Gaussian Process Regression with OCaml[1]
Version 0.9.3

Markus Mottl[2]

April 13, 2013

[1] https://bitbucket.org/mmottl/gpr
[2] markus.mottl@gmail.com

Abstract

This manual documents the implementation and use of the OCaml GPR library for Gaussian Process Regression with OCaml.

Contents

Chapter 1

Overview

The OCaml GPR library features implementations of many of the latest developments in the currently heavily researched machine learning area of Gaussian process regression.

1.1 Background

Gaussian processes define probability distributions over functions. This allows us to apply probabilistic reasoning to problems where we are dealing with *latent functions*, i.e. functions that cannot be directly observed or known with certainty. By specifying prior knowledge about these distributions in form of a *mean* function and a *covariance function*[1] and making use of Bayes' theorem, Gaussian processes provide us with a rigorous nonparametric way of computing posterior distributions over latent functions given data, e.g. to solve regression problems[2]. As more data becomes available, the Gaussian process framework learns an ever more accurate reflection of the probability distribution of latent functions that probably generate the observed data.

Due to their mathematically elegant nature, Gaussian processes allow for analytically tractable calculation of the posterior mean and covariance functions. Though it is easy to formulate the required equations, GPs come at a usually intractably high computational price for large problems. Typically, only problems of up to a few thousand samples can be solved within reasonable time. Efficient approximation methods have been developed in the recent past to address this shortcoming, and this library makes heavy use of them.

Gaussian processes are true generalizations of e.g. linear regression, ARMA processes, single-layer neural networks with an infinite number of hidden units,

[1] Sometimes also called a *kernel*.

[2] Gaussian processes can also be used for classification purposes. This is by itself a large research area, which is not covered by this library.

and many other more widely known approaches and modeling techniques. GPs are closely related to support vector (SVM) and other kernel machines, but have features that may make them a more suitable choice in many situations. For example they offer predictive variances, Bayesian model selection, sampling from the posterior distribution, etc.

It would go beyond the scope of this library documentation to provide for a detailed treatment of Gaussian processes. Hence, readers unfamiliar with this approach may want to consult online resources, of which there are plenty. This section presents an overview of recommended materials.

1.1.1 Video tutorials

Video tutorials are probably best suited for quickly developing an intuition and basic formal background of Gaussian processes and perspectives for their practical use.

- *Gaussian Process Basics*[3]: David MacKay's lecture given at the *Gaussian Processes in Practice Workshop* in 2006. This one hour video tutorial uses numerous graphical examples and animations to aid understanding of the basic principles behind inference techniques based on Gaussian processes.
- *Learning with Gaussian Processes*[4]: a slightly longer, two hour video tutorial series presented by Carl Edward Rasmussen at the Sheffield EPSRC Winter School 2008, which goes into somewhat more detail.
- *Bayesian Inference and Gaussian Processes*[5]: readers interested in a fairly thorough, from the ground up treatment of Bayesian inference techniques using Gaussian processes may want to watch this five hour video tutorial series presented by Carl Edward Rasmussen at the MLSS 2007 in Tübingen.

1.1.2 Books and papers

The following texts are intended for people who need a more formal treatment and theory. This is especially recommended if one wants to be able to implement Gaussian processes and their approximations efficiently, and build up the background knowledge necessary for quick comprehension of publications in the field.

- *Flexible and efficient Gaussian process models for machine learning*[6]: Edward Lloyd Snelson's PhD thesis [Sne08] offers a particularly readable treatment of modern inference and approximation techniques for Gaussian processes that avoids heavy formalism in favor of intuitive notation

[3] `http://videolectures.net/gpip06_mackay_gpb`
[4] `http://videolectures.net/epsrcws08_rasmussen_lgp`
[5] `http://videolectures.net/mlss07_rasmussen_bigp`
[6] `http://www.gatsby.ucl.ac.uk/~snelson/thesis.pdf`

and clearly presented high-level concepts without sacrificing detail needed for implementation. This library owes a lot to his work.

- *Gaussian Processes for Machine Learning*[7]: many researchers in this area would call this book [RW06], which was written by Carl Edward Rasmussen and Christopher K. I. Williams, the "bible of Gaussian processes". It presents a rigorous treatment of the underlying theory for both regression and classification problems, and more general aspects like properties of covariance functions, etc. The authors have kindly made the full text and Matlab sources available online. Their Gaussian process website[8] also lists a great wealth of other resources valuable for both researchers and practitioners.

