Projected Sequential Gaussian Processes: Flexible Interpolation for Large Data Sets

Ben Ingram[1], Dan Cornford[1], Remi Barillec[1]

[1]Neural Computing Research Group / Aston University
b.r.ingram, d.cornford, bariller@aston.ac.uk

Abstract. Recently within the machine learning and spatial statistics communities many papers have explored the potential of reduced rank representations of the covariance matrix, often referred to as projected or fixed rank approaches. In such methods the covariance function of the posterior process is represented by a reduced rank approximation which is chosen such that there is minimal information loss. In this paper a sequential framework for inference in such projected processes is presented, where the observations are considered one at a time. We introduce a C++ library for carrying out such projected, sequential estimation which adds several novel features. In particular we have incorporated the ability to use a generic observation operator, or sensor model, to permit data fusion. We can also cope with a range of observation error characteristics, including non-Gaussian observation errors. Inference for the variogram parameters is based on maximum likelihood estimation. We illustrate the projected sequential method in application to synthetic and real data sets. We discuss the software implementation and suggest possible future extensions.

1 Interpolation for large data sets

Techniques for treating large-scale spatial datasets can generally be organised into a small number of categories depending whether they utilise sparsely populated covariance matrices [3], spectral methods or low-rank covariance matrix approximations [5, 4, 1]. Each technique has associated advantages and disadvantages depending on the problem domain being considered. The popularity of these techniques in recent years is broadly due to the increased ability to collect data; whether it be from satellite–based sensors, sampling potentially vast areas across the globe, aerial photography, large monitoring networks or large repositories of data accessible from online sources. In this paper we consider low-rank covariance matrix approximations.

A common technique for treating large-datasets is simply to sub-sample the dataset and use only a smaller number of observations during analysis. This is probably the most naïve low-rank covariance matrix approximation. All but the smaller subset of observations are discarded. Choosing the subset of observations that are to be retained can be complex. One approach to this is described by [4]. The algorithm runs for a number of cycles sequentially inserting and removing observations determined by the magnitude of the reduction of uncertainty in the model.

[2] presents a similar approach. Instead of discarding some observations, it is suggested that the effect of the discarded observations can be projected onto the low-rank covariance matrix in a sequential manner identifying the most informative observations in the process. It is shown how this can be done in such a way so as to minimise loss of information. Related to this is the technique of [5] where, instead of a sequential projection scheme, the entire dataset is projected on to a set of *active points* in a batch framework.

Choosing these *active points* in a batch framework is complex. [1] presents an alternative idea where the Frobenius norm between the full and an approximate covariance matrices is minimised using a closed expression.

The approach of [2] has a number of advantages. Firstly, since it is a sequential algorithm, the complexity of the model, or the rank of the low-rank matrix can be determined during runtime. In scenarios where the complexity of the dataset is unknown *a priori* this is advantageous. Secondly, in a practical setting, it is not uncommon that observations arrive for processing sequential in time. Thirdly, by processing the data in a sequential fashion, non-Gaussian likelihood functions can be used. The projection step mentioned earlier is employed to project from a non-Gaussian posterior distribution to the nearest Gaussian posterior distribution, again minimising the induced error in doing so. There are two ways in which the update coefficients, necessary for the projection, can be calculated: analytic methods requiring the first and second derivatives of the likelihood function with respect to the model parameters and by population Monte Carlo sampling. The inclusion of this population Monte Carlo sampling algorithm is a novel feature of this implementation. The likelihood function for each observation can be easily described without having to calculate complex derivatives. Furthermore, our sampling based method enables observation operators (which map from latent space to observed space) to be included, permitting data fusion.

