Expander Framework for Generating High-Dimensional GLM Gradient and Hessian from Low-Dimensional Base Distributions: R Package RegressionFactory

Alireza S. Mahani
Scientific Computing Group
Sentrana Inc.

Mansour T.A. Sharabiani
National Heart and Lung Institute
Imperial College London

Abstract

The R package **RegressionFactory** provides expander functions for constructing the high-dimensional gradient vector and Hessian matrix of the log-likelihood function for generalized linear models, from the lower-dimensional base-distribution derivatives. The software follows a modular implementation using the chain rule of derivatives. Such modularity offers a clear separation of case-specific components (base distribution functional form and link functions) from common steps (e.g., matrix algebra operations needed for expansion) in calculating log-likelihood derivatives. In doing so, **RegressionFactory** offers several advantages: 1) It provides a fast and convenient method for constructing log-likelihood and its derivatives by requiring only the low-dimensional, base-distribution derivatives, 2) The accompanying definiteness-invariance theorem allows researchers to reason about the negative-definiteness of the log-likelihood Hessian in the much lower-dimensional space of the base distributions, 3) The factorized, abstract view of regression suggests opportunities to generate novel regression models, and 4) Computational techniques for performance optimization can be developed generically in the abstract framework and be readily applicable across all the specific regression instances.

Keywords: negative definiteness, regression, optimization, sampling.

1. Introduction

Generalized Linear Models (GLMs) (McCullagh and Nelder 1989) are one of the most widely-used classes of models in statistical analysis, and their properties have been thoroughly studied and documented (see, for example, Dunteman and Ho (2006)). Model training and prediction for GLMs often involves Maximum-Likelihood estimation (frequentist approaches) or posterior density estimation (Bayesian approaches), both of which require application of optimization or MCMC sampling techniques to the log-likelihood function or some function containing it. Differentiable functions often benefit from optimization/sampling algorithms that utilize the first and/or second derivative of the function (Press 2007). With proper choice of link functions, many GLMs have log-likelihood functions that are not only twice-differentiable, but also globally-concave (Gilks and Wild 1992), making them ideal candidates for optimization/sampling routines that take advantage of these properties. For example,

the most common optimization approach for GLMs is Iterative Reweighted Least Squares (IRLS) (Gentle 2007, Section 6.8.1). IRLS is a disguised form of Newton-Raphson optimization (Wright and Nocedal 1999), which uses both the gradient and Hessian of the function, and relies on global concavity for convergence. When Hessian is too expensive to calculate or lacks definiteness, other optimization techniques such as conjugate gradient (Press 2007, Section 10.6) can be used, which still require the first derivative of the function. Among MCMC sampling algorithms, Adaptive Rejection Sampler (Gilks and Wild 1992) uses the first derivative and requires concavity of the log-density. Stochastic Newton Sampler (Qi and Minka 2002; Mahani, Hasan, Jiang, and Sharabiani 2014), a Metropolis-Hastings sampler using a locally-fitted multivariate Gaussian, uses both first and second derivatives and also requires log-concavity. Other techniques such as Hamiltonian Monte Carlo (HMC) (Neal 2011) use the first derivative of log-density, while their recent adaptations can use second and even third derivative information to adjust the mass matrix to local space geometry (Girolami and Calderhead 2011). Efficient implementation and analysis of GLM derivatives and their properties, therefore, is a key component to our ability to build probabilistic models using the powerful GLM framework.

The R package **RegressionFactory** contributes to computational research and development on GLM-based statistical models by providing an abstract framework for constructing, and reasoning about, GLM-like log-likelihood functions and their derivatives. Its modular implementation can be viewed as code factorization using the chain rule of derivatives (Apostol 1974). It offers a clear separation of generic steps (expander functions) from model-specific steps (base functions). New regression models can be readily implemented by supplying their base function implementation. Since base functions are in the much lower-dimensional space of the underlying probability distribution (often a member of the exponential family with one or two parameters), implementation of their derivatives is much easier than doing so in the high-dimensional space of regression coefficients. A by-product of this code refactoring using the chain rule is an invariance theorem governing the negative definiteness of the log-likelihood Hessian. The theorem allows this property to be studied in the base-distribution space, again a much easier task than doing so in the high-dimensional coefficient space. The modular organization of **RegressionFactory** also allows for performance optimization techniques to be made available across a broad set of regression models. This is particularly true for optimizations applied to expander functions, but also applies to base functions since they share many concepts and operations across models. **RegressionFactory** contains a lower-level set of tools compared to the facilities provided by mainstream regression utilities such as the `glm` command in R, or the package **dglm** (Dunn and Smyth 2014) for building double (varying-dispersion) GLM models. Therefore, in addition to supporting research on optimization/sampling algorithms for GLMs as well as research on performance optimization for GLM derivative-calculation routines, exposing the log-likelihood derivatives using the modular framework of **RegressionFactory** allows modelers to construct composite models from GLM lego blocks, including Hierarchical Bayesian models (Gelman and Hill 2006).

The rest of the paper is organized as follows. In Section 2, we begin with an overview of GLM models and arrive at our abstract, and expanded, representation of GLM log-likelihoods (2.1). We then apply the chain rule of derviatives to this abstract expression to derive two equivalent sets of factorized equations (compact and explicit forms) for computing log-likelihood gradient and Hessian using their base-function counterparts (2.2). We use the explicit forms of the equations to prove a negative-definiteness invariance theorem for the log-likelihood

Hessian (2.3). Section 3 discusses the implementation of the aforementioned factorized code in **RegressionFactory** using the expander functions (3.1) and the base functions (3.2). In Section 4, we illustrate the use of **RegressionFactory** using examples from single-parameter and multi-parameter base functions. Finally, Section 5 offers a summary of results, and pointers to future research and development directions.

2. Theory

In this section we develop the theoretical foundation for **RegressionFactory**, beginning with an overview of GLM models.

