.

Algorithm 1 Efficient Computation of Additive GP Posterior Mean via Backfitting

Input: Training data $\{\mathbf{X}, \mathbf{y}\}$. Suitable covariance function. Hypers $\theta = \bigcup_{d=1}^{D}\{\theta_d\} \cup \sigma_n^2$.
outputs: Posterior training means: $\sum_{d=1}^{D} \boldsymbol{\mu}_d$, where $\boldsymbol{\mu}_d \equiv \mathbb{E}(\mathbf{f}_d|\mathbf{y}, \mathbf{X}, \theta_d, \sigma_n^2)$.

Zero-mean the targets $\mathbf{y}$
Initialise the $\boldsymbol{\mu}_d$ (e.g. to $\mathbf{0}$)
while The change in $\boldsymbol{\mu}_d$ is above a threshold **do**
  for $d = 1, \ldots, D$ **do**
    $\boldsymbol{\mu}_d \leftarrow \mathbb{E}(\mathbf{f}_d|\mathbf{y} - \sum_{j\neq d}\boldsymbol{\mu}_j, \mathbf{X}_{:,d}, \theta_d, \sigma_n^2)$
  end for
end while

solving for the exact posterior mean. It is precisely the additive Gauss-Markov Process structure that makes the backfitting update equivalent to a Gauss-Seidel step, the details of which can be found in our preliminary work (Saatci, 2011).

To calculate posterior variances and learn hyperparameters, we must investigate further. The observed variables are the targets $\mathbf{y}$, and the latent variables $\mathbf{Z}$ consist of the D Markov chains:

$$\mathbf{Z} \equiv \left(\underbrace{\mathbf{z}_1^1, \ldots, \mathbf{z}_1^N}_{\equiv \mathbf{Z}_1}, \underbrace{\mathbf{z}_2^1, \ldots, \mathbf{z}_2^N}_{\equiv \mathbf{Z}_2}, \ldots, \underbrace{\mathbf{z}_D^1, \ldots, \mathbf{z}_D^N}_{\equiv \mathbf{Z}_D} \right). \quad (8)$$

The true posterior $p(\mathbf{Z}_1, \ldots, \mathbf{Z}_D|\mathbf{y}, \mathbf{X}, \theta)$ is hard to handle computationally because all variables $\mathbf{Z}_i$ are coupled in the posterior. Although everything is still Gaussian, we are no longer able to use the efficient state-space methods of Section 1.2, returning us to the original computational intractability at large N. Thus, we require an approximate inference technique such as variational Bayesian expectation maximization (VBEM) or Markov Chain Monte Carlo (MCMC) (e.g., (Bishop, 2007)).

We now briefly introduce our use of these well-known technologies, as the details will demonstrate the important connection to the backfitting algorithm. Note, that the main benefits of using these algorithms comes from their scalability as they are able to inherit the linear time complexity of the state-space model.

Variational-Bayesian Expectation Maximization

E-Step: We use a variational-Bayesian (VB) approximation to the E-step by making the standard assumption of an approximate posterior that factorizes across the $\mathbf{Z}_i$, i.e.,

$$q(\mathbf{Z}) = \prod_{i=1}^{D} q(\mathbf{Z}_i). \quad (9)$$

Given such a factorized approximation, it can be shown that $\mathrm{KL}(q(\mathbf{Z})||p(\mathbf{Z}|\mathbf{y}, \theta))$ can be minimized in an iterative fashion, using the following central update rule (Bishop, 2007):

$$\log q(\mathbf{Z}_j) = \mathbb{E}_{i\neq j}(\log p(\mathbf{y}, \mathbf{Z}|\theta)) + \text{const.} \quad (10)$$

where $\mathbb{E}_{i\neq j}(\cdot)$ is an expectation with respect to $\prod_{i\neq j} q(\mathbf{Z}_i)$. Using Eqs. (7) and (8), we derive the iterative updates required for VBEM. We first write down the log joint over all variables, given by:

$$\begin{aligned}\log(p(\mathbf{y}, \mathbf{Z}|\theta)) &= \sum_{n=1}^{N} \log p\left(y_n|\mathbf{h}^T \sum_{d=1}^{D} \mathbf{z}_d^{t_d(n)}, \sigma_n^2\right) \\ &+ \sum_{d=1}^{D}\sum_{t=1}^{N} \log p(\mathbf{z}_d^t|\mathbf{z}_d^{t-1}, \theta_d), \quad (11)\end{aligned}$$

where we have defined $p(\mathbf{z}_d^t|\mathbf{z}_d^{t-1}, \theta_d) \equiv p(\mathbf{z}_d^1|\theta_d)$, for $t = 1$, and $\mathbf{h}^T\mathbf{z}$ gives the first element of $\mathbf{z}$. Note that it is also necessary to define the mapping $t_d(\cdot)$ which gives, for each dimension d, the state-space model index associated with y_n. The index t iterates over the sorted input locations along axis d. Because the expectation of the right hand side of Eq. (8) does not depend on $\mathbf{z}_j$, it will only have an effect on the first term of Eq. (**??**), allowing us to write:

$$\begin{aligned}&\log q(\mathbf{Z}_j) \\ &= \sum_{n=1}^{N} \log \mathcal{N}\left(y_n - \mathbf{h}^T \sum_{i\neq j} \mathbb{E}\left[\mathbf{z}_i^{t_d(n)}\right] | \mathbf{h}^T \mathbf{z}_j^{t_j(n)}, \sigma_n^2\right) \\ &+ \sum_{t=1}^{N} \log p(\mathbf{z}_j^t|\mathbf{z}_j^{t-1}, \theta_j) + \text{const}, \quad (12)\end{aligned}$$

where $\mathbb{E}\left[\mathbf{z}_i^k\right] = \int \mathbf{z}_i^k q(\mathbf{Z}_i) d\mathbf{Z}_i$. A key and somewhat surprising outcome of Eq. (9) is that in order to update the factor $q(\mathbf{Z}_j)$ in the E step, it is sufficient to run the standard state-space model inference procedure using only the pseudo-observations: $\left(y_n - \mathbf{h}^T \sum_{i\neq j} \mathbb{E}\left[\mathbf{z}_i^{t_d(n)}\right]\right)$ (Barber et al., 2011). There are a number of conclusions that can be drawn from this connection. First, since VB iterations are guaranteed to converge, any moment computed using the factors $q(\mathbf{Z}_i)$ is also guaranteed to converge. Convergence of these moments is important because they are used to learn the hyperparameters. Second, since the true posterior $p(\mathbf{Z}_1, \ldots, \mathbf{Z}_D|\mathbf{y}, \theta)$ is a large joint Gaussian over

all the latent variables, $\mathbb{E}_q(\mathbf{Z})$ will be equal to the true posterior mean. We will further detail these important conclusions in the Supplementary material (Section A). It is also interesting that the central VBEM update is precisely a backfitting update, thus illustrating a novel connection between approximate Bayesian inference for additive models and classical estimation techniques. Furthermore, this provides an alternative proof of why backfitting computes exact posterior means over latent function values.

M-Step: We must optimize $\mathbb{E}_q\left(\log p(\mathbf{y}, \mathbf{Z}|\theta)\right)$ over θ. Using Eq. (**??**) it is easy to show that the expected sufficient statistics required to compute derivatives with respect to θ are the set of expected sufficient statistics for the state-space model associated with each individual dimension. This separability is another major advantage of using the factorized approximation to the posterior. Thus, for every dimension d, we use the Kalman filter and RTS smoother to compute $\left\{\mathbb{E}_{q(\mathbf{z}_d)}(\mathbf{z}_d^n)\right\}_{n=1}^N$, $\left\{\mathbb{V}_{q(\mathbf{z}_d)}(\mathbf{z}_d^n)\right\}_{n=1}^N$ and $\left\{\mathbb{E}_{q(\mathbf{z}_d)}(\mathbf{z}_d^n\mathbf{z}_d^{n+1})\right\}_{n=1}^{N-1}$. We then use these expected statistics to compute derivatives of the expected complete data log-likelihood with respect to θ and use a standard minimizer (we use a conjugate gradient method) to complete the M step.