- *Pattern Recognition and Machine Learning*: this book [Bis07] by Christopher M. Bishop thoroughly introduces the reader to the field of machine learning, especially to Bayesian methods, including Gaussian processes.

- *Matrix Analysis and Applied Linear Algebra*: readers who would like to obtain a deeper understanding of the methods from linear algebra (e.g. matrix factorizations) used in the OCaml GPR library may find a most comprehensive and detailed treatment in this book [Mey00] by Carl D. Meyer.

References to research about specific techniques used in OCaml GPR are provided in the bibliography.

1.2 Features of OCaml GPR

Among other features the OCaml GPR library currently offers:

- Sparse Gaussian processes using the FI(T)C[9] approximations for computationally tractable learning (see [SG05], [Sne08]). Unlike some other approximations that lead to degeneracy, FI(T)C maintains sane posterior variances, at almost no extra computational cost.

- Safe and convenient API for computing posterior means, variances, covariances, log evidence, for sampling from the posterior distribution, calculating statistics of the quality of fit, etc. The OCaml type and module system as used by the API make sure that many easily made programming errors can be avoided and guide the user to make efficient use of the library.

[7] `http://www.gaussianprocess.org/gpml`
[8] `http://www.gaussianprocess.org`
[9] *Fully Independent (Training) Conditional*

- Optimization of hyper parameters via evidence maximization[10] including optimization of inducing inputs (SPGP algorithm[11]). The limited memory BFGS2 algorithm in the GNU scientific library[12] is employed for this purpose.
- Supervised dimensionality reduction and improved predictions under heteroskedastic noise conditions (see [SG06], [Sne08]).
- Sparse multiscale Gaussian process regression (see [WKS08]).
- Variational improvements to the approximate posterior distribution (see [Tit09]).
- Numerically stable GP calculations using QR-factorization to avoid the more commonly used and numerically unstable solution of normal equations via Cholesky factorization (see [FWA+09]).
- Consistent use of bindings to BLAS/LAPACK and C-code throughout key computational parts of the library for optimum performance and conciseness.
- Functors for plugging arbitrary covariance functions into the framework. There is no constraint on the type of covariance functions, i.e. also string inputs, graph inputs, etc., could potentially be used with ease given suitable covariance functions[13].
- Rigorous test suite for checking user-provided derivatives of covariance functions, which are usually quite hard to implement correctly, and self-test code to verify derivatives of marginal log likelihood functions using finite differences.

[10] Also known as type II maximum likelihood.

[11] This library exploits sparse matrix operations to achieve optimum big-O complexity when learning inducing inputs with the SPGP algorithm, but also for multiscales and other hyper parameters that imply sparse derivative matrices for the marginal log likelihood.

[12] `http://www.gnu.org/software/gsl`

[13] The library is currently only distributed with covariance functions that operate on multivariate numerical inputs. Interested readers may feel free to contribute others.

Chapter 2

Using the library

2.1 Interface documentation

The most important file for understanding the API is called `interfaces.ml` and contained in the `lib` directory. It is already heavily documented so we will only provide for a high-level view here. Please refer to the OCaml file for details. Besides defining a few types, e.g. representations for sparse matrices that users will have to use when communicating covariance matrices to the system, the interfaces file contains two important submodules:

- `Specs`, which contains signatures that users need to provide for specifying covariance functions:
  - `Kernel`, the signature for accessing the datastructure that determines a covariance function (= kernel) and its parameters.
  - `Eval`, the signature of modules users have to implement to evaluate covariance functions:
    - `Kernel`, a module satisfying the `Kernel` signature above.
    - `Inducing`, for evaluating covariances among inducing points.
    - `Input`, for evaluating covariances involving single input points and inducing inputs.
    - `Inputs`, for evaluating covariances involving multiple input points[1] and inducing inputs.
  - `Deriv`, the signature of modules users have to implement to compute derivatives of covariance functions:
    - `Eval`, a module satisfying the `Eval` signature above. Derivative code without the ability to evaluate functions would be rather useless.

[1] Required separately besides evaluation of single inputs to force the user to think about how to optimize for this case, which is quite important.

