Technical Report

UCAM-CL-TR-804
ISSN 1476-2986

Number 804

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

The HasGP user manual

Sean B. Holden

September 2011

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

HasGP is an experimental library implementing methods for supervised learning using *Gaussian process (GP) inference*, in both the regression and classification settings. It has been developed in the functional language *Haskell* as an investigation into whether the well-known advantages of the functional paradigm can be exploited in the field of machine learning, which traditionally has been dominated by the procedural/object-oriented approach, particularly involving C/C++ and Matlab. HasGP is open-source software released under the GPL3 license. This manual provides a short introduction on how install the library, and how to apply it to supervised learning problems. It also provides some more in-depth information on the implementation of the library, which is aimed at developers. In the latter, we also show how some of the specific functional features of Haskell, in particular the ability to treat functions as first-class objects, and the use of typeclasses and monads, have informed the design of the library. This manual applies to HasGP version 0.1, which is the initial release of the library.

Contents

1 Introduction

This user manual provides a straightforward description of how to use the HasGP library to address supervised learning problems, and explains some of the ideas underlying its implementation to the interested reader in more detail. HasGP implements several techniques for supervised machine learning using *Gaussian process (GP)* inference. It is being developed as an ongoing investigation into how machine learning might benefit from the advantages associated with the functional programming paradigm; a more detailed description of this background can be found in (Holden [1]). HasGP implements methods corresponding approximately to chapters 1 to 5 of (Rasmussen and Williams [2]), as a library in the functional programming language *Haskell*; the user manual assumes some degree of familiarity with the language and its associated infrastructure for handling packages, generating documentation and so on.

2 Installation

HasGP uses several existing libraries which need to be installed before it can be used. The installation sequence is as follows.

2.1 Install LAPACK, BLAS and GSL

An implementation of Gaussian processes for classification and regression inevitably rests on a considerable quantity of numerical linear algebra. In HasGP the required linear algebra, along with optimisation algorithms and certain special functions, are implemented using the hmatrix and hmatrix-special packages [3], which are in turn an interface to the now standard LAPACK [4], BLAS [5] and GSL libraries [6]. Consequently you will need to install these first. The details of how to install the libraries are somewhat dependent on your operating system, although in many cases they are available as pre-compiled libraries, so for example under Mac OS X with MacPorts[1] and Xcode installed all that should be needed is

```
port install gsl-devel +universal
```

and under Fedora Linux

```
yum install lapack-devel.i686 blas-devel.i686 gsl-devel.i686
```

or similar. For systems where such an installation method is not supported refer to the above references and to the respective projects' home pages for detailed instructions.

2.2 Install the Glasgow Haskell Compiler (GHC)

The *Glasgow Haskell Compiler (GHC)* can be obtained from www.haskell.org/ghc/. However in most cases it may well be preferable to obtain it as part of the *Haskell Platform* available at hackage.haskell.org/platform/. Haskell has a mature infrastructure for obtaining and installing libraries using cabal and either method should include this in the installation. Unless you have a specific reason to customize your installation then cabal is likely to be the most straightforward method for installing further libraries.

[1] http://www.macports.org/

2.3 Quick installation

At this point it should be possible to finish the installation with the command

```
cabal install HasGP
```

The `cabal` installer should locate and download `HasGP` and the relevant dependencies from the *Hackage* site at `hackage.haskell.org` after which the installation is complete. If you want to install things separately then proceed as follows.

2.4 Install `hmatrix` and `hmatrix-special`

The `hmatrix` and `hmatrix-special` libraries provide Haskell wrappers for the required parts of `LAPACK`, `BLAS` and `GSL`. The former provides common matrix operations and algorithms, and the latter supplies certain necessary special functions. They can be found at the Hackage site, and further information and the user manual can be found at

```
perception.inf.um.es/hmatrix/
```

However `cabal` installation is the most straightforward method:

```
cabal install hmatrix
```

and similar for `hmatrix-special`, as it will first install any dependencies and as it will also automatically obtain the most recent versions.