2.1. Overview of GLMs

In GLMs, response variable[1] y is assumed to be generated from an exponential-family distribution, and its expected value is related to linear predictor $\mathbf{x}^t\boldsymbol{\beta}$ via the link function g:

$$g(\mathrm{E}(y)) = \mathbf{x}^t\boldsymbol{\beta}. \tag{1}$$

where $\mathbf{x}$ is the vector of covariates and $\boldsymbol{\beta}$ is the vector of coefficients. For single-parameter distributions, there is often a simple relationship between the distribution parameter and its mean. Combined with Equation 1, this is sufficient to define the distribution in terms of the linear predictor, $\mathbf{x}^t\boldsymbol{\beta}$. For many double-parameter distributions, the distribution can be expressed as

$$f_Y(y;\theta,\Phi) = \exp\{\frac{y\theta - B(\theta)}{\Phi} + C(y,\Phi)\} \tag{2}$$

where range of y does not depend on θ or Φ. This function can be maximized with respect to θ without knowledge of Φ. Same is true if we have multiple conditionally-independent data points, where log-likelihood takes a summative form. Once θ is found, we can find Φ (dispersion parameter) through maximization or method of moments, as done by `glm` in R. Generalization to varying-dispersion models is offered in the R package **dglm**, where both mean and dispersion are assumed to be linear functions of covariates. In **dglm** estimation is done iteratively by alternating between an ordinary GLM and a dual GLM in which the deviance components of the ordinary GLM appear as responses (Smyth 1989).

In **RegressionFactory**, we take a more general approach to GLMs that encompasses the `glm` and `dglm` approaches but is more flexible. Our basic assumption is that log-density for each data point can be written as:

$$\log\left(\mathrm{P}(y\,|\,\{\,\mathbf{x}^j\}_{j=1,\ldots,J})\right) = f(<\mathbf{x}^1,\boldsymbol{\beta}^1>,\ldots,<\mathbf{x}^J,\boldsymbol{\beta}^J>,y) \tag{3}$$

where $<a,b>$ means inner product of vectors a and b. Note that we have absorbed the nonlinearities introduced through one or more link functions into the definition of f. For N conditionally-independent observations $y_1,\ldots,y_N$, the log-likelihood as a function of coefficients $\boldsymbol{\beta}^j$ is given by:

$$L(\boldsymbol{\beta}^1,\ldots,\boldsymbol{\beta}^J) = \sum_{n=1}^{N} f_n(<\mathbf{x}_n^1,\boldsymbol{\beta}^1>,\ldots,<\mathbf{x}_n^J,\boldsymbol{\beta}^J>), \tag{4}$$

[1] To simplify notation, we assume that response variable is scalar, but generalization to vector response variables is straightforward.

where we have absorbed the dependence of each term on y_n into the indexes of the base functions $f_n(u^1, \ldots, u^J)$. With proper choice of nonlinear transformations, we can assume that the domain of L is $\mathbb{R}^{\sum_j K^j}$, where K^j is the dimensionality of $\boldsymbol{\beta}^j$.

This view of GLMs naturally unites single-parameter GLMs such as Binomial (with fixed number of trials) and Poisson, constant-dispersion two-parameter GLMs (e.g. normal and Gamma), varying-dispersion two-parameter GLMs (e.g. heteroscedastic normal regression), and multi-parameter models such as multinomial logit. Several examples are discussed in Section 4.

Our next step is to apply the chain rule of derivatives to Equation 4 to express the high-dimensional ($\sum_j K^j$) derivatives of L in terms of the low-dimensional (J) derivatives of f_n's. We will see that the resulting expressions offer a natural way for modular implementation of GLM derivatives.

2.2. Application of chain rule

First, we define our notation for representing derivative objects. We concatenate all J coefficient vectors, $\boldsymbol{\beta}^j$'s, into a single $\sum_j K^j$-dimensional vector, $\boldsymbol{\beta}$:

$$\boldsymbol{\beta} \equiv (\boldsymbol{\beta}^{1,t}, \ldots, \boldsymbol{\beta}^{J,t})^t. \tag{5}$$

The first derivative of log-likelihood can be written as:

$$\mathbf{G}(\boldsymbol{\beta}) \equiv \frac{\partial L}{\partial \boldsymbol{\beta}} = ((\frac{\partial L}{\partial \boldsymbol{\beta}^1})^t, \ldots, (\frac{\partial L}{\partial \boldsymbol{\beta}^J})^t)^t, \tag{6}$$

where

$$(\frac{\partial L}{\partial \boldsymbol{\beta}^j})^t \equiv (\frac{\partial L}{\partial \beta^j_1}, \ldots, \frac{\partial L}{\partial \beta^j_{K^j}}). \tag{7}$$

For second derivatives we have:

$$\mathbf{H}(\boldsymbol{\beta}) \equiv \frac{\partial^2 L}{\partial \boldsymbol{\beta}^2} = \left[\frac{\partial^2 L}{\partial \boldsymbol{\beta}^j \partial \boldsymbol{\beta}^{j'}}\right]_{j,j'=1,\ldots,J}, \tag{8}$$

where we have defined $\mathbf{H}(\boldsymbol{\beta})$ in terms of J^2 matrix blocks:

$$\frac{\partial^2 L}{\partial \boldsymbol{\beta}^j \partial \boldsymbol{\beta}^{j'}} \equiv \left[\frac{\partial L}{\partial \beta^j_k \partial \beta^{j'}_{k'}}\right]_{j=1,\ldots,K^j; j'=1,\ldots,K^{j'}} \tag{9}$$

Applying the chain rule to the log-likelihood function of Equation 4, we derive expressions for its first and second derivatives as a function of the derivatives of the base functions $f_1, \ldots, f_N$:

$$\frac{\partial L}{\partial \boldsymbol{\beta}^j} = \sum_{n=1}^{N} \frac{\partial f_n}{\partial \boldsymbol{\beta}^j} = \sum_{n=1}^{N} \frac{\partial f_n}{\partial u^j} \mathbf{x}^j_n = \mathbf{X}^{j,t} \mathbf{g}^j, \tag{10}$$

with

$$\mathbf{g}^j \equiv (\frac{\partial f_1}{\partial u^j}, \ldots, \frac{\partial f_N}{\partial u^j})^t, \tag{11}$$

and

$$\mathbf{X}^j \equiv (\mathbf{x}^j_1, \ldots, \mathbf{x}^j_N)^t. \tag{12}$$

Similarly, for the second derivative we have:

$$\frac{\partial^2 L}{\partial \boldsymbol{\beta}^j \partial \boldsymbol{\beta}^{j'}} = \sum_{n=1}^{N} \frac{\partial^2 f_n}{\partial \boldsymbol{\beta}^j \partial \boldsymbol{\beta}^{j'}} = \sum_{n=1}^{N} \frac{\partial^2 f_n}{\partial u^j \partial u^{j'}} \left(\mathbf{x}_n^j \otimes \mathbf{x}_n^{j'}\right) = \mathbf{X}^{j,t} \mathbf{h}^{jj'} \mathbf{X}^{j'}, \tag{13}$$

where $\mathbf{h}^{jj'}$ is a diagonal matrix of size N with n'th diagonal element defined as:

$$h_n^{jj'} \equiv \frac{\partial^2 f_n}{\partial u^j \partial u^{j'}} \tag{14}$$

We refer to the matrix form of the Equations 10 and 13 as 'compact' forms, and the explicit-sum forms as 'explicit' forms. The expander functions in **RegressionFactory** use the compact form to implement the high-dimensional gradient and Hessian (see Section 3.1), while the definiteness-invariance theorem below utilizes the explicit-sum form of Equation 13.