    * `Hyper`, a module specifying the type of hyper parameters, for which derivatives can be computed.
    * `Inducing` and `Input`, which provide a similar abstraction for derivatives as the modules of same name provide for evaluation functions in `Eval`. Note that computations between covariance evaluations and derivatives can be shared. This is especially useful and efficient for covariance functions that use the exponential function.

- `Sigs`, which contains signatures a Gaussian process framework will provide once it has been instantiated with a given covariance specification:
  - `Eval`, which contains modules the user can access to perform computations in the Gaussian process framework:
    * `Spec`, the user-provided specification of the covariance function as mentioned further above.
    * `Inducing`, which contains functions to select and evaluate inducing inputs.
    * `Input`, for dealing with single inputs.
    * `Inputs`, for dealing with multiple inputs.
    * `Model`, for dealing with models. A model is specified by the inputs it consists of and the noise level.
    * `Trained`, for dealing with trained models. A trained model is a model that has been trained on a given target vector.
    * `Stats`, for computing statistics of the trained model (quality of fit, etc.).
    * `Mean_predictor`, minimalist datastructure for making mean predictions.
    * `Mean`, a posterior mean for a single point.
    * `Means`, multiple means.
    * `Co_variance_predictor`, minimalist datastructure for making (co-)variance predictions.
    * `Variance`, a posterior variance for a single point.
    * `Variances`, multiple posterior variances.
    * `Covariances`, posterior covariances.
    * `Sampler`, sampling at a single point.
    * `Cov_sampler`, sampling at multiple points (accounting for their covariance).
  - `Deriv`, which contains modules the user can access to perform derivative computations for marginal log likelihoods within the Gaussian process framework:
    * `Eval`, module satisfying the `Eval` signature mentioned above.
    * `Deriv`, module containing all the derivative code:

- `Spec`, the user specification for covariance function derivatives.
- `Inducing`, `Inputs`, `Model`, `Trained`, basically mirror the role of the modules of same name in the `Eval` signature.
- `Test` contains functions for testing both derivative code supplied by the user and internal code using finite differences.
- `Optim` contains submodules for optimizing Gaussian processes, currently only the `Gsl` submodule, which uses the GNU scientific library for that purpose.

2.2 Predefined covariance functions

The following modules implementing covariance functions already come with the library:

- `Cov_const`: the covariance of a constant function.
- `Cov_lin_one`: the covariance of a linear function with a single hyper parameter (bias).
- `Cov_lin_ard`: the covariance of a linear function with *Automatic Relevance Determination* (ARD).
- `Cov_se_iso`: isotropic squared exponential covariance with amplitude and length scale hyper parameters.
- `Cov_se_fat`: a highly parameterizable ("fat") squared exponential covariance function with amplitude, dimensionality reduction, multiscales, and heteroskedastic noise support.

Chapter 3

Example applications

There are currently three applications that are part of the distribution, two for testing the library, and one simple command-line tool for solving regression problems presented in comma-separated (CSV) files.

3.1 Test applications

The directory `test` contains two applications.

3.1.1 Derivative testing

The application `test_derivatives.native` generates random data to run the internal test suite for checking the correctness of derivatives of the marginal log likelihood function for the "fat" squared exponential covariance function. It prints out the hyper parameters it is currently testing and would fail with an appropriate message if there were a problem.

3.1.2 Test case for learning

The application `save_data.native` evaluates random inputs on some known, nonlinear, one-dimensional function, adds noise, and then trains a Gaussian process to learn the function from the data. The application writes out a number of results that can subsequently be visualized using R[1] and cross-verified with a simple Octave[2] script.

Just run `save_data.native`. It will print out progress information while performing evidence maximization to find suitable hyper parameters and locations for inducing inputs. This will not usually take more than a few seconds