2.5 Install `HasGP`

There are two options for using `HasGP` . One is to download the source tarball from the project web site at

```
http://www.cl.cam.ac.uk/~sbh11/HasGP/
```

or from Hackage, then incorporate it into your project and compile as usual. The alternative is again to use

```
cabal install HasGP
```

2.6 Detailed documentation

Haskell is accompanied by the `haddock` system for automatically producing documentation from commented code. If you have used `cabal` to install `HasGP` then this will have been done by default; if not then you can run `haddock` independently or ask for documentation to be generated if installing using another method. A current copy of this part of the documentation is maintained at the project web site.

2.7 Quick test

To make sure all is well, place the following in a file `Main.hs`

```
module Main where

import HasGP.Demos.ClassificationDemo1

main = demo
```

and compile it

```
ghc --make Main
```

If you are working under Mac OS X you might need to use

```
ghc -L/usr/lib --make Main
```

due to a known library-naming issue unrelated to `HasGP` . This will produce an executable which when run should do some learning on a simple data set, printing some results and producing some related `.txt` files.

The code `ClassificationDemo1` that you've just run learns using data generated in the same manner as that used in generating figure 3.6 on page 62 of [2]. `HasGP` includes a short piece of Matlab code

```
HasGP-0.1/src/HasGP/Demos/matlab/ClassificationDemo1.m
```

that turns the `.txt` files into a plot for comparison.

3 Demonstration programs: how to learn using `HasGP`

If you want to quickly run a GP on some data, you can use the demonstration programs as templates; if you are interested in extending `HasGP` or you need more detailed information about how the library is implemented, see the detailed information in the next section in conjunction with the library's `haddock` documentation.

We'll use `ClassificationDemo2.hs` to illustrate the process; its listing is in appendix A and it can be found in the directory

```
HasGP-0.1/src/HasGP/Demos/
```

The three files containing the training and test data required can be found in

```
HasGP-0.1/src/HasGP/Data/Files/
```

and the necessary `Main.hs` file is as above, with `ClassificationDemo2` inserted in place of `ClassificationDemo1`. The problem used is as described in the classification demonstration at `www.gaussianprocess.org`. Figure 1 shows the training data.

3.1 Define a stopping function

We first encounter the `stopEP` function.

```
stopEP :: EPConvergenceTest
stopEP s1 s2 = ((count s2) == 100) ||
               ((eValue s1) > (eValue s2)) ||
               (abs ((eValue s1) - (eValue s2)) < 0.001)
```

The library is implemented in a manner allowing us to define arbitrary stopping conditions when computing an approximation using the Laplace or expectation propagation (EP) methods. Stopping tests are functions taking two values, describing a pair of consecutive states. In this case `stopEP` will stop the approximation process when we reach 100 iterations, when the evidence falls or when the change in evidence is small.

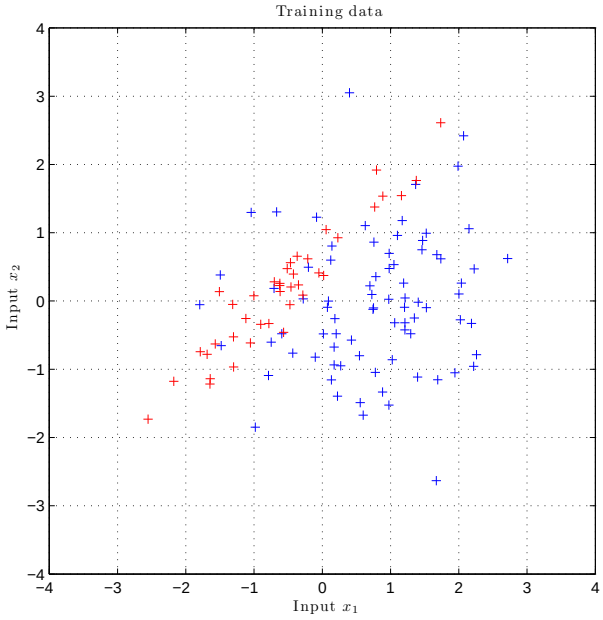

Figure 1: Training examples for the demonstration program `ClassificationDemo2.hs`.