2.3. Definiteness invariance of Hessian

Theorem 1. *If all f_n's in Equation 4 have negative definite Hessians AND if at least one of J matrices $\mathbf{X}^j \equiv (\mathbf{x}_1^j, \ldots, \mathbf{x}_N^j)^t$ is full rank, then $L(\boldsymbol{\beta}^1, \ldots, \boldsymbol{\beta}^J)$ also has a negative-definite Hessian.*

Proof. To prove negative-definiteness of $\mathbf{H}(\boldsymbol{\beta})$ (hereafter referred to as $\mathbf{H}$ for brevity), we seek to prove that $\mathbf{p}^t \mathbf{H} \mathbf{p}$ is negative for all non-zero $\mathbf{p}$ in $\mathbb{R}^{\sum_j K^j}$. We begin by decomposing $\mathbf{p}$ into J subvectors of length K^j each:

$$\mathbf{p} = (\mathbf{p}^{1,t}, \ldots, \mathbf{p}^{J,t})^t. \tag{15}$$

We now have:

$$\begin{aligned}
\mathbf{p}^t \mathbf{H} \mathbf{p} &= \sum_{j,j'=1}^{J} \mathbf{p}^{j,t} \frac{\partial^2 L}{\partial \boldsymbol{\beta}^j \partial \boldsymbol{\beta}^{j'}} \mathbf{p}^{j'} && (16)\\
&= \sum_{j,j'} \mathbf{p}^{j,t} \left(\sum_n \frac{\partial^2 f_n}{\partial u^j \partial u^{j'}} \cdot \left(\mathbf{x}_n^j \otimes \mathbf{x}_n^{j'}\right) \right) \mathbf{p}^{j'} && (17)\\
&= \sum_n \sum_{j,j'} \frac{\partial^2 f_n}{\partial u^j \partial u^{j'}} \mathbf{p}^{j,t} \left(\mathbf{x}_n^j \otimes \mathbf{x}_n^{j'}\right) \mathbf{p}^{j'} && (18)
\end{aligned}$$

If we define a set of new vectors $\mathbf{q}_n$ as:

$$\mathbf{q}_n \equiv \left[\mathbf{p}^{1,t}\mathbf{x}_n^1 \quad \cdots \quad \mathbf{p}^{J,t}\mathbf{x}_n^J\right], \tag{19}$$

and use $\mathbf{h}_n$ to denote the J-by-J Hessian of f_n:

$$\mathbf{h}_n \equiv [h_n^{jj'}]_{j,j'=1,\ldots,J}, \tag{20}$$

we can write:

$$\mathbf{p}^t \mathbf{H} \mathbf{p} = \sum_n \mathbf{q}_n^t \, \mathbf{h}_n \, \mathbf{q}_n. \tag{21}$$

Since all $\mathbf{h}_n$'s are assumed to be negative definite, all $\mathbf{q}_n^t \, \mathbf{h}_n \, \mathbf{q}_n$ terms must be non-positive. Therefore, $\mathbf{p}^t\mathbf{H}\mathbf{p}$ can be non-negative only if all its terms are zero, which is possible only if all $\mathbf{q}_n$'s are zero vectors. This, in turn, means we must have $\mathbf{p}^{j,t}\mathbf{x}_n^j = 0, \ \forall \, n, j$. In other words, we must have $\mathbf{X}^j\mathbf{p}^j = \emptyset, \ \forall \, j$. This means that all $\mathbf{X}^j$'s have non-singleton nullspaces and therefore cannot be full-rank, which contradicts our assumption. Therefore, $\mathbf{p}^T\mathbf{H}\mathbf{p}$ must be negative. This proves that $\mathbf{H}$ is negative definite. □

Proving negative-definiteness in the low-dimensional space of base functions is often much easier. For single-parameter distributions, we simply have to prove that the second derivative is negative. For two-parameter distributions, and according to Silvester's criterion (Gilbert 1991), it is sufficient to show that both diagonal elements of the base-distribution Hessian as well as its determinant are negative. Note that negative-definiteness depends not only on the distribution but also on the choice of link function(s). For twice-differentiable functions, negative-definiteness of Hessian and log-concavity are equivalent (Boyd and Vandenberghe 2009). Gilks and Wild (1992) have a list of log-concave distributions and link functions.

3. Implementation

RegressionFactory is a direct implementation of compact expressions in Equations 10 and 13. These expressions imply a code refactoring by separating model-specific steps (calculation of $\mathbf{g}^j$ and $\mathbf{h}^{jj'}$) from generic steps (calculation of linear predictors $\mathbf{X}^j\boldsymbol{\beta}^j$ as well as $\mathbf{X}^{j,t}\mathbf{g}^j$ and $\mathbf{X}^{j,t}\mathbf{h}^{jj'}\mathbf{X}^{j'}$). This decomposition is captured diagramatically in the system flow diagram of Figure 1.

3.1. Expander functions

Current implementation of **RegressionFactory** contains expander and base functions for one-parameter and two-parameter distributions. This covers the majority of interesting GLM cases, and a few more. A notable exception is multinomial regression models (such as logit and probit) which can have an unspecified number of slots. The package can be extended in the future to accommodate such more general cases.