2 Sequential approaches to inference

To highlight the approach, we give a simple algorithm outline (Algorithm 1) for the computation of the posterior distribution. The basis of this method is a parametrisation of the posterior distribution with mean $\mu(x)$ and covariance $c(x, x')$ where x denotes spatial location. The low-rank posterior distribution is parametrised using a vector $\boldsymbol{\alpha}$ and matrix $\boldsymbol{C}$ as shown by:

$$\mu_{posterior}(x) = \mu_{prior}(x) + \sum_{i}^{m} \alpha_i c(x, x_i), \tag{1}$$

and:

$$c_{posterior}(x, x') = c_{prior}(x, x') + \sum_{i,j=1}^{m} c(x, x_i) C(i, j) c(x_j, x') \tag{2}$$

Additional parameters, $(\boldsymbol{a}, \boldsymbol{\Lambda}, \boldsymbol{P})$, are necessary if the data recycling or expectation propagation stage of the algorithm is utilised but will not be elaborated here. Updating the model parameters $\boldsymbol{\alpha}$ and matrix $\boldsymbol{C}$ is performed sequentially using update coefficients q and r:

$$\mu_{t+1} = \mu_t + q_{t+1} c_t(x, x_{t+1}), \tag{3}$$

and:

$$c_{t+1}(x, x') = c_t(x, x') + r_{t+1} c_t(x, x_{t+1}) c_t(x_{t+1}, x') \tag{4}$$

where t indicates the algorithm iteration step. The benefits of this parametrisation are two-fold. First, the update coefficients (q & r) can be computed for a variety of likelihood models, and secondly, the update coefficients can be computed in a manner which does not require that the model complexity to be increased.

Algorithm 1 Projected Sequential Gaussian Process algorithm outline.

```
1: Initialise model parameters to empty values: $(\boldsymbol{\alpha}, \boldsymbol{C})$ , $(\boldsymbol{a}, \boldsymbol{\Lambda}, \boldsymbol{P})$
2: for epIter = 1 to Data Recycling Iterations do
3:     $t = 1$
4:     Randomly order the location/observation pairs.
5:     while t < numObservations do
6:         Get next location and observation $(\boldsymbol{x}_t, \boldsymbol{y}_t)$
7:         if epIter = 1 then
8:             No removal of observation contribution possible for previous cycle
9:         else
10:            Remove contribution from previous cycle of current observation $(\tilde{\boldsymbol{\alpha}}, \tilde{\boldsymbol{C}})$
11:        end if
12:        Compute model update coefficients $(q, r)$ for specified likelihood function
13:        if Likelihood update is analytic then
14:            Compute update coefficients $(q, r)$ from first and second derivatives of likeli-
               hood with respect to model parameters
15:        else
16:            Use population Monte Carlo sampling update scheme to compute update co-
               efficients $(q, r)$
17:        end if
18:        Calculate $\gamma$, a heuristic measure of the difference between the current GP and
           that from the previous iteration
19:        if $\gamma$ exceeds a predefined threshold $\epsilon$ then
20:            The location of this observation is added to the Active Set
21:            Increase the size of, and update $(\tilde{\boldsymbol{\alpha}}, \tilde{\boldsymbol{C}})$ and EP parameters $(\boldsymbol{a}, \boldsymbol{\Lambda}, \boldsymbol{P})$ using up-
               date coefficients $(q, r)$
22:        else
23:            The model parameters can be updated by a projection step
24:            Project observation effect $(q, r)$ onto $(\tilde{\boldsymbol{\alpha}}, \tilde{\boldsymbol{C}})$ and add observation to EP param-
               eter matrices $(\boldsymbol{a}, \boldsymbol{\Lambda}, \boldsymbol{P})$
25:        end if
26:        if Size of active set > Maximum Desired Size then
27:            Calculate for each active point a heuristic measure of informativeness
28:            Given the least informative active point, delete it from the active set, decrease
               size of, and update $(\tilde{\boldsymbol{\alpha}}, \tilde{\boldsymbol{C}})$ and update EP parameters $(\boldsymbol{a}, \boldsymbol{\Lambda}, \boldsymbol{P})$
29:        end if
30:        $t = t + 1$
31:        $\tilde{\boldsymbol{\alpha}} = \boldsymbol{\alpha}, \tilde{\boldsymbol{C}} = \boldsymbol{C}$
32:    end while
33: end for
```

3 Design of C++ library

The algorithm discussed in this paper is one of a number of Gaussian process algorithms that we intend to develop, hence we emphasise the importance of having a flexible and extensible framework. Code reuse is likewise of great importance. We have chosen C++ as the implementation language as this allows us to produce both fast and portable code. We utilise the Matlab like IT++[1] library which itself uses the highly optimised vector/matrix operations available from BLAS[2] and LAPACK[3].