3.2 Define a covariance

After loading the training and test data, the code sets up a covariance function and computes the corresponding covariance matrix:

```
let cov = SquaredExponentialARD (log 1.0) (constant (log 1.0) 2)
let c = covarianceMatrix cov inputs
```

Here, the covariance function is the standard squared exponential function incorporating variance hyperparameters for automatically determining the relevance of the two inputs

$$\mathrm{cov}(\mathbf{x},\mathbf{y}) = c\exp\left(-\frac{1}{2}|\mathbf{x}-\mathbf{y}|^T\mathbf{M}^{-1}|\mathbf{x}-\mathbf{y}|\right)$$

where $\mathbf{M} = \mathrm{diag}(\mathbf{v})^2$, and c and $\mathbf{v}$ are the parameters. All three parameters are given initial values of $\log 1$.

3.3 Construct a classifier

The code then constructs a classifier, to be training using the EP approximation

```
let f = (\v -> gpClassifierEPLogEvidenceVec inputs targets cov
                 generateRandomSiteOrder stopEP v)
```

This function takes as its input `v` the current hyperparameter values for the covariance, and it returns in a pair the resulting values for the evidence and its gradient.

The parameter `generateRandomSiteOrder` requires some further explanation. The EP approximation updates sites incrementally according to some predefined ordering. `HasGP` is implemented in such a way that this ordering can be completely arbitrary, and can be specified when constructing the classifier; `generateRandomSiteOrder` simply specifies that sites should be visited according to a random permutation that changes at each iteration, but the

user is free to define any ordering at all. The required parameter is in fact a *state transformer* and its implementation is via the `State` monad. This is an example of how the abstractions made available by the functional approach can be of benefit in the implementation of such techniques. Details of how this mechanism works can be found in the next section.

3.4 Optimise the hyperparameters

We want to optimise the hyperparameters in the usual way, by maximising the evidence. At present this is achieved using the `hmatrix` interface to a suitable `GSL` minimisation algorithm, and this requires us to convert the function `f` into two functions—one computing the evidence and the other its gradient—in order to call the optimisation code using the required format:

```
let ev = fst . f
let gev = snd . f
let (solution, path) =
      minimizeVD ConjugatePR 0.0001 50 1 0.0001 ev gev
                 (constant (log 1) 3)
```

Here, `minimizeVD` specifies the use of an optimisation method using both vectors and derivatives, `ConjugatePR` specifies the use of the Polak-Ribiere conjugate gradient algorithm, and the four numerical parameters specify tolerance, precision and so on—see the `hmatrix` documentation for details.

3.5 Learn and predict

Having optimised the hyperparameters we set up the covariance to use the optimum values:

```
let cov' = SquaredExponentialARD (solution @> 0)
                      (fromList [(solution @> 1), (solution @> 2)])
let c' = covarianceMatrix cov' inputs
```

and learn and classify accordingly:

```
let (epValue, epState) =
       gpClassifierEPLearn c' targets generateRandomSiteOrder stopEP
let classify =
      gpClassifierEPPredict (state epValue) inputs targets c' cov'
let newOuts = classify points
```

Figure 2 shows the output produced by `ClassificationDemo2.hs`.

4 Advanced use

The following material is aimed at those wishing to understand how specifically functional features of the Haskell language have been employed in developing the library, or who want to develop the library further.

4.1 What's in the folders and modules?

The source folders and modules for `HasGP` are structured as follows.

- The folder `Types` contains a short module `MainTypes.hs` employed purely to make the remainder of the code more readable.