Single-parameter expander function

Below is the source code for `regfac.expand.1par`:

```
R> regfac.expand.1par <- function(beta, X, y, fbase1, fgh = 2, ...) {
+    # obtain base distribution derivatives
+    ret <- fbase1(X %*% beta, y, fgh, ...)
+    # expand base derivatives
+    f <- sum(ret$f)
+    if (fgh == 0) return (f)
+    g <- t(X) %*% ret$g
+    if (fgh == 1) return (list(f = f, g = g))
+    xtw <- 0*X
+    for (k in 1:ncol(X)) xtw[, k] <- X[, k] * ret$h
+    h <- t(xtw) %*% X
```

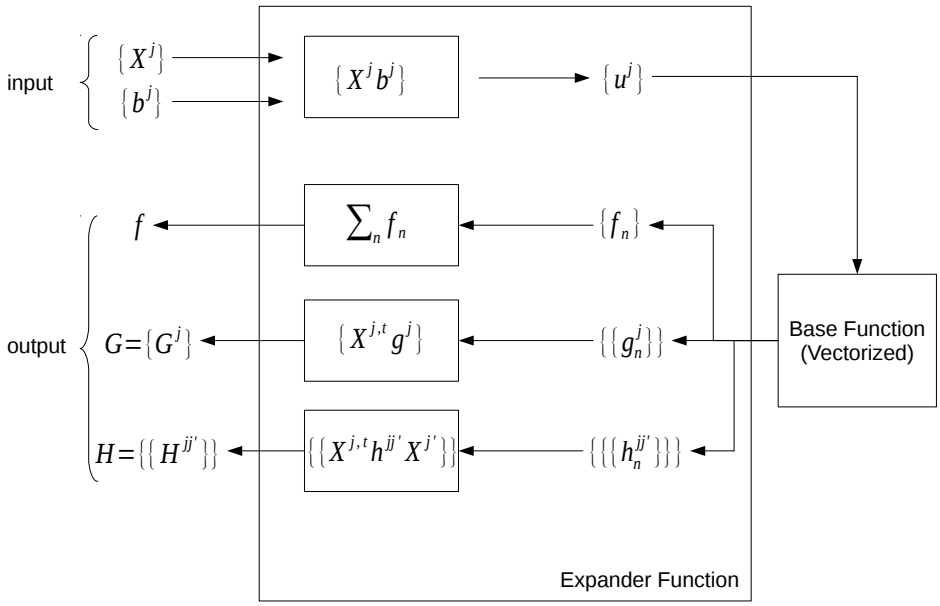

Figure 1: System flow diagram for **RegressionFactory**. The expander function is responsible for calculation of log-likelihood and its gradient and Hessian in the high-dimensional space of regression coefficients. It does so by calculating the linear predictors and supplying them to the base function, which is responsible for calculation of log-likelihood and its gradient and Hessian for each data point, in the low-dimensional space of the underlying probability distribution. The expander function converts these low-dimensional objects into the high-dimensional forms, using generic matrix-algebra operations.

.

```
+    return (list(f = f, g = g, h = h))
+  }
```

`beta` is the vector of coefficients, `X` is the matrix of covariates, `y` is the vector (or matrix) of response variable, `fbase1` is the single-parameter base function being expanded, and `fgh` is a flag indicating whether the gradient or Hessian must be returned or not. The `dots` argument (`...`) is used for passing special, fixed arguments such as the number of trials in a binomial regression. The vectorized function `fbase` is expected to return a list of three vectors: `f`, `g` and `h`, corresponding to the base distribution, its first derivative and its second derivative (all vectors of length N or `nrow(X)`). The second and third elements correspond to $\mathbf{g}^1$ and $\mathbf{h}^{11}$ in our notation. Several design aspects of the code are noteworthy for computational efficiency:

1. Since $\mathbf{h}$ is diagonal, we only need to return the N diagonal elements.
2. For the same reason, rather than multiplying $\mathbf{h}$ by $\mathbf{X}$, we only multiply the vector of diagonal elements by each of the K columns of $\mathbf{X}$.
3. The flag `fgh` controls whether a) only the function value must be returned (`fgh==0`), b) only the function and its first derivative must be returned (`fgh==1`), or c) the function as well as its first and second derivative must be returned (`fgh==2`). This allows optimization or sampling algorithms that do not the first or second derivative to avoid paying an unnecessary computational penalty. Since most often a higher-level derivative implies the need for lower-level derivative(s) (including the function as zero'th derivative), and also since the computational cost of higher derivatives is much higher, the `fgh` flag works in an incremental fashion (only 3 options) rather than covering all permutations of `f,g,h`.

Two-parameter expander function

Below is the source code for `regfac.expand.2par`, the 2D expander function in **RegressionFactory**:

```
R> regfac.expand.2par <- function(coeff, X
+    , Z=matrix(1.0, nrow = nrow(X), ncol = 1)
+    , y, fbase2, fgh = 2, block.diag = FALSE
+    , ...) {
+    # extracting coefficients of X and Z
+    K1 <- ncol(X); K2 <- ncol(Z)
+    beta <- coeff[1:K1]
+    gamma <- coeff[K1 + 1:K2]
+
+    # obtain base distribution derivatives
+    ret <- fbase2(X %*% beta, Z %*% gamma, y, fgh, ...)
+
+    # expand base derivatives
+    # function
+    f <- sum(ret$f)
+    if (fgh == 0) return (f)
```

```
+    # gradient
+    g <- c(t(X) %*% ret$g[, 1], t(Z) %*% ret$g[, 2])
+    if (fgh == 1) return (list(f = f, g = g))
+    # Hessian
+    h <- array(0, dim=c(K1+K2, K1+K2))
+    # XX block
+    xtw <- 0 * X
+    for (k in 1:K1) xtw[, k] <- X[, k] * ret$h[, 1]
+    h[1:K1, 1:K1] <- t(xtw) %*% X
+    # ZZ block
+    ztw <- 0 * Z
+    for (k in 1:K2) ztw[, k] <- Z[, k] * ret$h[, 2]
+    h[K1 + 1:K2, K1 + 1:K2] <- t(ztw) %*% Z
+          # XZ and ZX blocks
+    if (!block.diag) {
+            ztw2 <- 0*Z
+            for (k in 1:K2) ztw2[,k] <- Z[,k]*ret$h[,3]
+            h[K1 + 1:K2, 1:K1] <- t(ztw2)%*%X
+            h[1:K1, K1 + 1:K2] <- t(h[K1 + 1:K2, 1:K1])
+          }
+
+    return (list(f = f, g = g, h = h))
+  }
```

Aside from the same performance optimization techniques used for the one-parameter expander function, the two-parameter expander function has an additional parameter, `block.diag`. When `TRUE` it sets the cross-derivative terms between the two slots to zero. It can be useful in two scenarios: 1) When the full Hessian is not negative definite, but the Hessian for each parameter is. Block-diagonalization allows for optimization and sampling techniques that rely on this property to be used, at the expense of potentially slower convergence since the block-diagonalized Hessian is not accurate, 2) When optimization of one slot can proceed without knowledge of the value of the other slot, as in many two-parameter exponential family members where the dispersion parameter can be ignored in ML estimation of the mean parameter (e.g. in normal distribution).