The base functionality of any Gaussian process algorithm is the need to calculate covariances given different covariance functions. We provide a `CovarianceFunction` interface (abstract class, strictly) which all covariance functions must implement. The interface defines a number of typical operations such as computing the covariance between two vectors of locations or calculating a symmetric covariance function. Additionally, where available, the functions for calculating gradients of functions with respect to their inputs are implemented. We have implemented a basic number of covariance functions such as the common `ExponentialCF`, `GaussianCF` or `WhiteNoiseCF`. Combinations of covariance functions can be used using the `SumCovarianceFunction` class.

A `LikelihoodType` interface is implemented by: an `AnalyticLikelihood` or a `SamplingLikelihood`. These classes can be further extended for a specific likelihood type. `AnalyticLikelihood` requires the first and second derivative information to be calculated for each likelihood model whereas the `SamplingLikelihood` requires the likelihood model to be specified together with an optional observation operator function. The Gaussian process inference algorithm implements two interfaces, `ForwardModel` and `Optimisable` which requires that four functions are implemented for optimisation: log-likelihood, log-likelihood gradient, set model parameters and get model parameters. Four general purpose gradient-based local optimisers are included in the package. These optimisers take an `Optimisable` model as a parameter. A train function can then be called which finds the parameters that minimise the log-likelihood of the model. General options for optimisation can be specified such as number of iterations, parameter tolerance and objective function tolerance.

4 Example applications

To demonstrate the simplicity of using this software, we present a small segment of code which demonstrates calling our code. Algorithm 2 shows the simple outline. Lines 1 & 2 create two objects, one from the `SequentialGP` class which is passed locations (X), observations (y) and a covariance function object (covFunc) which was previously defined. The second class, `SCGModelTrainer`, is for optimising the model parameters. During instantiation, the only parameter that is required is an object derived from the `Optimisable` class, in this case it is the PSGP model. A loop control structure is created which controls how many data recycling steps are desired. The posterior of the Gaussian process is computed during each iteration. The `SequentialGP` member function `computePosterior` is overloaded. Two scenarios are contemplated. Firstly, that there is one likelihood model for the entire dataset. The second scenario is where observations may have differing likelihood models. A `vector` of references to these

[1]`http://itpp.sourceforge.net/`
[2]`http://www.netlib.org/blas`
[3]`http://www.netlib.org/lapack`

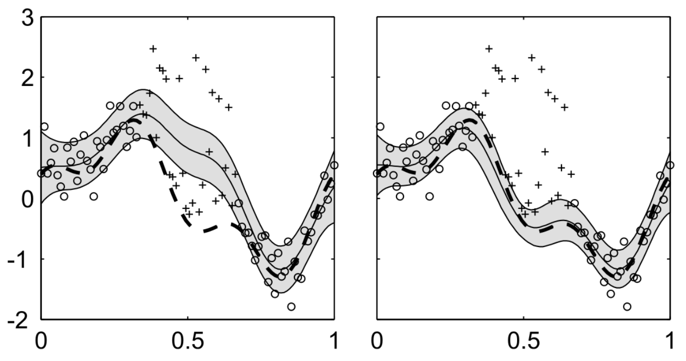

Figure 1: Synthetic example of using the projected sequential Gaussian process algorithm, incorrect (left) and correct (right) likelihood models. Observations with Gaussian additive noise (o) and one-sided exponential noise (+). Dashed line is underlying noiseless function and the black solid line is the mean prediction.