[1] `http://www.r-project.org`
[2] `http://www.gnu.org/software/octave`

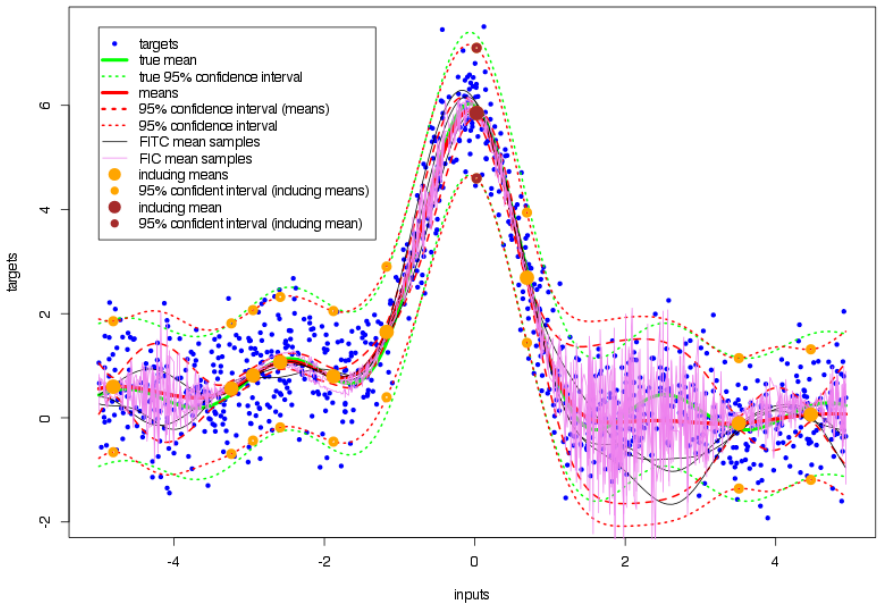
(often just a fraction) unless the randomly chosen initial state leads to a bad local optimum that is surrounded by an almost flat surface. Just restart the run in this unlikely case. Gaussian processes do not seem overly prone to overfitting, but depending on the problem and the chosen covariance function, evidence maximization may become trapped in local optima. These local optima can be interpreted as alternative solutions if they fit about equally well though. Once the application finishes, it will store results in the `test/data` subdirectory.

Visualisation of results

We can now execute `R` on the command-line to run the visualization script for this data by typing:

```
source('display.R')
```

This will visualize the data points, true mean and true confidence intervals, the inferred mean function and its confidence intervals, samples of candidate latent functions from the posterior distribution, locations of inducing inputs, etc. Here is an example:



Verification against Octave implementation

There is also a small test suite for comparing results of the OCaml library to equations written in Octave. It, too, depends on the data saved above by

`save_data.native`. The Octave test suite does not use particularly efficient ways of computing its results, but is fairly simple and readable. It calls Edward Snelson's SPGP implementation[3] for reference. The user may want to compile the more efficient `dist.c` file from within Octave first:

```
mex dist.c
```

Then source the test suite:

```
source "oct.m"
```

3.2 Command-line tool

The application `ocaml_gpr.native` employs the OCaml GPR library for implementing a simple utility to train and evaluate Gaussian process models using the "fat" squared exponential covariance function and the variational improvement [Tit09] for model selection. It reads comma-separated values from standard input for both training and testing. This application is considered to be an example only, but will likely be extended in the future for more serious usage.

Datasets for regression problems that one may want to try out for testing can be downloaded from many sites, one of the most well-known being the UCI Machine Learning Repository[4].

3.2.1 Training models

Here is an example invocation:

```
ocaml_gpr.native -verbose -cmd train -model foo.model < data.csv
```

It is assumed that the file `data.csv` is comma-separated and that the last column contains the target values. The trained model will be stored in file `foo.model`. It is generally recommended to use the `-verbose` flag for training, which will display various statistics at most once a second during training iterations on standard error, e.g.:

```
target variance: 84.41956
iter    1: MSLL=18.9074776 SMSE=0.3878503 MAD=3.8968803 MAXAD=32.1662739
iter    1: |gradient|=29911.88895
iter  171: MSLL=-0.8875856 SMSE=0.2672445 MAD=2.9789019 MAXAD=34.0733409
iter  171: |gradient|=57.08112
```

The user can interrupt training at any time by pressing `CTRL-C` if the result seems good enough. The best model found so far, as determined by the mean standardized log loss (MSLL), will then be saved to the specified model file.

[3] `http://www.gatsby.ucl.ac.uk/~snelson/SPGP_dist.tgz`
[4] `http://archive.ics.uci.edu/ml/index.html`

It is also possible to specify a maximum number of iterations using the flag `-max-iter`. Otherwise the optimizer parameters (see below) determine when learning stops.