3.2. Base distributions

Corresponding to the one-parameter and two-parameter expander functions, **RegressionFactory** offers many of the standard base distributions used in GLM models. Using the nomenclature of `glm`, current version contains the following base distributions and link functions:

- One-parameter distributions:
  - Binomial (logit, probit, cauchit, cloglog)
  - Poisson (log)
  - Exponential (log)
  - Geometric (logit)

- Two-parameter distributions:
  - Gaussian (identity / log)
  - Inverse Gaussian (log / log)
  - Gamma (log / log)

A few points are worth mentioning regarding the choice of base distributions and link functions:

1. Naming convention: We generally follow this convention for single-parameter distributions:

   `fbase.<distribution>.<mean link function>`

   and this convention for two-parameter distributions:

   `fbase.<distribution>.<mean link function>.<dispersion link function>`

   There are can be exceptions. For example, in geometric regression

   `fbase.geometric.logit`

   the linear predictor is assumed to be `logit` of the sucess probability, which is inverse of the distribution mean. Thus, technically the link function is `-log(mu-1)`, but for brevity we simply refer to this link function as `logit`. Ultimately, naming conventions are less important than the definition of log-likelihood function, which combines the distribution and the link functions.

2. Since the focus of **RegressionFactory** is on supporting optimization and sampling algorithms for GLM-like models, we are not interested in constant terms in the log-likelihood, i.e. terms that are independent of the regression coefficients. Therefore, we can omit them from the base functions for computational efficiency. An example is the log-factorial term in the Poisson base distribution. (Note that such constant terms are automatically differentiated out of the gradient and Hessian.) If needed, users can implement thin wrappers around the base functions to add the constant terms to the log-likelihood.

3. Our preference is to choose link functions that map the natural domain of the distribution parameter to the real space. For example, in Poisson distribution the natural domain of the distribution mean is the positive real space. The `log` link function maps this natural domain to the entire real space. However, for `identity` and `sqrt` link functions the range is positive real space.

4. We also prefer link functions that produce negative-definiteness for the entire Hessian, or at least for Hessian blocks (corresponding to a subset of the base-distribution parameters). This allow for more optimization/sampling algorithms that take advantage of concavity to be applied to the expanded log-likelihood (according to Theorem 1).

5. We have chosen to absorb the link functions into the function names and their implementation, rather than making distribution names and lonk functions parameters of a single base function. Doing the latter is certainly possible, offering usability at computational cost. Our current choice is driven by the fact that the primary target of **RegressionFactory** is developers rather than end-users.

4. Using RegressionFactory

The most basic application of **RegressionFactory** is to use the readily-available log-likelihood functions and derivatives. For example, one might be developing a Bayesian model where the log-likelihood is combined with the prior to form the posterior, which is then supplied to a sampling algorithm. Or one might be working on a new optimization algorithm and would like to test its correctness and performance on regression log-likelihood functions as an important use-case. Users can also supply their own base functions to the expander functions of **RegressionFactory** and readily obtain the log-likelihood and its derivatives. Implementation of functions for calculating base distribution derivatives is often quite simple, which can significantly reduce the time needed for prototyping a new regression model.

There are two equivalent approaches for passing the log-likelihood functions to an optimization/sampling routine: 1) Pass the expander function as the primary function, and the base function as an argument of the primary function, 2) write a thin wrapper that combines the expander and base functions, and pass this wrapper function to the optimization/sampling routine. If the log-likelihood function must be added to another function (such as a prior), then the second approach is the only option where the wrapper implements the logic for adding the two functions. Due to its higher versatility as well as higher code readability, we recommend the second approach.

The above point as well as other usage details are illustrated below, with several examples from single-parameter and double-parameter distributions.

4.1. Example 1: Bayesian GLM

The easiest way to take advantage of **RegressionFactory** is to utilize its standard GLM base functions in custom applications, either for testing the performance of a new optimization/sampling technique, or for composing more complex models from these lego blocks. In the first example, we show how a Bayesian GLM can be constructed in the **RegressionFactory** framework.

We begin with a basic implementation of Bayesian logistic regression using flat normal priors on each coefficient. First we must load the package into our R session:

```
R> library(RegressionFactory)
```

Log-likelihood for logistic regression can be readily constructed by applying the single-parameter expander function to the binomial base function and setting the number of trials equal to `1`:

```
R> loglike.logistic <- function(beta, X, y, fgh) {
+    regfac.expand.1par(beta, X, y, fbase1.binomial.logit, fgh, n=1)
+  }
```

We also need a prior for `beta`, which we assume to be a normal distribution on each of the `K` elements of `beta` with the same mean (`mu.beta`) and standard deviation (`sd.beta`):

```
R> logprior.logistic <- function(beta, mu.beta, sd.beta, fgh) {
+    f <- sum(dnorm(beta, mu.beta, sd.beta, log=TRUE))
+    if (fgh==0) return (f)
```

```
+    g <- -(beta-mu.beta)/sd.beta^2
+    if (fgh==1) return (list(f=f, g=g))
+    h <- diag(-1/sd.beta^2, nrow=length(beta))
+    return (list(f=f, g=g, h=h))
+  }
```

We can now combine the likelihood and prior according to Bayes rule to construct the log-posterior:

```
R> logpost.logistic <- function(beta, X, y, mu.beta, sd.beta, fgh) {
+    ret.loglike <- loglike.logistic(beta, X, y, fgh)
+    ret.logprior <- logprior.logistic(beta, mu.beta, sd.beta, fgh)
+    regfac.merge(ret.loglike, ret.logprior, fgh=fgh)
+  }
```

In the above, we have taken advantage of the utility function `regfac.merge` for combining two lists containing function values and its first two derivatives.