Markov Chain Monte Carlo (MCMC)

An important and customary comparison to VB is MCMC, which carries the usual benefits of approximate hyperparameter integration, but at a reduced efficiency. Here we briefly discuss our fairly standard MCMC implementation, noting only the important differences.

As in standard MCMC, we extend the model to include a prior over the hyperparameters. The hyperparameters for each univariate function $\mathbf{f}_d$ are given a prior parameterized by $\{\mu_l, v_l, \alpha_\tau, \beta_\tau\}$, where $\{\mu_l, v_l\}$ correspond to the covariance function hyperparameter ℓ and $\{\alpha_\tau, \beta_\tau\}$ to τ_d. We also place a $\Gamma(\alpha_n, \beta_n)$ prior over the noise precision hyperparameter τ_n. We run Gibbs sampling where we block-sample the latent chains. The algorithm used to sample from the latent Markov chain in a state-space model has been called the forward-filtering, backward sampling algorithm, where forward filtering is followed by a backward sampling from the conditionals $p(\mathbf{z}_k|\mathbf{z}_{k+1}^{sample}; \mathbf{y}; \mathbf{X}_{:,d}; \theta_d)$ (Douc et al., 2011). The sampling is initialized by sampling from $p(z_K|\mathbf{y}; \mathbf{X}_{:,d}; \theta_d)$, which is computed in the final step of the forward filtering run, to produce z_K^{sample}. The forward-filtering, backward sampling algorithm generates a sample of the entire state vector jointly (over training and test input locations).

2.2. Efficient Projected Additive GP Regression

In this section, we will present the main contribution of this paper – projection pursuit Gaussian Process regression (PPGPR).

So far, we have shown how the assumption of additivity can be exploited to derive non-sparse GP regression algorithms which scale as $\mathcal{O}(N)$. These considerable efficiency gains can however decrease accuracy and predictive power versus a full unstructured GP, due to the limited expressivity of the simple additive model. To address this, we now demonstrate a relaxation of the additivity assumption *without* sacrificing the $\mathcal{O}(N)$ scaling, by considering an additive GP regression model in a feature space linearly related to original space of covariates (Snelson & Ghahramani, 2006).

We show that learning and inference for such a model can be performed by using *projection pursuit* GP regression, a novel fusion of the classical projection pursuit regression algorithm with GP regression, with no change to computational complexity. We refer to the following *projected additive* GP prior:

$$y_i = \sum_{m=1}^{M} \mathbf{f}_m(\boldsymbol{\phi}_m(i)) + \epsilon, \tag{13}$$

$$\boldsymbol{\phi}_m = \mathbf{X}\mathbf{w}_m, \tag{14}$$

$$\mathbf{f}_m(\cdot) \sim \mathcal{GP}\left(\mathbf{0}, k_m(\boldsymbol{\phi}_m, \boldsymbol{\phi}'_m; \theta_m)\right), \tag{15}$$

$$\epsilon \sim \mathcal{N}(0, \sigma_n^2).$$

for $i = 1, \ldots, N$, and $m = 1, \ldots, M$. Each of the linear projections $\{\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_M\}$ projects the D dimensional input space to a different scalar input space. Notice that the number of projections, M, can be less or greater than D. Forming linear combinations of the inputs before feeding them into an additive GP model significantly enriches the flexibility of the functions supported by the prior above, including many terms which are formed by taking products of covariates, and thus can capture relationships where the covariates jointly affect the target variable. In fact, Eqs. (10) through to (12) are identical to the standard neural network model where the nonlinear activation functions are modeled using GPs.

