lbfgs: Efficient L-BFGS and OWL-QN Optimization in R

Antonio Coppola  
Stanford University

Brandon M. Stewart  
Princeton University

Abstract

This vignette introduces the lbfgs package for R, which consists of a wrapper built around the libLBFGS optimization library written by Naoaki Okazaki. The lbfgs package implements both the Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) and the Orthant-Wise Limited-memory Quasi-Newton (OWL-QN) optimization algorithms. The L-BFGS algorithm solves the problem of minimizing an objective, given its gradient, by iteratively computing approximations of the inverse Hessian matrix. The OWL-QN algorithm finds the optimum of an objective plus the $L_1$ norm of the problem’s parameters. The package offers a fast and memory-efficient implementation of these optimization routines, which is particularly suited for high-dimensional problems. The lbfgs package compares favorably with other optimization packages for R in microbenchmark tests.

Keywords: optimization, optim, L-BFGS, OWL-QN, R.

1. Introduction

In this vignette we demonstrate how to use the lbfgs R package. While the optim function in the R core package stats provides a variety of general purpose optimization algorithms for differentiable objectives, there is no comparable general optimization routine for objectives with a non-differentiable penalty. Non-differentiable penalties such as the $L_1$ norm are attractive because they promote sparse solutions (Hastie et al. 2009). However it is this same lack of smoothness which makes the quasi-Newton methods in optim inapplicable.

The lbfgs package addresses this issue by providing access to the Orthant-Wise Limited-memory Quasi-Newton (OWL-QN) optimization algorithm of Andrew and Gao (2007), which allows for optimization of an objective with an $L_1$ penalty. The package uses the libLBFGS C++ library by Okazaki (2010), which itself is a port of the Fortran implementation by Nocedal (1980). In addition to OWL-QN the package provides an implementation of L-BFGS which complements optim. The linkage between R and C++ is achieved using Rcpp (Eddelbuettel 2013b).

The package provides general purpose access to these two optimization algorithms which are suitable for large-scale applications with high-dimensional parameters. The objective and gradient can be programmed in R or directly in C++ for high performance.

In Section 2 we provide a brief summary of the L-BFGS and OWL-QN algorithms. In Section 3

---

1We are extremely thankful to Dustin Tingley for his advice and guidance. We also thank Jennifer Shephard and Harvard’s Behavioral Laboratory in the Social Sciences for providing support for this project.
we proceed to describe the features of the package using applied examples with functions coded in R. In Section 4 we demonstrate how to achieve higher performance by coding objective and gradient functions in C++. Section 5 concludes.

2. Background

2.1. Notation

Throughout this vignette, we adopt notation from Andrew and Gao (2007). Let $f : \mathbb{R}^n \rightarrow \mathbb{R}$ be the objective function to be minimized. We also let the $||\cdot||$ operator denote the $L_2$ norm of a vector, and $||\cdot||_1$ denote the $L_1$ norm. $\mathbf{B}_k$ is the Hessian matrix (or its approximation) of $f$ at $\mathbf{x}^k$, and $\mathbf{g}^k$ if the gradient of $f$ at the same point. We also let $\mathbf{H}_k = \mathbf{B}_k^{-1}$ be the inverse of the (approximated) Hessian matrix.

2.2. The L-BFGS Algorithm

The Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) algorithm (Liu and Nocedal 1989) is employed for solving high-dimensional minimization problems in scenarios where both the objective function and its gradient can be computed analytically. The L-BFGS algorithm belongs to the class of quasi-Newton optimization routines, which solve the given minimization problem by computing approximations to the Hessian matrix of the objective function. At each iteration, quasi-Newton algorithms locally model $f$ at the point $\mathbf{x}^k$ using a quadratic approximation:

$$Q(\mathbf{x}) = f(\mathbf{x}^k) + (\mathbf{x} - \mathbf{x}^k)^T \mathbf{g}^k + \frac{1}{2}(\mathbf{x} - \mathbf{x}^k)^T \mathbf{B}_k(\mathbf{x} - \mathbf{x}^k)$$

A search direction can then be found by computing the vector $\mathbf{x}^*$ that minimizes $Q(\mathbf{x})$. Assuming that the Hessian is positive-definite, this is $\mathbf{x}^* = \mathbf{x}^k - \mathbf{H}_k \mathbf{g}^k$. The next search point is then found along the ray defined by $\mathbf{x}^k - \alpha \mathbf{H}_k \mathbf{g}^k$. The procedure is iterated until the gradient is zero, with some degree of convergence tolerance.