In order to test the above posterior function, we simulate some data using the generative model for logistic regression and estimate the coefficients using `glm` for reference:

```
R> N <- 1000
R> K <- 5
R> X <- matrix(runif(N*K, min=-0.5, max=+0.5), ncol=K)
R> beta <- runif(K, min=-0.5, max=+0.5)
R> y <- rbinom(N, size = 1, prob = 1/(1+exp(-X%*%beta)))
R> beta.glm <- glm(y~X-1, family="binomial")$coefficients
```

We now draw `1000` MCMC samples from the posterior of `beta` using Stochastic Newton Sampler (SNS), via R package **sns** (Mahani *et al.* 2014). We are taking advantage of the fact that the sum of two negative-definite Hessians is also negative-definite, a condition needed by SNS. Also, we assume that `mu.beta` and `sd.beta` are both given to provide a non-informative prior on `beta`. Finally, we run `sns` in non-stochastic mode via the flag `rnd=FALSE` to allow for better comparison of output with `glm`:

```
R> library(sns)
R> nsmp <- 10
R> mu.beta <- 0.0
R> sd.beta <- 1000
R> beta.smp <- array(NA, dim=c(nsmp,K))
R> beta.tmp <- rep(0,K)
R> for (n in 1:nsmp) {
+    beta.tmp <- sns(beta.tmp, fghEval=logpost.logistic, X=X, y=y
+      , mu.beta=mu.beta, sd.beta=sd.beta, fgh=2, rnd=FALSE)
+    beta.smp[n,] <- beta.tmp
+  }
R> beta.sns <- colMeans(beta.smp[(nsmp/2+1):nsmp,])
R> cbind(beta.glm, beta.sns)
```

```
      beta.glm    beta.sns
X1 -0.15663759 -0.15663759
X2  0.55149684  0.55149681
X3 -0.05057037 -0.05057037
X4  0.35592297  0.35592296
X5 -0.46922796 -0.46922794
```

Next, we consider a less-trivial example. We create a hierarchical structure where coefficients of `J` groups are assumed to be pooled from normal distribution. This is a simple example of Hierarchical Bayesian models, which due to lack of explanatory variables at the upper level is reduced to a random-coefficient model. We begin with data generation to provide the reader with a tangible grasp of the assumed generative model:

```
R> J <- 20
R> mu.beta.hb <- runif(K, min=-0.5, max=+0.5)
R> sd.beta.hb <- runif(K, min=0.5, max=1.0)
R> X.hb <- list()
R> y.hb <- list()
R> beta.hb <- array(NA, dim=c(J,K))
R> for (k in 1:K) {
+    beta.hb[,k] <- rnorm(J, mu.beta.hb[k], sd.beta.hb[k])
+  }
R> for (j in 1:J) {
+    X.hb[[j]] <- matrix(runif(N*K, min=-0.5, max=+0.5), ncol=K)
+    y.hb[[j]] <- rbinom(N, size=1, prob=1/(1+exp(-X%*%beta.hb[j,])))
+  }
```

Again, we generate `glm` coefficient estimates for reference. Note that `glm` treats the groups completely independently of each other, i.e. without any pooling:

```
R> beta.glm.all <- array(NA, dim=c(J,K))
R> for (j in 1:J) {
+    beta.glm.all[j,] <- glm(y.hb[[j]]~X.hb[[j]]-1
+      , family="binomial")$coefficients
+  }
```

Again, we draw samples from posterior on coefficients using SNS, turning the `rnd` flag off for better comparison. Also for code brevity and maintaining focus on how to use pkgRegressionFactory, we ignore sampling from the posterior of `mu.beta` and `sd.beta`, and assume their value is given. We must first construct the log-posteriors. Note that we do not need to change the definition of log-posteior, but the interpretation of `mu.beta` and `sd.beta` has changed fronm scalars to vector of length `K` each:

```
R> beta.smp.hb <- array(NA, dim=c(nsmp,J,K))
R> beta.tmp.hb <- array(0.0, dim=c(J,K))
R> for (n in 1:nsmp) {
+    for (j in 1:J) {
```

```
+      beta.tmp.hb[j,] <- sns(beta.tmp.hb[j,], fghEval=logpost.logistic
+        , X=X.hb[[j]], y=y.hb[[j]]
+        , mu.beta=mu.beta.hb, sd.beta=sd.beta.hb, fgh=2, rnd=F)
+    }
+    beta.smp.hb[n,,] <- beta.tmp.hb
+  }
R> beta.sns.hb <- apply(beta.smp.hb[(nsmp/2+1):nsmp,,], c(2,3), mean)
```

We have taken advantage of conditional independence [ref] of the coefficients of each group, given the values of `mu.beta` and `sd.beta`. We examine the coefficients of the first few groups between `glm` and HB methods:

```
R> head(beta.glm.all)
```

```
           [,1]        [,2]        [,3]        [,4]         [,5]
[1,] -0.3396661  0.27305217 -0.19075041  0.55307108  0.101447032
[2,] -0.2644097  0.01532541 -0.35153321 -0.11901128  0.119664117
[3,]  0.0622089 -0.06354912  0.05785568  0.64981819  0.005468755
[4,] -0.2215044  0.43696647 -0.17410285  0.16836517  0.128660638
[5,] -0.2042091 -0.05354939  0.14407166 -0.20358941 -0.464781283
[6,] -0.3175850 -0.07353676 -0.07471171  0.02768178  0.457276063
```

```
R> head(beta.sns.hb)
```

```
            [,1]         [,2]        [,3]        [,4]        [,5]
[1,] -0.30747643  0.239148576 -0.22249483  0.53561583  0.04358088
[2,] -0.23665109  0.002823354 -0.36521902 -0.04993022  0.05879485
[3,]  0.06850758 -0.075530322 -0.01005119  0.61489957 -0.04555926
[4,] -0.19297655  0.388902321 -0.20479515  0.19812199  0.06785943
[5,] -0.18050867 -0.061335483  0.07500439 -0.12775920 -0.46660111
[6,] -0.28769464 -0.076681661 -0.12594342  0.07171535  0.36110637
```

Plotting unpooled (`glm`) and pooled (`sns`) coefficients shows the typical shrinkage pattern of Bayesian models.

```
R> plot(beta.glm.all[,1], beta.sns.hb[,1]
+       , xlab="Unpooled Coefficients"
+       , ylab="Pooled Coefficients")
R> abline(a=0, b=1)
```

4.2. Example 2: Double-parameter GLM with varying dispersion

As a second example, we consider a double-parameter GLM with varying dispersion, i.e., dependent on the covariates. As of version 0.7.1, **RegressionFactory** contains three double-parameter base distributions: Gaussian, inverse Gaussian, and Gamma. These double-parameter distributions can be used in a constant-dispersion or varying-dispersion setting.