Consider the case where $M = 1$. In this case, the resulting projected additive GP regression model reduces to a scalar GP with inputs given by $\mathbf{X}\mathbf{w}_1$. Recall from Section 1.2 that, for a kernel that can be represented as a state-space model, we can use the EM algorithm to optimize θ with respect to the marginal likelihood efficiently, for some fixed $\mathbf{w}_1$. It is possible to extend this idea and jointly optimize $\mathbf{w}_1$ and θ with

respect to the marginal likelihood, although we opt to optimize the marginal likelihood directly. Notice that every step of this optimization scales as $\mathcal{O}(N)$, since at every step we need to compute the marginal likelihood of a scalar GP (and its derivatives). These quantities are computed using the Kalman filter by differentiating the Kalman filtering process with respect to $\mathbf{w}_1$ and θ. This process is described in detail in Supplementary material (Section B).

We now handle the case where $M > 1$ using a greedy approach. At each iteration we find the optimal projection weight $\mathbf{w}_m$. The greedy nature of the algorithm allows the learning of the dimensionality of the feature space, M, rather naturally – one keeps on adding new feature dimensions until there is no significant change in performance (e.g., normalized mean-squared error). One important issue which arises involves the initialization of the projection vector $\mathbf{w}_m$ at step m. In our implementation, as an educated guess, we chose to initialize the weights as those obtained from a linear regression of $\mathbf{X}$ onto the target/residual vector $\mathbf{y}^m$. We call this algorithm, which learns $\{\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_M\}$ and $\{\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \ldots, \boldsymbol{\theta}_M\}$, projection pursuit GP regression (PPGPR). For more examples and information please see the Supplementary material (Section B).

3. Results

In this section we will compare methods for multidimensional regression on both simulated and real experimental data. For each experiment presented, we will compare both runtime and accuracy. If a particular algorithm has a stochastic component to it (e.g., if it involves MCMC) its performance will be averaged over 10 runs. Every experiment was composed of training (i.e., smoothing and hyperparameter learning given $\{\mathbf{X}, \mathbf{y}\}$) and testing phases. In each experiment, we used 1000 points for test sets.

In terms of accuracy, we use two standard performance measures: normalized mean square error (NMSE) and test-set Mean Negative Log Probability (MNLP).

$$\text{NMSE} = \frac{\sum_{i=1}^{N_\star}(\mathbf{y}_\star(i) - \boldsymbol{\mu}_\star(i))^2}{\sum_{i=1}^{N_\star}(\mathbf{y}_\star(i) - \bar{y})^2},$$

$$\text{MNLP} = \frac{1}{2N_\star}\sum_{i=1}^{N_\star}\left[\frac{(\mathbf{y}_\star(i) - \boldsymbol{\mu}_\star(i))^2}{\mathbf{v}_\star(i)} + \log 2\pi\mathbf{v}_\star(i)\right],$$

where $\boldsymbol{\mu}_\star \equiv \mathbb{E}(\mathbf{f}_\star|X, \mathbf{y}, X_\star, \theta)$, $\mathbf{v}_\star \equiv \mathbb{V}(\mathbf{f}_\star|X, \mathbf{y}, X_\star, \theta)$, and $\bar{y}$ is the training-set average target value. These measures have been chosen to be consistent with those commonly used in the sparse GP regression literature. We compare runtime performance in seconds, taking into account both the learning and prediction phases.