Training flags

Various flags can be passed to parameterize the learning process:

- `-n-inducing` sets the number of inducing inputs. The more points are used, the more flexible the function that can be learnt. Note that using as many inducing points as there are inputs will not necessarily yield the full Gaussian process, because the used approximation methods may also model heteroskedastic noise. Furthermore, the computational effort increases as $O(M^3)$, M being the number of inducing inputs. The number of inducing inputs will by itself not lead to overlearning, i.e. more is usually rather better than worse. But increasing this number may lead to a larger number of local optima and hence not necessarily better results.
- `-sigma2` sets the initial noise level hyper parameter.
- `-amplitude` sets the initial amplitude hyper parameter.
- `-dim-red` allows setting the target dimension for dimensionality reduction of the input data. None will be performed otherwise. Note that one can also specify the full dimensionality of the original input data, in which case it will be subject to a fully general linear transformation, which will be learnt in a supervised way in order to reveal useful features.
- `-log-het-sked` turns on support for improved learning of heteroskedastic noise and sets the initial value for the logarithm of the associated hyper parameters. Negative values may often be required to avoid getting trapped in bad optima right at the start.
- `-multiscale` turns on learning of multiscale parameters.
- `-tol`, `-step`, and `-eps` set the line search tolerance, the initial step size, and the stopping criterion (gradient norm) for the GSL-optimizer respectively.

It usually requires some experimentation to find out what kinds of parameters may be most suitable for a given problem.

3.2.2 Applying models

Here is an example on how to apply models to test sets:

```
ocaml_gpr.native -cmd test -model foo.model < test.csv
```

It is assumed that the test set only contains inputs in its columns. The mean predictions for each input will be printed in the same order to standard output.

By specifying `-with-stddev` on the command line, a second column will be printed separated by a comma, which contains the uncertainty of this mean prediction expressed as a standard deviation. If the flag `-predictive` is used, the noise will be included to yield a predictive distribution.

Chapter 4

Future work

Besides improving the usability of the example application, a few extensions to the library are considered for the near future:

- More flexible covariance functions. Besides adding more such functions and more features to e.g. the "fat" squared exponential covariance function and making parameterization simpler, a very interesting approach would be to support combining covariance functions. As described in [RW06], the sum and product of these are also covariance functions, which would hence allow making better use of problem-specific background knowledge.
- Warping (see [SRG03]) for nonlinear, nonparametric transformations of the target variable.
- Sparse convolved GPs (see [AL08]), which would support multiple non-linearly correlated target variables. This would require implementing the PI(T)C[1] approximation (see [Sne08]), which may also be beneficial for solving particular problems.
- Initialisation of inducing inputs using partial Cholesky factorization instead of randomly chosen points. This may be especially useful in the future for problems that have non-numerical inputs, because these cannot be optimized with numerical and even less so with efficient gradient-based methods.
- A global optimisation framework that uses Gaussian processes to model loss functions that are expensive to evaluate would seem like a great application to add.

The interested reader may feel free to contribute these or other features.

[1] *Partially Independent (Training) Conditional*

Appendix A

Implementation details

This section consists of equations used for computing the FI(T)C predictive distribution, and the log likelihood and its derivatives in the OCaml GPR library. The implementation factorizes the computations in this way for several reasons: to minimize computation time and memory usage, and to improve numerical stability by e.g. using QR factorization to avoid normal equations, and by avoiding inverses whenever possible without great loss of efficiency[1]. It otherwise aims for ease of implementation, e.g. combining derivative terms to simplify dealing with sparse matrices.

The presentation and notation here is somewhat similar to [Sne08]. Thus, interested readers are encouraged to first read his work, especially the derivations in the appendix. Our presentation deviates in minor ways, but should hopefully still be fairly easy to compare. The log likelihood derivatives have been heavily restructured though. The mathematical derivation of this restructuring would be extremely tedious, hence only the final result is presented.

Here are a few definitions:

- diag_m is the function that returns the matrix consisting of only the diagonal of a given matrix. diag_v returns the diagonal as a vector.
- $\otimes$ represents element-wise multiplication of vectors. A vector raised to a power means element-wise application of that power.
- Parts in red represent terms used for Michalis K. Titsias' variational improvement to the posterior marginal likelihood (see [Tit09]).
- Parts in blue provide for an alternative, more compact, direct and hence more efficient way of computing some result if the required terms are already available.