In high dimensional settings even storing the Hessian matrix can be prohibitively expensive. The L-BFGS algorithm avoids storing the sequential approximations of the Hessian matrix which allows it to generalize well to the high-dimensional setting. Instead, L-BFGS stores curvature information from the last $m$ iterations of the algorithm, and uses them to find the new search direction. More specifically, the algorithm stores information about the spatial displacement and the change in gradient, and uses them to estimate a search direction without storing or computing the Hessian explicitly. We refer interested readers to Nocedal and Wright (2006) for additional details.

2.3. The OWL-QN Algorithm

The L-BFGS method cannot be applied to problems with an objective function of the form $r(\mathbf{x}) = C \cdot ||\mathbf{x}||_1 = C \cdot \sum|x_i|$, such as LASSO regression or $L_1$-penalized log-linear models, given the non-differentiability of the objective function at any point where at least one of the parameters is zero. The OWL-QN algorithm developed by Andrew and Gao (2007), modifies the L-BFGS algorithm to allow for $L_1$ penalties.
The algorithm exploits the fact that $L_1$-regularized objective functions will still be differentiable in any given orthant of the function space. At each iteration, the algorithm chooses an orthant within which to evaluate the function by estimating the sign of each of its parameters. The algorithm then constructs a quadratic approximation to the function in the given orthant using a regular L-BFGS procedure, and searches in the direction of the minimum of the approximation within the same orthant. For further details regarding OWL-QN, we refer the interested reader to the original article by Andrew and Gao (2007).

3. The lbfgs package

The lbfgs package provides a general-purpose library for numerical optimization with L-BFGS and OWL-QN. As such, its syntax and usage closely mirror those of other popular packages for numerical optimization in R. While there are many alternatives for smooth unconstrained optimization, most optimization methods including an $L_1$ penalty are limited to end-user regression functions rather than general optimization frameworks. These functions can be more efficient than lbfgs for the particular problems they solve, but they do not allow easy extension or modification.

The following list provides brief comparisons between lbfgs and several other packages:

- **optim**: The lbfgs package can be used as a drop-in replacement for the L-BFGS-B method of the optim (R Development Core Team 2008) and optimx (Nash and Varadhan 2011), with performance improvements on particular classes of problems, especially if lbfgs is used in conjunction with C++ implementations of the objective and gradient functions. In addition, the possibility of introducing $L_1$ penalization of the objective function allows for solution vectors with much higher sparsity, as most of the otherwise near-zero parameters are driven to zero.

- **penalized**: The penalized package (Goeman et al. 2012) fits generalized linear models with both $L_1$ (lasso and fused lasso) and $L_2$ (ridge) penalizations. However, penalized does not permit general optimization of $L_1$ regularized functions.

- **glmnet**: The glmnet package (Friedman et al. 2010) fits a regularization path for lasso and elastic-net generalized linear models using extremely fast cyclical coordinate descent algorithms coded in Fortran. As in the previous case, however, glmnet cannot perform general-purpose optimization.

We also note that the mlegp package also makes use of the libLBFGS library but does not provide general purpose access to the optimization functions (Dancik 2013).

3.1. Package API

All optimization routines are handled by the lbfgs() function. The accompanying manual provides detailed and exhaustive documentation regarding the package API, which is closely modeled after Okazaki (2010). Here we present some of the details that are likely to be of greatest interest to a user who is familiarizing herself with the package.

---

2See for example the Optimization Taskview
• **Basic Inputs:** The objective and gradient functions should be supplied as either R functions or external pointers to C++ functions compiled using the inline interface (see Section 4). Extra parameters can be passed to the objective and gradient functions using either the ... construct or by encapsulating them in an R environment, which is then passed to lbfgs(). If the functions are implemented in R, the ... construct should be used. If the functions are otherwise implemented in C++, users should resort to the environment parameter.