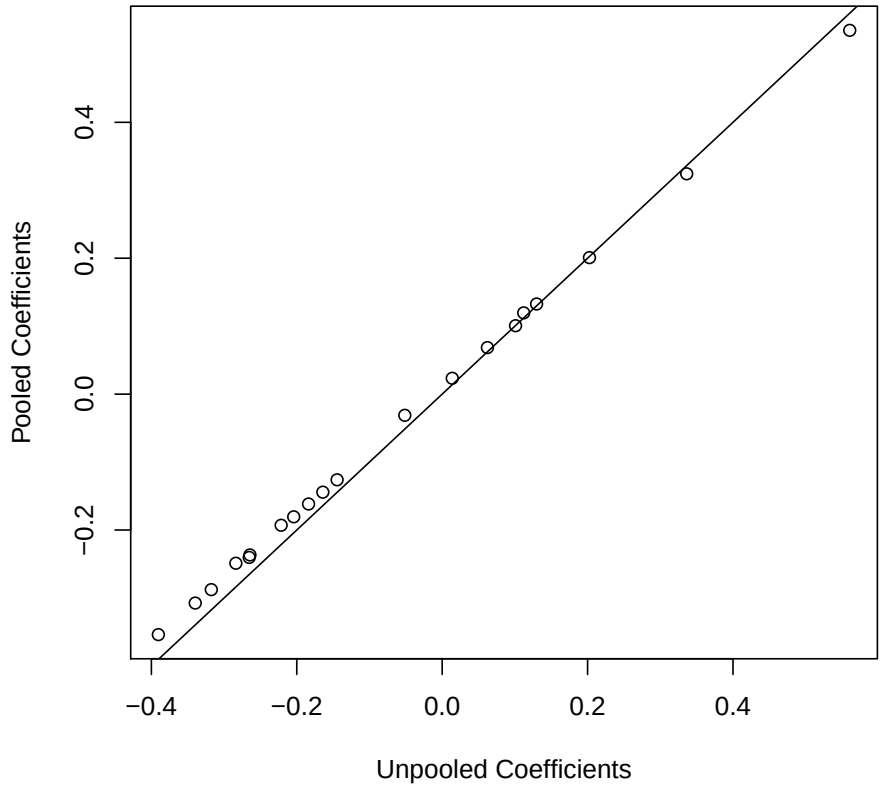

Figure 2: Pooling of logistic regression coefficients using a hierarchical Bayesian framework produces the familiar shrinkage towards the mean effect.

Constant-dispersion scenario is a special case of varying-dispersion scenario where the only covariate used to explain the dispersion parameter of the base distribution is intercept. This corresponds to the default value of `Z` in the function `regfac.expand.2par`.

First, we load the R package **dglm**:

```
R> library(dglm)
```

To use **RegressionFactory**, as before we implement a thin wrapper to combine the 2D expander with the normal base distribution:

```
R> loglike.linreg <- function(coeff, X, y, fgh, vd = F) {
+    if (vd) regfac.expand.2par(coeff = coeff, X = X, Z = X, y = y
+      , fbase2 = fbase2.gaussian.identity.log, fgh = fgh, block.diag = F)
+    else regfac.expand.2par(coeff = coeff, X = X, y = y
+      , fbase2 = fbase2.gaussian.identity.log, fgh = fgh, block.diag = F)
+  }
```

The boolean flag `vd` indicates whether we want to use covariates to explain the dispersion or not. If `FALSE`, the model is reduced to ordinary linear regression. Next, we simulate data according to the assumed generative model:

```
R> N <- 1000
R> K <- 5
R> X <- matrix(runif(N*K, min=-0.5, max=+0.5), ncol=K)
R> beta <- runif(K, min=-0.5, max=+0.5)
R> gamma <- runif(K, min=-0.5, max=+0.5)
R> mean.vec <- X%*%beta
R> sd.vec <- exp(X%*%gamma)
R> y <- rnorm(N, mean.vec, sd.vec)
```

We now estimate constant-dispersion and varying-dispersion models using the R commands `lm` and `dglm`:

```
R> # constant-dispersion model
R> est.glm <- lm(y~X-1)
R> beta.glm <- est.glm$coefficients
R> sigma.glm <- summary(est.glm)$sigma
R> # varying-dispersion model
R> est.dglm <- dglm(y~X-1, dformula = ~X-1, family = "gaussian", dlink = "log")
R> beta.dglm <- est.dglm$coefficients
R> gamma.dglm <- est.dglm$dispersion.fit$coefficients
```

Finally, we estimate the same models using the expander framework of **RegressionFactory**:

```
R> # constant-dispersion
R> coeff.smp <- array(NA, dim=c(nsmp, K+1))
R> coeff.tmp <- rep(0, K+1)
R> for (n in 1:nsmp) {
```

```
+    coeff.tmp <- sns(coeff.tmp, fghEval=loglike.linreg
+      , X=X, y=y, fgh=2, vd = F, rnd = F)
+    coeff.smp[n,] <- coeff.tmp
+  }
R> beta.sns.cd <- colMeans(coeff.smp[(nsmp/2+1):nsmp, 1:K])
R> sigma.sns.cd <- sqrt(exp(mean(coeff.smp[(nsmp/2+1):nsmp, K+1])))
R> cbind(beta.glm, beta.sns.cd)

      beta.glm beta.sns.cd
X1 -0.28185927 -0.28185927
X2  0.05293854  0.05293854
X3 -0.50058975 -0.50058975
X4 -0.08192276 -0.08192276
X5 -0.22391526 -0.22391526

R> cbind(sigma.glm, sigma.sns.cd)

     sigma.glm sigma.sns.cd
[1,]  1.024638     1.022073

R> # varying-dispersion
R> coeff.smp <- array(NA, dim=c(nsmp, 2*K))
R> coeff.tmp <- rep(0, 2*K)
R> for (n in 1:nsmp) {
+    coeff.tmp <- sns(coeff.tmp, fghEval=loglike.linreg
+      , X=X, y=y, fgh=2, vd = T, rnd = F)
+    coeff.smp[n,] <- coeff.tmp
+  }
R> beta.sns.vd <- colMeans(coeff.smp[(nsmp/2+1):nsmp, 1:K])
R> gamma.sns.vd <- colMeans(coeff.smp[(nsmp/2+1):nsmp, K+1:K])
R> cbind(beta.dglm, beta.sns.vd)

     beta.dglm beta.sns.vd
X1 -0.32106471 -0.32106971
X2  0.08583698  0.08583901
X3 -0.47687822 -0.47687481
X4 -0.12768455 -0.12768905
X5 -0.21386222 -0.21385871

R> cbind(gamma.dglm, gamma.sns.vd)

    gamma.dglm gamma.sns.vd
X1  0.36512842   0.36516871
X2 -0.17765127  -0.17769078
X3  0.04329333   0.04320735
X4  0.80316800   0.80323558
X5 -0.44307133  -0.44307854
```

Note that the mean coefficients from **dglm** and **RegressionFactory** match exactly in constant-dispersion case, but the dispersion parameters do not match since `dglm` uses a method of moments to estimate dispersion, rather than log-likelihood maximization. For varying-dispersion scenario, since mean and dispersion coefficients are estimated simultaneously, neither sets match exactly between the two methods, but they are very close, and the discrepancy becomes smaller for larger data.

