Technical Report

UCAM-CL-TR-885
ISSN 1476-2986

Number 885

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

HasGP: A Haskell library for Gaussian process inference

Sean B. Holden

April 2016

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

HasGP: A Haskell library for Gaussian process inference

Sean B Holden
sbh11@cl.cam.ac.uk

Abstract

HasGP is a library providing supervised learning algorithms for *Gaussian process (GP)* regression and classification. While only one of many GP libraries available, it differs in that it represents an ongoing exploration of how machine learning research and deployment might benefit by moving away from the imperative/object-oriented style of implementation and instead employing the *functional programming (FP)* paradigm. HasGP is implemented in *Haskell* and is available under the GPL3 open source license.

1 Introduction

It is commonplace for machine learning libraries to be implemented in an imperative/object-oriented style. In particular, Matlab has become the standard for research use and C/C++ is common for commercial implementations. Other languages are of course also employed, but the imperative/object-oriented paradigm dominates.

The HasGP library is an attempt to explore the use of FP in machine learning by implementing a comprehensive collection of algorithms for supervised learning based on Gaussian processes [1], making use of the FP style to the full. In making HasGP available it is not our aim to extend or to improve upon the GP functionality that is already available using open source software; a large quantity of such material exists providing at least as much functionality as HasGP[1]. Rather, the aim is to provide a starting point for the investigation of how functional programming techniques might benefit us in the implementation of large machine learning systems, and over time to investigate the extent to which we might close any gap in performance.

2 Functional programming: pros and cons

Functional programming can historically be identified with two contradictory ongoing developments:

1. An increasing body of evidence that its informed use can lead to software that is more quickly developed, and of high quality by many of the standard metrics. For example Wiger [2] addresses the use of FP by Ericsson, and a compelling description of how the use of FP has been beneficial in the finance industry can be found in the work of Frankau et al. [3].

2. A relatively small take-up within the wider academic and industrial communities, although FP is by no means of only academic interest. For example, the annual *Haskell Communities and Activities Report* [4] provides numerous examples of its use.

An extensive discussion of the positive and negative points of FP is provided by O'Sullivan et al. [5]. On the positive side, Hughes [6] gives an elegant argument for why the FP style tends to lead to compact, maintainable, clear and correct code. The main point made is that FP provides exceptionally flexible ways

[1] A list of libraries can be found at http://www.gaussianprocess.org/.

of modularizing code and joining modules together through the employment of *higher-order functions*, *partial application* and *lazy evaluation*.

The primary disadvantage typically associated with FP languages is that of speed. Early compilers tended to produce code severely lacking in this respect. However while speed comparisons between programming languages are notoriously difficult to perform in a meaningful way, recent attempts at making reliable comparisons[2] suggest that the *Glasgow Haskell Compiler (GHC)* in particular is highly competitive even against C/C++, and very often produces faster code than dynamically-typed interpreted languages. Perhaps more important in the long term given the growing availability and deployment of cheap multicore processors is that functional languages often provide the developer with a degree of *automated* exploitation of parallelism (see O'Sullivan et al. [5], page 169).

Given that the speed gap is closing the question naturally arises of whether the benefits of FP for software development can be harnessed in the field of machine learning without an insurmountable speed penalty being incurred.

3 Related work

Very little attempt has been made to date systematically to apply FP to machine learning. Given the popularity of *Lisp* among artificial intelligence researchers there have undoubtedly been significant implementations in this language; however, the prevaiing situation is that the use of Lisp is rare in machine learning. To date the only alternative library, of which we are aware, addressing GPs within a modern functional context is the *GPR* library implemented in *OCaml*[3]. This library implements sparse regression, aiming to obtain efficient approximate solutions; it does not yet implement GP classification.

Allison [7] attempts to use the Haskell type system to provide a basic framework for representing classifiers, data and other commonly used concepts. However this work is limited by the fact that it rests heavily on the assumption that learning methods will be based on the minimum description length formulation (see Mitchell [8]).

There is increasing interest in *probabilistic programming languages* (Goodman et al. [9], Pfeffer [10] and Kollmansberger [11]). These are clearly relevant to machine learning, however they are intended for application in a much wider context than the present work, and are not at present in widespread application.

4 The `HasGP` library

`HasGP` implements the regression techniques and the two-class classification techniques corresponding to chapters 1 to 5 of the text of Rasmussen and Williams [1]. It includes exact GP regression, the Laplace and expectation propagation (EP) approximations for classification, and allows optimization of hyperparameters to be performed. At present it makes extensive use of the open source `hmatrix`[4] library to perform the underlying linear algebra. `hmatrix` itself uses bindings to the established and reliable BLAS, LAPACK and GSL libraries (Anderson et al. [12] and Galasssi et al. [13]).

