Stan Overview: Language and Inference

Stan Development Team
http://mc-stan.org/

Abstract

Stan is a tool for specifying statistical models in terms of log densities, typically Bayesian posteriors specified in terms of priors and likelihoods. Automatic differentiation of the density function enables state-of-the-art gradient-based inference algorithms: full Bayesian inference with Hamiltonian Monte Carlo, approximate Bayesian inference with gradient-descent variational inference, and penalized maximum likelihood with quasi-Newton optimization. Stan is open source, with interfaces available for R, Python, Julia, MATLAB, Stata, and the shell.

1 Introduction

Stan [1] is intended first and foremost to allow applied statisticians (scientists, as well as more general data scientists, analysts, machine learning researchers) to fit realistic models and perform inferences for quantities of interest, including predictions and event probabilities.

Implementing Bayesian inference proceeds in two steps: first we construct a posterior distribution, which represents uncertainty about the parameters in the model given the data, then we perform a simulation or some approximation to the relevant integrals which represent averages over the posterior [2]. Speed and scalability are critical for scientific exploration and analysis, so we need simulation methods that remain efficient while scaling the size of the data and the size of the model. In particular, the most efficient algorithms use gradients—multivariate derivatives of the logarithm of the probability density—and even more advanced algorithms will require second- or higher-order derivatives to account for the curvature found in the posteriors of complex models.

Stan separates the task of designing and evaluating a probability model, the core activity of applied statistics, from the task of writing estimation and inference algorithms, the purview of computational statisticians and machine learning researchers. Probability models are specified by end users in Stan's imperative, domain-specific language, the variables of which represent observed and unobserved random variables and constants. Estimation and inference are carried out automatically by Stan, given data. Data manipulation, results summarization, and plotting are handled through popular interface languages for statistics, as well as from the shell for ease of cluster and cloud distribution. This makes it easy for users to integrate Stan into their data analysis and modeling workflows.

From a software perspective, the core of Stan consists of three modular layers, a math library, the language compiler, and a suite of statistical algorithms. These are all written in optimized, portable, standards-compliant C++ (C++03, for portability). User-friendly interfaces are available for R, Python, MATLAB, Julia, Stata, and the command-line. Stan is an open-source project hosted on GitHub (`https://github.com/stan-dev`); the core is BSD licensed.

2 Example Program

Figure 1 defines a linear regression in Stan, specifically the joint density

$$p(y, \beta, \sigma \mid x) = \mathsf{Lognormal}(\sigma \mid 0, 1) \times \prod_{k=1}^{K} \mathsf{Normal}(\beta_k \mid 0, 2.5) \times \prod_{n=1}^{N} \mathsf{Normal}(y_n \mid x_n \beta, \sigma).$$

```
data {
  int<lower=3> K;
  int<lower=0> N;
  matrix[N, K] x;
  vector[N] y;
  int<lower=0> N_tilde;
  matrix[M, K] x_tilde;
}
parameters {
  vector[K] beta;
  real<lower=0> sigma;
}
model {
  beta ~ normal(0, 2.5);
  sigma ~ lognormal(0, 1);
  y ~ normal(x * beta, sigma);
}
generated quantities {
  vector[N_p] y_tilde;
  int<lower=0, upper=1> Pr_gt;
  for (n in 1:N_tilda)
    y_tilde[n] <- normal_rng(x_tilde[n] * beta, sigma);
  Pr_gt <- (beta[2] > beta[3]);
}
```

Figure 1: Stan program for simple linear regression with example of predictive quantities in the generated quantities block. The model fits the regression parameters β and noise scale σ and then uses them to generate (a) posterior predictive quantities $\tilde{y}$ according to $p(\tilde{y} \mid \tilde{x}, x, y)$ and (b) posterior event probability $\Pr[\beta_2 > \beta_3 \mid x, y]$.

The model uses the generated quantities block to compute and randomly generate posterior predictive predictive quantities based on the values of the parameters and data. The posterior for new outcomes $\tilde{y}$ for new predictors $\tilde{x}$ is given by

$$p(\tilde{y} \mid \tilde{x}, x, y) \propto p(\tilde{y} \mid \tilde{x}, \beta, \sigma)\, p(\beta, \sigma | x, y).$$

The posterior mean,

$$\bar{\tilde{y}} \ = \int \tilde{y}\, p(\tilde{y} \mid \tilde{x}, x, y)\, d(\beta, \sigma),$$

may be computed by MCMC and provides the Bayesian estimate minimizing expected square error.