• **Hessian Approximation:** As mentioned in Section 2, the L-BFGS algorithm stores the results of its previous m iterations to approximate the inverse Hessian matrix for the current iteration. The parameter m specifies the number of previous computations to be stored. The default value is 6, and values less than 3 are not recommended (Okazaki 2010). Note also that `optim`'s equivalent parameter, lmm, defaults to 5 instead.

• **Line Search Strategies:** The libLBFGS library implements a number of line search strategies for use with the L-BFGS and OWL-QN methods. The default strategy for the L-BFGS method is the one described by More and Thuente (1994). The More-Thuente strategy uses quadratic and cubic interpolations to find a step length that satisfies the Wolfe conditions (Wolfe 1969). The OWL-QN method does not support the More-Thuente strategy, and instead employs a backtracking strategy by default. Given a starting point, the algorithm backtracks toward zero until a certain set of conditions is met. On top of the regular Wolfe conditions, the Armijo curvature rule (Armijo 1966) and the strong Wolfe conditions (Nocedal and Wright 2006) are supported. We refer the readers to the referenced literature for background regarding the Wolfe and Armijo conditions.

• **L_1 Regularizations for OWL-QN:** The OWL-QN method is invoked by specifying a nonzero coefficient C for the L_1 norm of the parameters of the objective function. Given an objective function f(\vec{x}), the OWL-QN algorithm will minimize the L_1-penalized version of the function: f(\vec{x}) + C \cdot ||\vec{x}||_1. Note that several other packages built for handling L_1-penalized models (for instance, glmnet) minimize a regularized objective of the following form: \frac{f(\vec{x})}{N} + C \cdot ||\vec{x}||_1, where N is the number of variables in the parameter vector. Users should be aware that `lbfgs` does not automatically perform any regularization of this kind.

### 3.2. Simple test functions

We begin by using `lbfgs` to minimize a suite of simple test functions, and benchmarking the package against the L-BFGS-B `optim` method.

• **The Rosenbrock function:** We define the Rosenbrock function (Rosenbrock 1960) mapping \( \mathbb{R}^2 \) to \( \mathbb{R} \) as \( f(x, y) = 100 \cdot (y - x^2)^2 + (1 - x)^2 \). The function has a global minimum at \((0, 0)\) that lies within a long, flat valley. We define the function and its gradient as R objects, and then run the optimization routine using both `lbfgs` and `optim`. Note that the functions must accept all variables as a single numeric vector:

```r
> objective <- function(x) {
  100 * (x[2] - x[1]^2)^2 + (1 - x[1])^2
```
\[
\text{gradient} \leftarrow \text{function}(x) \{
\begin{align*}
&-400 \cdot x[1] \cdot (x[2] - x[1]^2) - 2 \cdot (1 - x[1]), \\
&200 \cdot (x[2] - x[1]^2)
\end{align*}
\}
\]

\[
\text{out.lbfgs} \leftarrow \text{lbfgs(}\text{objective, gradient, c(-1.2, 1)}\text{)}
\]

\[
\text{out.optim} \leftarrow \text{optim(}c(-1.2, 1), \text{objective, gradient, method=}"L-BFGS-B""\text{)}
\]

The results are the following:

\[
\text{out.lbfgs}$\text{value}$
\]

[1] 3.545445e-13

\[
\text{out.lbfgs}$\text{par}$
\]

[1] 1.000001 1.000001

\[
\text{out.optim}$\text{value}$
\]

[1] 2.267577e-13

\[
\text{out.optim}$\text{par}$
\]