The library includes covariance and likelihood functions, a parser for reading files in the SVMLite format, commonly used data normalization methods, and demonstration code for regression and for both approximations for classification.

The web site for the `HasGP` project[5] contains the current source for the library and in-depth documentation on the code itself. It also includes a user manual providing an explanation of how to install

[2] An extensive language comparison can be found at `http://shootout.alioth.debian.org/`.

[3] This library is available at `http://www.ocaml.info/software.html#machine_learning`.

[4] The `hmatrix` library is available at `http://hackage.haskell.org/`.

[5] The project's web site is at `http://www.cl.cam.ac.uk/~sbh11/HasGP/`

and apply the library, and giving an introduction into how functional techniques have been used in the implementation.

`HasGP` is implemented in Haskell—a pure, lazy functional language that also supports the simulation of impure computations through he use of monads, and allows strict evaluation to be employed where desired.

5 Lessons learned

In developing the initial release of `HasGP` some preliminary findings have quickly become apparent:

1. The claims made in promoting functional programming, in terms of improvements to productivity and to the quality of code produced appear entirely justified. Most errors during development were caught immediately by the type system, with those not caught in this way usually being numerical problems unrelated to choice of programming paradigm.

2. The type system is extremely effective in allowing us to write code that is easy to extend. For example, adding a further likelihood or covariance function requires us only to write a new `instance` of the `LogLikelihood` or `CovarianceFunction` typeclass respectively; the new function can then immediately be used with no further changes required.

3. Similarly, the ability to simulate state in a computation without losing the benefits of referential transparency, by employing the `State` monad, allows us very easily to add extensions. For example the library allows the specification of completely arbitrary stopping functions for iterative processes. It allows the specification of arbitrary orderings, both deterministic and random, for visiting sites in the EP computation. In either case all that is required is to write a function and supply it as a parameter to the relevant learning algorithm.

4. The compromise expected in terms of speed is certainly present. However, it should be noted that to date the library has undergone only limited profiling. What profiling has been performed has resulted in very considerable speed increase. For example, a well-known difficulty in pure functional languages is that changing a single element in a data structure can involve copying the entire data structure. Here again it has been straightforward to employ monadic programming allowing us safely to implement in-place updates to vectors.

6 Ongoing development

We aim to continue the development of the `HasGP` library in at least the following respects:

1. Further profiling of the code to investigate the extent to which the gap in speed might be closed, and investigation of the use of automated parallelism.

2. Further work on refining the use of the Haskell type system in the implementation, building on the work of Allison [7], and further investigation of the use of monadic programming in supporting extension of the library, in particular the probability monad (see Erwig and Kollmansberger [11]).

3. Investigation of numerical techniques in a functional context. For example, Eaton [14] has proposed using the type system to check consistency of matrix dimensions at compile time, and it would also be of interest to assess the effectiveness of FP for tasks such as optimization.

References

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

[2] Ulf Wiger. Four-fold increase in productivity and quality: Industrial strength programming in telecom-class products. In *Third Workshop on Formal Design of Safety Critical Embedded Systems*, Munich, March 2001.

[3] Simon Frankau, Diomidis Spinellis, Nick Nassuphis, and Christoph Burgard. Commercial uses: Going functional on exotic trades. *Journal of Functional Programming*, 19(1):27–45, January 2009.

[4] Janis Voigtlnder. Haskell communities and activities report, 2011. Available electronically at `http://www.haskell.org/haskellwiki/`.

[5] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, 2008.

[6] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[7] Lloyd Allison. Models for machine learning and data mining in functional programming. *Journal of Functional Programming*, 15(1):15–32, January 2005.

[8] Tom Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[9] Noah Goodman, Vikash Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua Tenenbaum. Church: a language for generative models. In *Uncertainty in Artificial Intelligence (UAI)*, pages 220–229, 2008.

[10] Avi Pfeffer. The design and implementation of IBAL: A general-purpose probabilistic language. In Lise Getoor and Ben Taskar, editors, *Introduction to Statistical Relational Learning*. The MIT Press, November 2007.

[11] Martin Erwig and Steve Kollmansberger. Functional pearls: Probabilistic functional programming in Haskell. *Journal of Functional Programming*, 16(1):21–34, 2005.

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

[13] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, and F. Rossi. *GNU Scientific Library Reference Manual*. Network Theory Ltd, 3rd edition, 2009.

[14] Frederik Eaton. Statically typed linear algebra in Haskell. In *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 120–121. Association for Computing Machinery, 2006.