The second generated quantity is an indicator for whether $\beta_2 > \beta_3$, conditioned on the observed data x and y. With MCMC, it is then possible to compute the event probability that $\beta_2 > \beta_3$,

$$\Pr[\beta_2 > \beta_3 \mid x, y] = \int \mathbb{I}[\beta_2 > \beta_3]\, p(\beta, \sigma | x, y)\, d(\beta, \sigma).$$

3 Stan Programming Language

Stan enforces strong static typing of variables and expressions (like Java, C, and Fortran but unlike R or Python). Stan's underlying basic data types are integer, real, vector, row vector, and matrix; arrays of any type (including other arrays) are allowed.

Expressions include integer and double literals, variables, functions applied to a sequence of expressions, logical, comparison, and arithmetic operators for scalars, matrices, and vectors, indexed containers including slicing and index composition, and an ordinary differential equation solver. Type inference calculates the type of each object, with integer promotion to real values where necessary (including covariant types so that integer arrays may be assigned to real arrays). There is an extensive library of built-in functions, including most commonly used linear algebra, probability, and special functions used for statistical modeling.

Stan also allows user-defined functions, including new probability functions or random number generators, to be defined in the Stan language. These functions can be recursive.

Stan is an imperative programming language, with syntax that should be familiar to those with experience in Fortran, Java, C, R, or Python. Statements include assignments, log-density increments (including sampling statements with or without), for- and while-loops, conditionals (if-then-else), print, and rejection (exception) statements; blocks may contain local variable declarations.

Variables are declared based on their purpose. Known quantities, such as constants (N in the example), predictors (x) and modeled observed data (y) are declared as data. Unknown quantities, typically model parameters, missing data, or latent values, are declared as parameters. Predictive quantities ($\tilde{y}$) are declared and defined in the generated quantities block. The generated quantities block does not affect inference; semi-supervised learning is supported directly by declaring predictive quantities as parameters and defining them in the model block.

There are further transformed data and transformed parameter blocks which allow new constants or random variables to be defined in terms of other ones, just as in the generated quantities block; variables declared in these top-level blocks scope over subsequent blocks.

4 Stan Progam Execution

The Stan compiler parses a Stan program into an abstract syntax tree, then generates the code for a C++ class that is used to store data and define the log densities for estimation and inference.

Procedurally, the data is first read in through the constructor, validated against any declared constraints (with located error messages), and stored in member variables in the class instance. Next, transformed data variables are computed and their constraints validated.

After the class is instantiated, it provides a templated method to compute log densities for new parameter values on the unconstrained scale. This log density may be set to (a) automatically drop constant terms in the density (implemented via traits template metaprograms) and (b) include or exclude the Jacobian for the inverse transform from the unconstrained to constrained parameters.

To evaluate a log density, first the parameters are transformed to the unconstrained scale. For example, in the above program, the variable σ is declared with a lower-bound of zero, and Stan uses a log transfrom to convert that to the unconstrained value $\log(\sigma)$. The inverse transform is applied to $\log \sigma$ to produce $\exp(\log(\sigma))$, and the log Jacobian adjustment of $\log |\exp'(\log(\sigma))| = \sigma$ is added to the log density if Jacobians are configured to apply (through function template parameters). Log-odds transforms (with offsets) are used for upper- and lower-bounded variables, a stick-breaking process is used for simplexes, Cholesky factors with positive diagonals are used for covariance matrices, and Cholesky factors with positive diagonals and rows of unit Euclidean length are used for correlation matrices. Stan also supports Cholesky factor data types for correlation and covariance matrices, which are much more efficient for evaluating multivariate densities.

After the parameters are inverse transformed back to the constrained scale, the transformed parameters block is executed and the constraints validated. In every block other than the parameters block, constraints are simply validated at the end of the block.

Then the code in the model block is evaluated directly in the order statements are written. Sampling statements such as `sigma ~ lognormal(0, 1)` are just syntactic sugar; they increment the log density directly and are equivalent (up to constant dropping) to `increment_log_prob(lognormal_log(sigma, 0, 1))`, which directly increments the underlying log density accumulator.