We test the following algorithms (with the following names): the full naive GP implementation (Full GP), additive models (Section 2.1) using VBEM inference (Additive-VB) and the MCMC inference (Additive-MCMC), projected additive models using greedy projection pursuit of Section 2.2 (PPGPR-Greedy) and a variation of MCMC (PPGPR-MCMC). Finally, for the sparse GP method we used the sparse pseudo-input Gaussian process (SPGP) (Snelson & Ghahramani, 2006). For SPGP, to be conservative, we did not learn the pseudo inputs (which can potentially greatly increase the algorithm complexity and runtime) but rather used a random subset of the inputs as the active set. For both the SPGP and the Full GP, we used the GPML Matlab Code version 3.1 (Rasmussen & Nickisch, 2010). Also note that, for Additive-VB and PPGPR-greedy we have set the number of outer loop iterations (the number of VBEM iterations for the former, and the number of projections for the latter) to be at maximum 10 for all N. Increasing this number increased the cost with no meaningful change to accuracy, so this is a reasonable choice. All algorithms were run both as a single thread and using a parallel multicore, but since SPGP and Full GP do not offer efficient implementation of the parallel schemes, their results were the same for both cases[2].

3.1. Synthetic Data Experiments

First we used synthetic data generated by the following model:

$$\begin{aligned} y_i &= \sum_{d=1}^{D}\mathbf{f}_d(x_{:,d}) + \epsilon \quad i = 1, \ldots, N, \qquad (16)\\ \mathbf{f}_d(\cdot) &\sim \mathcal{GP}\left(\mathbf{0}, k_d((\mathbf{x}_d, \mathbf{x}'_d); [1, 1])\right) \quad d = 1, \ldots, D,\\ \epsilon &\sim \mathcal{N}(0, 0.01), \end{aligned}$$

where $k_d(\mathbf{x}_D, \mathbf{x}'_d; [1, 1])$ is given by the Matérn(7/2) kernel with unit lengthscale and amplitude. We used $D = 8$ dimensions, and collected 15 runtimes for N ranging from 1000 to 50000.

Figure 1 illustrates the significant computational savings attained by exploiting the structure of the additive kernel. To find the relationship between the number of inputs to the runtime, we calculated a linear slope of the data in log-log scale. As expected, the slope of the Full GP is close to three (2.52) due to its cubic complexity, and all the approximation algo-

[2]When discussing parallel schemes we refer to only the learning stage. As in all GP frameworks, parallelism can always be used for prediction, since we are only interested in the predictive marginals per test point. However, this does not have any noticeable effect on the runtime and is thus unimportant to the comparison.

rithms have runtimes that scale linearly (0.97, 0.62, 1.01, 0.98, 0.97) with the input size. We can also see that parallel processing of the state-space model matrices offers further improvement in scaling. These results serve only as a rough estimate, because the performance can depend on the chosen algorithm parameters, such as: number of outer loop iterations in the Additive-VB, number of projections in PPGPR-greedy, or number of samples in the MCMC methods. This runtime/accuracy consideration should be used when comparing the efficiency of the algorithms.

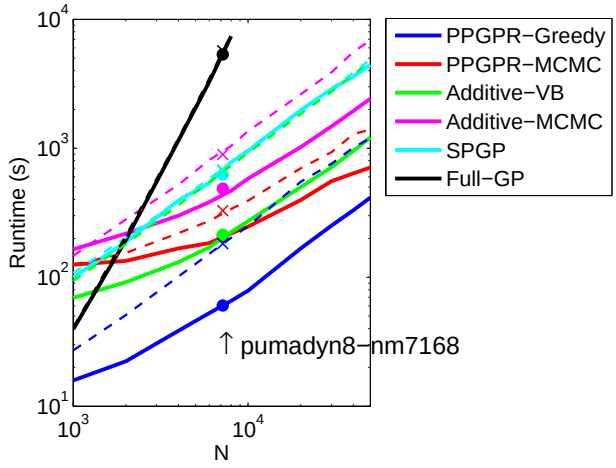

Figure 1. A comparison of runtimes for efficient Bayesian additive GP regression, with $D = 8$, $N = [2; 4; 6; 8; 10; 20; 30; 40; 50] \times 10^3$, presented as a log-log plot. The algorithms ran on a Linux server, once as a single thread (dash lines) and once in a multicore parallel scheme using 8 processors (solid lines). At N=7168, we added an overlay of the runtime results for the pumadyn8-nm dataset (Section 3.2) for both single 'x' and multicore 'o' runs.