[1] 0.9999997 0.9999995

The results are essentially the same, but \texttt{lbfgs} achieves better running speeds in a microbenchmark \textsuperscript{3} test done using the \texttt{microbenchmark} package (Mersmann 2013):

\[
\text{microbenchmark(}\text{out.lbfgs} \leftarrow \text{lbfgs(}\text{objective, gradient, c(-1.2, 1), invisible=1}\text{)}, \text{out.optim} \leftarrow \text{optim(}c(-1.2, 1), \text{objective, gradient, method=}"L-BFGS-B"\text{)})
\]

\[\begin{array}{llllllll}
\text{expr} & \text{min} & \text{lq} & \text{median} & \text{uq} & \text{max} & \text{neval} \\
\text{out.lbfgs} \leftarrow \text{lbfgs(} \ldots \text{)} & 288.673 & 298.2110 & 303.8770 & 320.326 & 561.389 & 100 \\
\text{out.optim} \leftarrow \text{optim(} \ldots \text{)} & 389.958 & 402.1195 & 411.6735 & 432.590 & 691.975 & 100 \\
\end{array}\]

- **Booth’s function**: Booth’s function is \(f(x, y) = (x + 2y - 7)^2 + (2x + y - 5)^2\). The function has a global minimum at \((1, 3)\). The code and microbenchmark test are as follows:

\[
\text{objective} \leftarrow \text{function}(x)\{
\begin{align*}
\end{align*}
\}
\]

\[
\text{gradient} \leftarrow \text{function}(x)\{
\}
\]

\textsuperscript{3}All microbenchmark test were performed on a machine running OS X, Version 10.9.3, with a 2.9 GHz Intel Core i7 processor, and 8 GB 1600 MHz DDR3 memory.
3.3. A Logistic Regression with L-BFGS

In the following example we use \texttt{lbfgs} on the \texttt{Leukemia} example in Friedman et al. (2010), with data originally due to Golub et al. (1999). As in Friedman et al. (2010) we use logistic regression for the purposes of cancer classification based on gene expression monitoring in a microarray study. The dataset contains both a vector \( \tilde{y} \) of binary values specifying the cancer class for 72 leukemia patients, and a 72 \( \times \) 3571 matrix \( X \) specifying the levels of expressions of 3571 genes for the 72 different patients. In the vector \( \tilde{y} \), 0 corresponds to patients with acute lymphoblastic leukemia (ALL), and 1 to patients with acute myeloid leukemia (AML). First, we load the data to the \texttt{R} workspace:

\begin{verbatim}
> data(Leukemia)
> X <- Leukemia$x
> y <- Leukemia$y
\end{verbatim}

The likelihood function and its gradient for the standard logit setup with a ridge penalty are specified as follows:

\begin{verbatim}
> likelihood <- function(par, X, y, prec) {
  Xbeta <- X %*% par
  -(sum(y * Xbeta - log(1 + exp(Xbeta))) - .5 * sum(par^2*prec))
}

> gradient <- function(par, X, y, prec) {
  p <- 1/(1 + exp(-X %*% par))
  -(crossprod(X,(y - p)) - par * prec)
}
\end{verbatim}

We bind a constant term to the \( X \) matrix, and define a numerical vector with origin parameters for the algorithm initialization:

\begin{verbatim}
> X1 <- cbind(1, X)
> init <- rep(0, ncol(X1))
\end{verbatim}

Then we use both \texttt{lbfgs} and \texttt{optim} to run the regression with a penalty coefficient of 2:
> lbfgs.out <- lbfgs(likelihood, gradient, init, invisible=1, 
X=X1, y=y, prec=2)
> optim.out <- optim(init, likelihood, gradient, method = "L-BFGS-B", 
X=X1, y=y, prec=2)

> all.equal(optim.out$value, lbfgs.out$value)
[1] TRUE

In this particular case, **optim** outperforms **lbfgs**, but it is to be noted that this problem has a high number of parameters and a low number of observations:

<table>
<thead>
<tr>
<th>expr</th>
<th>min</th>
<th>lq</th>
<th>median</th>
<th>uq</th>
<th>max</th>
<th>neval</th>
</tr>
</thead>
<tbody>
<tr>
<td>optim.out &lt;- optim(...)</td>
<td>84.57455</td>
<td>99.03664</td>
<td>102.7622</td>
<td>119.6641</td>
<td>189.1403</td>
<td>100</td>
</tr>
<tr>
<td>lbfgs.out &lt;- lbfgs(...)</td>
<td>121.46801</td>
<td>138.16293</td>
<td>150.5174</td>
<td>192.3550</td>
<td>234.5430</td>
<td>100</td>
</tr>
</tbody>
</table>

Although both **optim** and **lbfgs** are using the same algorithm here there are subtle differences in the implementation. This underscores the importance of benchmarking performance for the individual application of interest.