Derivatives of the log density function (with respect to the unconstrained parameters) are computed by instantiating the function's scalar template parameters to automatic differentiation variables, evaluating the function, then propagating derivatives backwards from the result through the expression graph of all of the operations defining the log density to the parameters. Gradients (first-order derivatives) are evaluated to machine precision, requiring only a constant multiplier (independent of dimensionality) beyond the log density function; Hessians (second-order derivatives) are evaluated similarly, but require a log density evaluation per dimension, leading to a total computation time of $\mathcal{O}(N^2)$ when the log density evaluates in $\mathcal{O}(N)$ time. Stan's math library supports even

higher-order derivatives, as well as gradient-vector products, Hessian-vector products and a few more esoteric operations.

5 Statistical Algorithms

Stan provides three varieties of gradient-based statistical algorithms, all of which can be applied automatically given a Stan program. In other words, inference and estimation is a separate module from the language defining a log density and generated quantities.

Markov chain Monte Carlo sampling: Given the Stan program and data values, Stan is able to sample from the posterior $p(\beta, \sigma|y, x)$ in order to compute expectations such as parameter estimates and event probabilities. Stan uses an adaptive form of Euclidean Hamiltonian Monte Carlo [3], which, unlike other inference algorithms, scales well with dimension [4].

Laplace approximation: Stan can also find the Laplace approximation to the posterior, which is a multivariate normal distribution centered on the posterior mode, $(\beta^*, \sigma^*) = \arg\max_{\beta,\sigma} p(\beta, \sigma|x, y)$ with covariance estimated by curvature at the mode (specifically, the inverse Hessian of the negative log posterior). Posterior modes are found with L-BFGS, a quasi-Newton method based on gradients and an approximation of the Hessian [5]. Posterior curvature is calculated with automatic differentiation.

Penalized maximum likelihood estimation This is just an alternative perspective on the Laplace approximation—if the prior is uniform, the posterior mode is the maximum liklelihood estimate and the posterior covariance gives the standard error. Interpreting a non-uniform prior as a penalty function, the posterior mode is a penalized maximum likelihood estimate.

Variational inference Stan also supplies an experimental variational inference algorithm which fits a multivariate normal approximation to the posterior. Unlike the Lapace approximations, which finds posterior modes, Bayesian variational techniques aim to find an approximate posterior mean. Stan supports diagonal (aka mean-field) covariance matrices, as well as dense covariance matrices. (the latter leading to "mean-field" approximations). Variational inference is performed using Monte Carlo estimation of the gradient of the required expectation calculations [6].

6 Stan Math Library

The Stan math library [7] is a standalone project underlying the rest of Stan and supplying the differentiable matrix, linear algebra, vectorized probability functions (along with CDFs and complementary CDFs to support truncated distributions), and other special functions (log sum of exponents, Bessel functions, etc.). It is pure header-only C++ and may be used on its own outside of the context of Stan.

References

[1] Stan Development Team (2015) Stan Modeling Language User's Guide and Reference Manual. `http://mc-stan.org/documentation`

[2] Gelman, A., J. B. Carlin, Hal S. Stern, David B. Dunson, Aki Vehtari, Donald B. Rubin (2013) *Bayesian Data Analysis*, 3rd Edition. Chapman & Hall.

[3] Hoffman, M. and A. Gelman (2014) The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research* 15(Apr):15931623.

[4] Neal, R. (2011) MCMC using Hamiltonian dynamics. In *Handbook of Markov Chain Monte Carlo*, ed. Brooks, S., A. Gelman, G. L. Jones, and X.-L. Meng, 116–162. Chapman & Hall.

[5] Nocedal, J. (1980) Updating quasi-Newton matrices with limited storage. *Mathematics of Computation* 35(151):773–782.

[6] Kucukelbir, A., R. Ranganath, A. Gelman, D. M. Blei (2015) Automatic variational inference in Stan. *arXiv*:1506.03431.

[7] Carpenter, B., M. D. Hoffman, M. Brubaker, D. Lee, P. Li, and M. Betancourt (2015) The Stan Math Library: Reverse-mode automatic differentiation in C++. *arXiv*:1509.07164.