[1] Unfortunately, inverses and symmetric rank-k operations are unavoidable in some cases to preserve optimum big-O complexity.

$$
\begin{aligned}
V &= K_{NM} K_M^{-\frac{\top}{2}} \\
\widetilde{K}_N &= V V^\top \\
\Lambda &= \mathrm{diag_m}(K_N - \widetilde{K}_N) \\
\Lambda_{\sigma^2} &= \Lambda + \sigma^2 I \\
\\
\boldsymbol{r} &= \mathrm{diag_v}(\Lambda) \\
\boldsymbol{s} &= \mathrm{diag_v}(\Lambda_{\sigma^2}) \\
\\
\underline{K}_{NM} &= \Lambda_{\sigma^2}^{-\frac{1}{2}} K_{NM} \\
QR &= \begin{pmatrix} \underline{K}_{NM} \\ K_M^{\frac{\top}{2}} \end{pmatrix} \quad \text{(QR-factorization of } \underline{K}_{NM} \text{ stacked on } K_M^{\frac{\top}{2}}\text{)} \\
B &= K_M + \underline{K}_{NM}^\top \underline{K}_{NM} = R^\top Q^\top Q R = R^\top R \\
\widetilde{Q} &= \lfloor Q \rfloor^2{}_N \longrightarrow \underline{K}_{NM} = \widetilde{Q} R \\
S &= \Lambda_{\sigma^2}^{-\frac{1}{2}} \widetilde{Q} R^{-\top} \\
\\
l_1 &= -\tfrac{1}{2}(\log|B| - \log|K_M| + \log|\Lambda_{\sigma^2}| + N \log 2\pi) + -\tfrac{1}{2} \boldsymbol{s}^{-1} \cdot \boldsymbol{r} \\
\\
\underline{\boldsymbol{y}} &= \boldsymbol{s}^{-\frac{1}{2}} \otimes \boldsymbol{y} \\
\boldsymbol{t} &= R^{-1} \widetilde{Q}^\top \underline{\boldsymbol{y}} \\
\boldsymbol{u} &= \underline{\boldsymbol{y}} - \widetilde{Q} \widetilde{Q}^\top \underline{\boldsymbol{y}} \\
\\
l_2 &= -\tfrac{1}{2} \boldsymbol{u} \cdot \underline{\boldsymbol{y}} = -\tfrac{1}{2}(\|\underline{\boldsymbol{y}}\|^2 - \|\widetilde{Q}^\top \underline{\boldsymbol{y}}\|^2) \\
l &= l_1 + l_2 \\
\\
T &= K_M^{-1} - B^{-1} \\
\\
\mu_* &= K_{*M} \boldsymbol{t} \\
\sigma_*^2 &= K_* - K_{*M} T K_{*M}{}^\top + \sigma^2 I
\end{aligned}
$$

[2] Take first N rows.

$$
\begin{aligned}
U &= V K_M^{-\frac{1}{2}} \\
\\
\boldsymbol{v}_1 &= \boldsymbol{s}^{-1} \otimes (\vec{1} + \vec{1} - \boldsymbol{s}^{-1} \otimes \boldsymbol{r} - \mathrm{diag_v}(\widetilde{Q}\widetilde{Q}^\top)) \\
U_1 &= \mathrm{diag_m}(\boldsymbol{v}_1^{\frac{1}{2}}) U \\
W_1 &= T - U_1^\top U_1 \\
X_1 &= S - \mathrm{diag_m}(\boldsymbol{v}_1) U \\
\dot{l}_1 &= -\tfrac{1}{2}(\boldsymbol{v}_1 \cdot \mathrm{diag_v}(\dot{K}_N) - \mathrm{tr}(W_1^\top \dot{K}_M)) - \mathrm{tr}(X_1^\top \dot{K}_{NM}) \\
\\
\boldsymbol{w} &= \boldsymbol{s}^{-\frac{1}{2}} \otimes \boldsymbol{u} \\
\boldsymbol{v}_2 &= \boldsymbol{w} \otimes \boldsymbol{w} \\
U_2 &= \mathrm{diag_m}(\boldsymbol{w}) U \\
W_2 &= \boldsymbol{t}\boldsymbol{t}^\top - U_2^\top U_2 \\
X_2 &= \boldsymbol{w}\boldsymbol{t}^\top - \mathrm{diag_m}(\boldsymbol{v}_2) U \\
\dot{l}_2 &= \tfrac{1}{2}(\boldsymbol{v}_2 \cdot \mathrm{diag_v}(\dot{K}_N) - \mathrm{tr}(W_2^\top \dot{K}_M)) + \mathrm{tr}(X_2^\top \dot{K}_{NM}) \\
\\
\dot{l} &= \dot{l}_1 + \dot{l}_2 \\
\\
\frac{\partial l_1}{\partial \sigma^2} &= -\tfrac{1}{2}(\mathrm{sum}(\boldsymbol{v}_1) - \mathrm{sum}(\boldsymbol{s}^{-1})) \\
\frac{\partial l_2}{\partial \sigma^2} &= \tfrac{1}{2}\mathrm{sum}(\boldsymbol{v}_2) \\
\frac{\partial l}{\partial \sigma^2} &= \frac{\partial l_1}{\partial \sigma^2} + \frac{\partial l_2}{\partial \sigma^2}
\end{aligned}
$$