### 3.4. A Poisson Regression with OWL-QN

Next, we use the OWL-QN method in **lbfgs** to perform a $L_1$ regularized Poisson regression comparing performance to **glmnet**. We emphasize that this could not be done directly with **optim** due to the presence of the $L_1$ penalty. We set up a simulated dataset the simple data generating process given in the manual of **glmnet** ([Friedman et al. 2010](#)). First, we define the variables of interest:

> N <- 500
> p <- 20
> nzc <- 5
> x <- matrix(rnorm(N * p), N, p)
> beta <- rnorm(nzc)
> f <- x[, seq(nzc)] %*% beta
> mu <- exp(f)
> y <- rpois(N, mu)
> X1 <- cbind(1, x)
> init <- rep(0, ncol(X1))

We can perform a Poisson regression on this simulated data using **glmnet**:

> fit <- glmnet(x, y, family="poisson", standardize=FALSE)

We choose a value of the regularization parameter from the model fitted with **glmnet** as the OWL-QN penalty coefficient to obtain analogous results using **lbfgs**:

> C <- fit$lambda[25]*nrow(x)

To perform the same regression with **lbfgs**, we define the model’s likelihood function and its gradient in R:
> likelihood <- function(par, X, y, prec=0) {
  Xbeta <- X %*% par
  -(sum(y * Xbeta - exp(Xbeta)) - .5 * sum(par^2*prec))
}

> gradient <- function(par, X, y, prec=0) {
  Xbeta <- X %*% par
  -(crossprod(X, (y - exp(Xbeta))) - par * prec)
}

Hence we make a call to \texttt{lbfgs}:

\[
\text{out} \leftarrow \text{lbfgs}(\text{likelihood}, \text{gradient}, \text{init}, X=X_1, y=y, \text{prec}=0, \\
\tag{4.1.1}
\text{invisible}=1, \text{orthantwise}_c=C, \\
\text{linesearch}_\text{algorithm}=\text{"LBFGS LINESEARCH BACKTRACKING"}, \\
\text{orthantwise}\_\text{start} = 1, \\
\text{orthantwise}\_\text{end} = \text{ncol}(X_1))
\]

The microbenchmark test yields:

\[
\begin{array}{llllllll}
\text{Unit: milliseconds} & \text{expr} & \text{min} & \text{lq} & \text{median} & \text{uq} & \text{max} & \text{neval} \\
\text{out} \leftarrow \text{lbfgs}(\ldots) & 2.340520 & 2.441575 & 2.544409 & 2.952162 & 10.47467 & 100 \\
\text{fit} \leftarrow \text{glmnet}(\ldots) & 9.959642 & 10.343768 & 10.694795 & 12.425912 & 18.21408 & 100
\end{array}
\]

The \texttt{lbfgs} solution is a little over 4 times as fast. We emphasize that this is not strictly a fair comparison. \texttt{glmnet} is calculating the entire regularization path and thus is solving 100 problems of the type. Indeed using OWL-QN to calculate all 100 problems might take hundreds of times longer than \texttt{glmnet}. However, as noted in Friedman et al. (2010), \texttt{glmnet}'s reliance on warm starts means that there is no straightforward method for optimizing with a single value of the regularization parameter. In GLMs it is often desirable to have the regularization path but in iterative algorithms the additional computation may be unnecessary.

For straightforward GLMs it would be difficult to find a solution faster than \texttt{glmnet}'s. \texttt{lbfgs} provides the additional flexibility of allowing user-defined functions and can provide significantly faster optimization at a single value of the regularization parameter.