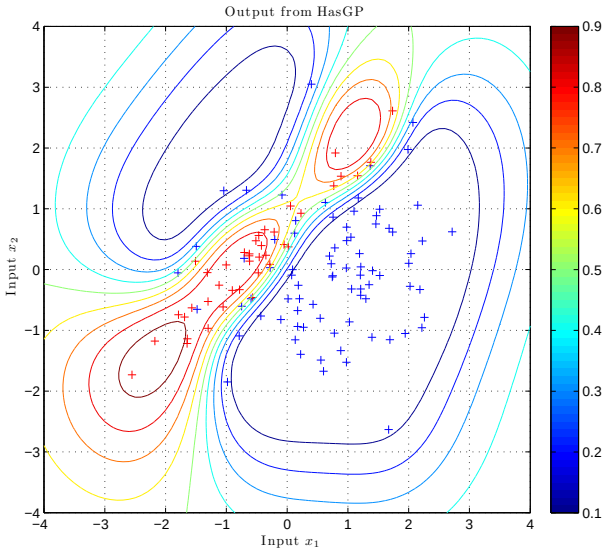

Figure 2: Result produced by the demonstration program `ClassificationDemo2.hs`.

- The folder `Support` contains six modules:

  1. `Functions` implements some simple functions that appear regularly in machine learning applications, such as the sigmoid and delta functions. More importantly, it provides implementations of several functions for which numerical problems quickly produce overwhelming numerical inaccuracy if they are not implemented correctly.
  2. `Iterate` As Haskell is a pure functional language, programs have no state. In implementing GPs we often need to maintain state within an iterative algorithm. The solution is to use the `State` monad to implement state in a functional manner. This module implements a selection of functions supporting this abstraction.
  3. `Linear` implements some simple functions on vectors and matrices. It also implements some specific functions where efficiency can be improved by avoiding the straightforward approach; for example, if we need only the diagonal of a product of two matrices.
  4. `MatrixFunctions` implements some functions performing GP-specific computations on matrices.
  5. `Random` implements functions for producing commonly-required random matrices and vectors.
  6. `Solve` implements functions related to the solution of specifically structured sets of linear equations.

- The folder `Parsers` contains a single module `SvmLight` that can be used to read files of data structured in the commonly used format described at `svmlight.joachims.org`.

- The folder `Likelihood` contains the module `Basic` defining the `LogLikelihood` typeclass, and two further modules implementing the logistic and probit likelihoods.

- The folder `Covariance` contains the module `Basic` defining the `CovarianceFunction` typeclass and some functions for dealing with covariances, such as generating a co-

variance matrix. It also contains two further modules implementing the squared exponential covariance, and its extension incorporating hyperparameters for automatic relevance determination.

- The folder `Data` contains modules for generating common, simple data sets, and a further folder containing training and test data for one of the demonstration programs. It also contains the module `Normalise` implementing common functions for normalising data.
- The folder `Demos` contains three modules providing examples of how to use `HasGP` in practice.
- The folders `Classification` and `Regression` contain the algorithms for performing inference. The former is further divided into two subfolders containing the algorithms for the Laplace and Expectation Propagation (EP) approximations.

4.2 Likelihoods

Likelihood functions are implemented as instances of the `LogLikelihood` typeclass.

```
class LogLikelihood b where
    likelihood     :: b -> Double -> Double -> Double
    dLikelihood    :: b -> Double -> Double -> Double
    ddLikelihood   :: b -> Double -> Double -> Double
    dddLikelihood  :: b -> Double -> Double -> Double
```

where the `likelihood` method computes the value of the log likelihood itself and the remaining methods compute the first three derivatives with respect to the latent variable. So for example, in the case of the included `LogLogistic` likelihood we have

```
data LogLogistic = LogLogistic

instance LogLikelihood LogLogistic where
    likelihood     LogLogistic y f = log (1 / (1 + (exp (-(f * y)))))
    dLikelihood    LogLogistic y f = ...
```

and to compute the first derivative we would use the function

```
dlikelihood LogLogistic
```

having type

```
Double -> Double -> Double.
```

At present the likelihoods implemented are `LogLogistic` and `LogPhi`, which implements the probit likelihood. In order to add further likelihoods it is only necessary to implement them as an `instance` of `LogLikelihood`.