Additionally, runtime on a modern computer is by no means a perfect measure of algorithmic complexity. Nonetheless, we will see that the results of Fig. 1 agree with all the results from the real datasets. For example, in Fig. 1 we overlay the results of one of the real datasets, and one sees a close correspondence between synthetic and real data. Thus, these and subsequent results are highly representative and assert the primary point of this section: the runtime of our approximation algorithms do indeed scale linearly with N, versus the cubic scaling of the naive GP.

Fig. 2 shows the effects of increased dimensionality on the approximate algorithms. In this figure we show the runtime speedup of the algorithms with respect to the runtime of the Full GP on the synthetic data generated with dimensionality of either $D = 8$ or $D = 32$. In all the runs the number of inputs was set to $N = 8000$, and the algorithms were run once with a single thread (1 worker = 1W), and once using the parallel scheme (8 workers = 8W). In the multidimensional case, the projection pursuit algorithm exhibits the largest speedup, as it allows for a reduction in the number of effective dimensions (via the greedy selection). Notably, PPGPR-Greedy achieves consistently an order of magnitude improvement over SPGP and VBEM.

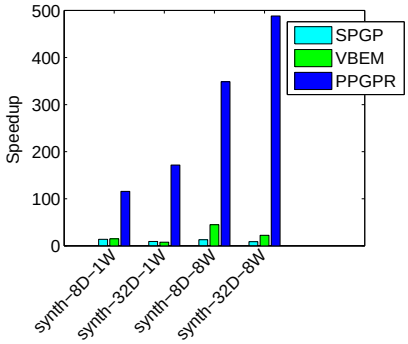

Figure 2. A comparison of the speed up offered by the approximation algorithms compared with exact GP. The runtime was measured on the learning stage for three approximation algorithms: sparse GP, Additive VB, and greedy Projection-Pursuit. The comparison was done using synthetic results with different dimensions (8D and 32D), and running on both a single and multicore (8-core) computer.

3.2. Real Data Experiments

Next, we extend the comparison to real datasets, which will allow thorough accuracy comparisons. We test over seven well-known datasets. These data sets are: `synth-8D` ($N = 8000$ synthetic data from Section 3.1). Next, the `pumadyn` family is a robotic arm dataset, and consists of three datasets: `pumadyn8-fm1000` ($N = 1000$, fairly linear data with $D = 8$ dimensions), `pumadyn8-fm7168` ($N = 7168$, fairly linear data with $D = 8$ dimensions), `pumadyn32-nm` ($N = 7168$, highly nonlinear data with $D = 32$). `Elevators` dataset consists of the current state of the f16 aircraft ($N = 8752$, 17-dimensional) (Alcalá-Fdez et al., 2011), and `kin40k` is a highly nonlinear dataset ($N = 10000$, 8-dimensional)[3]. Fig. 3 demonstrates the central analysis of this section. In each subplot, we calculate speedup, MNLP, and NMSE across all seven datasets and six algorithmic options. To reiterate, we compare our PPGPR algorithm to the two additive methods of Section 2, SPGP (Snelson & Ghahramani, 2006) and a naive full GP implementation. The top subplot in Fig. 3 indicates the substantial speedups offered by all

[3]Pumadyn and Kin40k datasets are from the DELVE archive. Elevators from KEEL archive.

algorithms over the full GP, with the exception only of the $N = 1000$ dataset (`pumadyn8-fm1000`; this is not surprising given small N). Further, as indicated in Figure 1, our PPGPR-Greedy achieves the largest speedup across all datasets, and in most cases the error (MNLP and NMSE) is the same as competing methods. The first four or five datasets tell a very similar accuracy story across PPGPR-Greedy, SPGP, and the full GP. We also see that the simple additive models almost always underperform in accuracy, which is as expected given their limited expressivity compared to PPGPR-Greedy. The one exception where Additive-VB outperforms PPGPR-Greedy is the synthetic data set. However, this is expected as we used an additive model to generate data and the greedy nature of PPGPR-Greedy causes it to underperform. In the final two datasets, we see that SPGP and the full GP have considerably better accuracy. This may be explained as both these datasets are highly nonlinear, making the additive assumption inaccurate.