$$
\begin{aligned}
\boldsymbol{v} &= \boldsymbol{v}_1 - \boldsymbol{v}_2 \\
W &= W_1 - W_2 = T - \boldsymbol{t}\boldsymbol{t}^\top - U_1^\top U_1 + U_2^\top U_2 \\
X &= X_1 - X_2 = S - \boldsymbol{w}\boldsymbol{t}^\top - \mathrm{diag_m}(\boldsymbol{v}) U \\
\dot{l} &= -\tfrac{1}{2}(\boldsymbol{v} \cdot \mathrm{diag_v}(\dot{K}_N) - \mathrm{tr}(W^\top \dot{K}_M)) - \mathrm{tr}(X^\top \dot{K}_{NM}) \\
\\
\frac{\partial l}{\partial \sigma^2} &= -\tfrac{1}{2}(\mathrm{sum}(\boldsymbol{v}) - \mathrm{sum}(\boldsymbol{s}^{-1}))
\end{aligned}
$$

Bibliography

[AL08] Mauricio Alvarez and Neil D. Lawrence. Sparse convolved gaussian processes for multi-output regression. In *NIPS*, pages 57–64, 2008.

[Bis07] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 1 edition, October 2007.

[FWA+09] Leslie Foster, Alex Waagen, Nabeela Aijaz, Michael Hurley, Apolonio Luis, Joel Rinsky, Chandrika Satyavolu, Michael J. Way, Paul Gazis, and Ashok Srivastava. Stable and efficient gaussian process calculations, April 2009.

[Mey00] Carl D. Meyer, editor. *Matrix analysis and applied linear algebra*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[RW06] Carl Edward Rasmussen and Christopher Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006.

[SG05] Edward Snelson and Zoubin Ghahramani. Sparse gaussian processes using pseudo-inputs. In *NIPS*, 2005.

[SG06] Edward Snelson and Zoubin Ghahramani. Variable noise and dimensionality reduction for sparse gaussian processes. In *UAI*. AUAI Press, 2006.

[Sne08] Edward Lloyd Snelson. *Flexible and efficient Gaussian process models for machine learning*. PhD thesis, Gatsby Computational Neuroscience Unit, University College London, February 06 2008.

[SRG03] Edward Snelson, Carl Edward Rasmussen, and Zoubin Ghahramani. Warped gaussian processes. In Sebastian Thrun, Lawrence K. Saul, and Bernhard Schölkopf, editors, *NIPS*. MIT Press, 2003.

[Tit09] Michalis K. Titsias. Variational model selection for sparse gaussian process regression. Technical report, School of Computer Science, University of Manchester, UK, 2009.

[WKS08] Christian Walder, Kwang In Kim, and Bernhard Schölkopf. Sparse multiscale gaussian process regression. In William W. Cohen, Andrew McCallum, and Sam T. Roweis, editors, *Machine Learning, Proceedings of the Twenty-Fifth International Conference (ICML 2008), Helsinki, Finland, June 5-9, 2008*, volume 307 of *ACM International Conference Proceeding Series*, pages 1112–1119. ACM, 2008.