4.3 Covariance

Covariance functions are implemented as instances of the `CovarianceFunction` typeclass.

```
class CovarianceFunction a where
    trueHyper :: a -> DVector
    covariance :: a -> DVector -> DVector -> Double
    dCovarianceDParameters :: a -> DVector -> DVector -> DVector
    makeCovarianceFromList :: a -> [Double] -> a
    makeListFromCovariance :: a -> [Double]
```

Parameters are stored as logs. The `trueHyper` method returns the actual parameter values, the `covariance` method computes the covariance, the `dCovarianceDParameters` method computes the first derivative of a covariance with respect to its parameters, and the final two methods convert between lists of log parameters and `CovarianceFunctions`.

For example, the included `SquaredExponential` covariance function is defined as

$$\mathrm{cov}(\mathbf{x},\mathbf{y}) = c\exp\left(-\frac{1}{2\sigma^2}|\mathbf{x}-\mathbf{y}|^2\right)$$

and implemented as

```
data SquaredExponential = SquaredExponential
    {
      f :: Double,          -- ^ log \sigma_f^2
      l :: Double           -- ^ log l
    }

instance CovarianceFunction SquaredExponential where
    trueHyper se = ...
    covariance se in1 in2 =
        f2 * exp (-(1/(2 * (l2^2))) * (diff <.> diff))
            where
              diff = in1 - in2
              f2 = exp (f se)
              l2 = exp (l se)
    dCovarianceDParameters se in1 in2 = ...
```

and so the function

```
covariance (SquaredExponential (log a) (log b))
```

having type

```
DVector -> DVector -> Double
```

takes two vectors and computes their covariance using the squared exponential function with parameters `a` and `b`.

At present the covariance functions implemented are `SquaredExponential` and its more general form `SquaredExponentialARD` allowing automatic relevance determination and defined as

$$\mathrm{cov}(\mathbf{x},\mathbf{y}) = c\exp\left(-\frac{1}{2}(\mathbf{x}-\mathbf{y})^T\mathbf{M}^{-1}(\mathbf{x}-\mathbf{y})\right)$$

where $\mathbf{M} = \mathrm{diag}(\mathbf{v})^2$, and c and $\mathbf{v}$ are the parameters.

In order to add further covariance functions it is only necessary to implement them as an `instance` of `CovarianceFunction`.

4.4 Monads for state

Both the Laplace and EP approximations for GP classification involve iterative algorithms for which stopping conditions need to be specified. In addition the EP version involves updating sites in a specified order, often either fixed or random. The `State` monad allows these aspects of the algorithms to be implemented in a particularly general, straightforward and extendable manner in Haskell. In the following it is assumed the reader is familiar with the use of monads in functional programming; details can be found in, for example O'Sullivan *et al.* [7].

4.4.1 Ordering of EP sites

When computing the EP approximation, the state of the computation includes EP-specific information, a random number generator and the number of iterations. A `SiteOrder` is a state transformer that produces a site ordering as a list of integers while updating the relevant part of the state, namely the random number generator.

```
type EPState = (EPSiteState, StdGen, Int)

type SiteOrder = State EPState [Int]
```

So for a fixed site ordering we have

```
generateFixedSiteOrder :: SiteOrder
generateFixedSiteOrder = do
  (state, g, n) <- get
  return [1..(dim $ tauTilde state)]
```

whereas for a random ordering we have

```
generateRandomSiteOrder :: SiteOrder
generateRandomSiteOrder = do
  (state, g, n) <- get
  let (newG, p) = randomPermutation g (dim $ tauTilde state)
  put (state, newG, n)
  return p
```

The function `doOneUpdate` takes such a state transformer as one of its parameters.

4.4.2 Arbitrary stopping conditions

We outline the mechanism for the case of the EP approximation; for details of the equivalent approach for the Laplace approximation see the `haddock` documentation.

Information regarding the current state of the EP approximation is encapsulated by the type `EPValue`, and general stopping tests have type `EPConvergenceTest`

```
type EPConvergenceTest = (EPValue -> EPValue -> Bool)

data EPValue = EPValue {
      eValue::Double,
      siteState::EPSiteState,
      count::Int
    }
```

Stopping tests take two values, describing a pair of consecutive states, and produce a boolean. The type `EPSiteState` encapsulates several more detailed parameters used in the approximation, and can be used to compute when to stop if desired. A stopping test is used when constructing an iteration as follows.