Understanding the runtime-accuracy tradeoff based on problem requirements is essential. As we just described, PPGPR-greedy achieves the best runtimes but at times with an accuracy cost. Thus we want to quantify the notion of a runtime-accuracy tradeoff. To do so we plot all data sets and algorithms in a runtime vs. error plot (Fig. 4), and we use the economics concept of Pareto efficiency: *efficient* points in the runtime vs error plot represent algorithms with minimum runtime for a given error rate. Pareto inefficient algorithms are then those points that are unambiguously inferior. The efficient frontier is the convex hull of all {runtime,error} points (algorithms) for a given dataset. This will give us a clear picture of which algorithms are optimal choices across a range of datasets. Fig. 4 details this, with one efficient frontier for each dataset (a given color). Each algorithm has a given marker type. This immediately shows what one would expect: the full GP implementation is typically most accurate, but only if one is willing to invest substantial runtime. This choice is often Pareto efficient. Secondly, most often the PPGPR-greedy is the other efficient choice for a substantially reduced runtime, albeit higher error. Surprising to note is the relative weakness of SPGP over several datasets.

Three algorithms stand out in their overall efficiency: PPGPR-Greedy (efficient in all 7 datasets), SPGP (efficient in 4), and full GP (efficient in 6 datasets). Unsurprisingly, the additive model is typically inferior to the more expressive PPGPR model. The PPGPR-Greedy is the *only* consistent efficient algorithm for *all* datasets as it achieves the fastest runtime. However, more interestingly, it also achieves very good accuracy results making most other algorithms inefficient. Of course, any trivial algorithm could achieve efficiency by having minimal runtime and arbitrary error, but the data demonstrates that this is not the case with our algorithms: the PPGPR-greedy error in almost all datasets is competitive or better than all alternatives. Thus the frequent efficiency of PPGPR-greedy is legitimate.

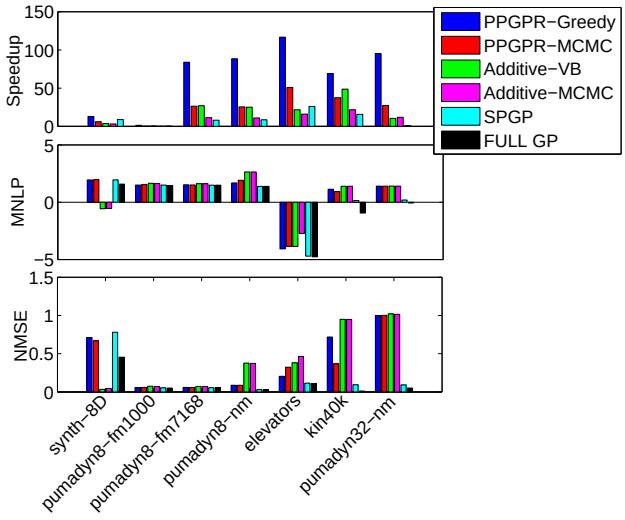

Figure 3. These figures offer a comparison between the different GP methods discussed in the text, taking into account both speedup and accuracy. For comparison we used several known datasets from literature and ran the algorithms on a multicore (8-core) computer. The top figure illustrates the speedup of the approximation algorithms runtimes with respect to the full GP (exact inference) runtime. The bottom two figures show two metrics for calculating regression accuracy.

4. Discussion and Conclusion

Gaussian Processes are perhaps the most popular nonparametric Bayesian method in machine learning, but their adoption across other fields - and notably in application domains - has been limited by their burdensome scaling properties.