4.3. Example 3: Geometric regression

In the last example, we illustrate how a new GLM regression can be easily constructed using the **RegressionFactory** framework. This involves three steps: 1) identify a base distribution, 2) select the link function(s), and 3) combine 1 and 2 to arrive at the log-likelihood function and its derivatives, preferrably to make the Hessian negative-definite. According to Theorem 1, this property can be proven in the base-distribution space, which is often quite easy. Consider the geometric distribution:

$$P(y = k; p) = (1 - p)^{k-1} p. \tag{22}$$

Using a logit link function for p, we arrive at the following log-likelihood:

$$f(u; y) = -\left(y\, u + (1 + y) \log\left(1 + e^{-u}\right)\right). \tag{23}$$

Concavity of the above function can be easily verified:

$$f_{uu} = -(1 + y)e^u/(1 + e^u)^2 < 0 \tag{24}$$

The base function `fbase1.goemetric.logit` implements the above log-likelihood and its first two derivatives. To test the function, we first simulate data from the distribution:

```
R> N <- 1000
R> K <- 5
R> X <- matrix(runif(N*K, min=-0.5, max=+0.5), ncol=K)
R> beta <- runif(K, min=-0.5, max=+0.5)
R> y <- rgeom(N, prob = 1/(1+exp(-X%*%beta)))
```

We now use SNS in non-stochastic mode (i.e. Newton optimization) to estimate the coefficients. We begin by our usual thin wrapper around the expander function to fully implement the log-likelihood.

```
R> loglike.geometric <- function(beta, X, y, fgh) {
+    regfac.expand.1par(beta, X, y, fbase1.geometric.logit, fgh)
+  }
R> beta.est <- rep(0,K)
R> for (n in 1:10) {
+    beta.est <- sns(beta.est, fghEval=loglike.geometric
+      , X=X, y=y, fgh=2, rnd = F)
+  }
R> cbind(beta, beta.est)
```

```
             beta     beta.est
[1,]  0.09753321  0.003590316
[2,] -0.27663711 -0.357902816
[3,] -0.23133262 -0.073516246
[4,]  0.14400388  0.128459312
[5,]  0.04067138  0.129075056
```

5. Summary

We presented R package **RegressionFactory**, a modular framework for evaluating GLM log-likelihood functions and their derivatives. We illustrated its utility in rapidly developing composite GLM models such as Hierarchical Bayesian as well as new regression models such as geometric and exponential regression. The accompanying definiteness-invariance theorem allows us to reason about logl-likelihood Hessian in a much lower-dimensional space.

Another advantage of our modular implementation is that it allows for performance optimization strategies to be readily applied across all GLM models. For example, the linear algebra steps contained in the expansion functions `regfac.expand.1par` and `regfac.expand.2par` can be thoroughly studied from the following perspectives:

- Row-major vs. column-major layout of the covariate matrices $\mathbf{X}^j$'s, for single-threaded and multi-threaded scenarios.
- Non-Uniform Memory Access (NUMA) implications of memory allocation for $\mathbf{X}^j$'s.
- Loop and cache fusion strategies.
- Coarse- vs. fine-grained parallelization in composite models such as HB.

While base functions contain model-specific code, yet they also present broad optimization opportunities. For example, they are all vectorized by definition, suggesting that they can benefit from optimized Single-Instruction, Multiple-Data (SIMD) implementation. In particular, access to vectorized transcendental functions can greatly improve the peformance of many base functions. Many of the above issues have been studied here (Mahani and Sharabiani 2013). A natural next step for **RegressionFactory** would be to implement the expander and base functions in compiled code such as C/C++, which allows for many of the advanced optimization techniques to be applicable subsequently.

References

Apostol TM (1974). *Mathematical Analysis*. Addison Wesley Publishing Company.

Boyd S, Vandenberghe L (2009). *Convex optimization*. Cambridge university press.

Dunn PK, Smyth GK (2014). *dglm: Double Generalized Linear Models*. R package version 1.8.1, URL http://CRAN.R-project.org/package=dglm.

Dunteman GH, Ho MHR (2006). *An Introduction to Generalized Linear Models*. 145. Sage.

Gelman A, Hill J (2006). *Data analysis using regression and multilevel/hierarchical models*. Cambridge University Press.

Gentle JE (2007). *Matrix Algebra: Theory, Computations, and Applications in Statistics (Springer Texts in Statistics)*. Springer-Verlag.

Gilbert GT (1991). "Positive definite matrices and Sylvester's criterion." *American Mathematical Monthly*, pp. 44–46.

Gilks WR, Wild P (1992). "Adaptive rejection sampling for Gibbs sampling." *Applied Statistics*, pp. 337–348.

Girolami M, Calderhead B (2011). "Riemann manifold langevin and hamiltonian monte carlo methods." *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, **73**(2), 123–214.

Mahani AS, Hasan A, Jiang M, Sharabiani MT (2014). *sns: Stochastic Newton Sampler (SNS)*. R package version 0.9.1, URL http://CRAN.R-project.org/package=sns.

Mahani AS, Sharabiani MT (2013). "SIMD Parallel MCMC Sampling with Applications for Big-Data Bayesian Analytics." *arXiv preprint arXiv:1310.1537*.

McCullagh P, Nelder JA (1989). "Generalized Linear Models."

Neal R (2011). "MCMC using Hamiltonian dynamics." *Handbook of Markov Chain Monte Carlo*, **2**.

Press WH (2007). *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press.

Qi Y, Minka TP (2002). "Hessian-based markov chain monte-carlo algorithms."

Smyth GK (1989). "Generalized Linear Models with Varying Dispersion." *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 47–60.

Wright S, Nocedal J (1999). *Numerical optimization*, volume 2. Springer New York.

Affiliation:

Alireza S. Mahani
Scientific Computing Group
Sentrana Inc.
1725 I St NW
Washington, DC 20006
E-mail: alireza.mahani@sentrana.com