The function for learning using the EP approximation is

```
gpClassifierEPLearn :: CovarianceMatrix
                    -> Targets
                    -> SiteOrder
                    -> EPConvergenceTest
                    -> (EPValue, EPState)
gpClassifierEPLearn k t siteOrder converged =
    runState (iterateToConvergence'' doOnce converged) start
```

```
        where
          doOnce = doOneUpdate k t siteOrder
          start = ((generateInitialSiteState k (dim t)), mkStdGen 0, 0)
```

Here, `k` is the covariance matrix and `t` is the targets for the training data. The important section of this code for the purposes of the current discussion is

```
iterateToConvergence'' doOnce converged
```

as `converged` is an arbitrary convergence test as described. The function `iterateToConvergence''` is implemented as a small modification to the function `iterateToConvergence`

```
iterateToConvergence doOnce converged currentValue =
    do newValue <- doOnce
       if (converged currentValue newValue)
         then return newValue
         else iterateToConvergence doOnce converged newValue
```

where the use of the arbitrary convergence test should be clear. The critical issue here however is that `iterateToConvergence` has type

```
iterateToConvergence::State s a -> (a -> a -> Bool) -> a -> State s a
```

Consequently, we can apply it to any descriptions of state and data that are appropriate.

A Listing of `ClassificationDemo2.hs`

```
-- | This function defines when iteration stops.
stopEP::EPConvergenceTest
stopEP s1 s2 = ((count s2) == 100) ||
               ((eValue s1) > (eValue s2)) ||
               (abs ((eValue s1) - (eValue s2)) < 0.001)

demo = do
  putStrLn "Loading the training data..."

  inputs <- loadMatrix "gpml-classifier-x.txt"
  targets <- fscanfVector "gpml-classifier-y.txt" 120
  points <- loadMatrix "gpml-classifier-test.txt"

  putStrLn "Learning and predicting: EP + hyperparameter optimization..."

  let cov = SquaredExponentialARD (log 1.0) (constant (log 1.0) 2)
  let c = covarianceMatrix cov inputs

  let f = (\v -> gpClassifierEPLogEvidenceVec inputs targets cov
                 generateRandomSiteOrder stopEP v)
  let ev = fst . f
  let gev = snd . f
  let (solution, path) =
       minimizeVD ConjugatePR 0.0001 50 1 0.0001 ev gev
                  (constant (log 1) 3)

  putStrLn $ "Solution: " ++ (show $ mapVector exp solution)
  putStrLn $ "Path: "
  putStrLn $ show path

  let cov' = SquaredExponentialARD (solution @> 0)
                                   (fromList [(solution @> 1), (solution @> 2)])
  let c' = covarianceMatrix cov' inputs
  let (epValue, epState) =
        gpClassifierEPLearn c' targets generateRandomSiteOrder stopEP
  let classify =
         gpClassifierEPPredict (siteState epValue) inputs targets c' cov'
  let newOuts = classify points

  fprintfVector "gpml-hasgp-outputs.txt" "%g" newOuts

  putStrLn $ "Done"


  return ()
```

References

[1] Sean B. Holden. HasGP: a Haskell library for gaussian process inference. *Journal of Machine Learning Research*, 2011. Submitted.

[2] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, 2006.

[3] Alberto Ruiz. Introduction to *hmatrix*, 2011. Available at `perception.inf.um.es/hmatrix/`.

[4] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongara, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1999.

[5] J. Dongara, J. Du Croz, I. S. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16:1–17, 1990.

[6] Mark Galassi, James Theiler, Jim Davies, Brian Gough, Gerard Jungman, Patrick Alken, Michael Booth, and Fabrice Rossi. *GNU Scientific Library Reference Manual*. Network Theory Ltd, third edition, January 2009.

[7] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 2009.