Algorithm 2 Example script.

```
1: SequentialGP ssgp(2, 1, X, y, covFunc); // create psgp object
2: SCGModelTrainer gpTrainer(ssgp); // create optimiser object
3: for epIter = 1 to Data Recycling Iterations do
4:     ssgp.computePosterior(likIndex, likVector); // compute GP posterior
5:     gpTrainer.Train(5); // optimise the model parameters
6: end for
7: ssgp.makePredictions(meanP, varP, pX); // make predictions at pX
```

different likelihood models is the second parameter, the first is an integer `vector` of indexes corresponding to the references in the `vector` of likelihood models. For each observation there is an integer index. This enables a mixture of likelihood models to be used. After the model posterior has been computed, the `train` member function is called with a parameter indicating how many iterations of the optimisation algorithm should be performed.

The software uses a number of default settings which are described in the user manual and documentation. However, the default parameters may not always be appropriate for all applications. Functions for setting and getting particular parameter values are included.

When the error associated with observations are non-Gaussian and/or have varying magnitudes, we can include this knowledge in our model. Figure 1 shows an example of why utilising additional knowledge about a dataset is crucial to obtaining good results during prediction. Additive noise of two types is applied to observations of a simple underlying function. By assuming a Gaussian noise distribution on the centre section of the dataset, prediction values are severely over-estimated. A correct specification of the likelihood for each observation leads to a much improved prediction.

To demonstrate our software in a real-world context, gamma dose radiation data was collected by a large network of sensors administered by Bundesamt für Strahlenschutz[4]. An estimate of the measurement error was derived based on a variety sensor characteristics and was assumed to have the form of a Gaussian distribution. We set the number

[4]`http://www.bfs.de`

of active points to be 150 in this example and leave all other algorithm parameters at their default values. We train our model using 1211 locations for computing the posterior distribution and test it at 700 locations where data was known, but withheld.

Table 1: Summary statistics using 3 different implementations, based on standard Gaussian processes (maximum likelihood based inference), and PSGP in two implementations.

Method	ME	MAE	R	Time(s)
GP	0.0001	0.0091	0.873	349
PSGP (Matlab)	0.0001	0.0094	0.865	1232
PSGP (C++)	0.0001	0.0095	0.865	19

5 Discussion and conclusions

The examples employed to demonstrate the software described are naïve in the sense that a minimal analysis has been reported due to space constraints. Instead, two simple examples have shown and motivated why such an algorithm is important, particularly when implemented in a fast, efficient manner. Timings (Table 1) show that the C++ implementation is a significant improvement in terms of computation time while maintain prediction quality. The tiny discrepancy between PSGP runs is due to the random ordering of the data when applying the data recycling algorithm. The configuration of the algorithm was not discussed in detail as there is insufficient space, however the software provides significant scope for configuration. Fine tuning specific parameters can induce a further computational efficiency increase. For example, specifying the active set *a priori* instead of the default swapping-based iterative refinement is a useful option for reducing computation time. If the threshold for acceptance into the active set, γ, is set too low, then this can cause an increase in computation time due to frequent swapping in and out of active points. Future work will consider multiple outputs (co-kriging), the use of external predictors (universal kriging and regression kriging) in a batch projected Bayesian setting, and focus further on validation of the models.

References

[1] N. Cressie and G. Johannesson. Fixed rank kriging for very large spatial data sets. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 70(1):209–226, 2008.

[2] L. Csató and M. Opper. Sparse online Gaussian processes. *Neural Computation*, 14(3):641–669, 2002.

[3] R. Furrer, M. G. Genton, and D. Nychka. Covariance tapering for interpolation of large spatial datasets. *Journal of Computational and Graphical Statistics*, 15(3):502–523, 2006.

[4] N.D. Lawrence, M. Seeger, and R. Herbrich. Fast sparse Gaussian process methods: The informative vector machine. *Advances in Neural Information Processing Systems*, 15:609–616, 2003.

[5] E. Snelson and Z. Ghahramani. Sparse Gaussian processes using pseudo-inputs. *Advances in Neural Information Processing Systems*, 18:1257, 2006.