While important sparsification work has somewhat addressed this scalability issue, the problem is by no means closed. Our aim here has been to explore the use of projected additive approximations for multidimensional GP models. While PPGPR accuracy was often slightly lower than a full GP, the linear scaling properties of PPGPR mean that it can be efficiently used across a much broader range of data sizes and applications. The primary takeaway of this work is thus: while the naive GP implementation may often

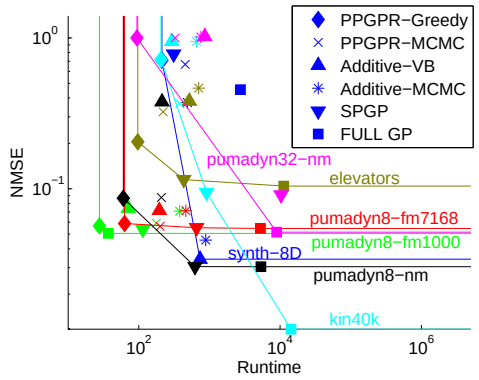

Figure 4. The two fundamental desiderata of our algorithms are accuracy and speed. Here we plot error vs runtime to quantify the tradeoff between these two objectives using the notion of Pareto efficiency. Every algorithm is represented using a unique marker and with a color scheme chosen according to the datasets. For each dataset, the Pareto efficient frontier is shown as a color line passing through the efficient algorithms for that dataset.

produce the highest accuracy, the PPGPR-greedy algorithm that we introduced offers the best runtime-accuracy tradeoff across many datasets and is able to scale well beyond the realm of a naive GP.

Having fast, scalable methods for Gaussian Processes may mean the difference between a theoretically interesting approach and a method that is widely used in practice.

References

Alcalá-Fdez, J., Fernandez, A., Luengo, J., Derrac, J., García, S., Sánchez, L., and Herrera, F. Keel data-mining software tool: Data set repository, integration of algorithms and experimental analysis framework. *Journal of Multiple-Valued Logic and Soft Computing*, 17(2-3), 2011.

Barber, D., Cemgil, A. T., and Chiappa, S. *Bayesian Time Series Models*. Cambridge University Press, 2011.

Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer, 2007.

Douc, R., Garivier, A., Moulines, E., and Olsson, J. On the Forward Filtering Backward Smoothing Particle Approximations of the Smoothing Distribution in General State Space Models. *Annals of Applied Probability*, 2011.

Duvenaud, D.K., Nickisch, H., and Rasmussen, C.E. Additive Gaussian processes. In *NIPS*, pp. 226–234, 2011.

Hartikainen, J. and Särkkä, S. Kalman filtering and smoothing solutions to temporal Gaussian process regression models. In *Machine Learning for Signal Processing (MLSP)*, pp. 379–384, Kittilä, Finland, August 2010. IEEE.

Hastie, T., Tibshirani, R., and Friedman, J. H. *The elements of statistical learning: data mining, inference, and prediction*. New York: Springer-Verlag, Second edition, 2009.

Quiñonero-Candela, J. and Rasmussen, C.E. A unifying view of sparse approximate Gaussian process regression. *Journal of Machine Learning Research*, 6:1939–1959, December 2005.

Rasmussen, C.E. and Nickisch, H. Gaussian processes for machine learning (gpml) toolbox. *Journal of Machine Learning Research*, 11:3011–3015, December 2010.

Rasmussen, C.E. and Williams, C.K.I. *Gaussian Processes for Machine Learning*. The MIT Press, 2006.

Saatci, Y. *Scalable Inference for Structured Gaussian Process Models*. PhD thesis, University of Cambridge, 2011.

Snelson, E. and Ghahramani, Z. Sparse Gaussian processes using pseudo-inputs. In *Advances in Neural Information Processing Systems 18*, pp. 1257–1264, Cambridge, MA, USA, December 2006. The MIT Press.