## 4. Faster Performance: Objective and Gradient in C++

### 4.1. The Basics

The package supports the implementation of the objective and gradient functions in C++, which may yield significant speed improvements over the respective R implementations. The optimization routine’s API accepts both R function objects and external pointers to compiled C++ functions. The C++ evaluation routines use code from Ardia et al. (2013) and
In order to be compatible with the `lbfgs` API, the C++ functions must return an object of type `Rcpp::NumericVector`, and take in either one or two objects of type `SEXP`. The first argument of type `SEXP` must be the pointer to an R numerical vector containing the values of the function’s parameters. The second (optional) argument must be the pointer to an R environment holding all extra parameters to be fed into the objective and gradient functions. To perform optimization on the Rosenbrock function, we begin by defining the C++ implementations of the objective and of the gradient as character strings, using the `Rcpp` library:

```r
> objective.include <- '\texttt{Rcpp::NumericVector rosenbrock(SEXP xs) \{\}
Rcpp::NumericVector x(xs);
  double x1 = x[0];
  double x2 = x[1];
  double sum = 100 * (x2 - x1 * x1) * (x2 - x1 * x1) + (1 - x1) * (1 - x1);
Rcpp::NumericVector out(1);
  out[0] = sum;
  return(out);\}
',

> gradient.include <- '\texttt{Rcpp::NumericVector rosengrad(SEXP xs) \{\}
Rcpp::NumericVector x(xs);
  double x1 = x[0];
  double x2 = x[1];
  double g1 = -400 * x1 * (x2 - x1 * x1) - 2 * (1 - x1);
  double g2 = 200 * (x2 - x1 * x1);
Rcpp::NumericVector out(2);
  out[0] = g1;
  out[1] = g2;
  return(out);\}
',
```

Then we assign two character strings with the bodies of two functions to generate external pointers to the objective and the gradient:

```r
> objective.body <- '\texttt{\}
  typedef Rcpp::NumericVector (*funcPtr)(SEXP);
  return(XPtr<funcPtr>(new funcPtr(&rosenbrock)));
',

> gradient.body <- '\texttt{\}
  typedef Rcpp::NumericVector (*funcPtr)(SEXP);
  return(XPtr<funcPtr>(new funcPtr(&rosengrad)));
',
```

The idea of supplying external pointers to the underlying C++ library was introduced by Dirk Eddelbuettel. See for instance the slides from his presentation at the University of Kansas in November 2013.
Finally, we compile this ensemble using the `inline` package by Sklyar et al. (2013):

```r
> objective <- cxxfunction(signature(), body=objective.body,
  inc=objective.include, plugin="Rcpp")

> gradient <- cxxfunction(signature(), body=gradient.body,
  inc=gradient.include, plugin="Rcpp")
```

The external pointers to the objective and the gradient generated by the two pointer-assigners can then be supplied to the `lbfgs` routine:

```r
> out.CPP <- lbfgs(objective(), gradient(), c(-1.2,1), invisible=1)
```

We define the same functions in R for comparison purposes:

```r
> objective.R <- function(x) {
  100 * (x[2] - x[1]^2)^2 + (1 - x[1])^2
}

> gradient.R <- function(x) {
}
```

A microbenchmark comparison reveals significant speed improvements:

```r
> microbenchmark(out.CPP <- lbfgs(objective(), gradient(), c(-1.2,1),
                   invisible=1),
       out.R <- lbfgs(objective.R, gradient.R, c(-1.2,1),
                   invisible=1))
```

The results are the following, including also the `optim` routine as a benchmark (`neval=100` for all runs):

```
Unit: microseconds 
expr   min lq median uq  max
lbfgs(objective(), ...) 79.430 85.1265 90.6680 97.7615 269.533
lbfgs(objective.R, ...) 260.552 272.8125 292.2045 312.5050 561.668
optim(...) 368.788 384.6445 412.3530 448.1345 1719.914
```

### 4.2. Extra Parameters in C++ Implementations

Much like in the R case, the passing of extra parameters with C++ implementations of the objective and gradient is achieved through the use of R environments. The following is an example replicating the logistic regression example with C++ function implementations. As before, we set up the objective and gradient as character strings. We include the extra environment argument and obtain data by evaluating it using the `Rcpp::Environment` class.

In order to perform matrix operations, we use the `RcppArmadillo` library (Eddelbuettel and Sanderson 2014):
Figure 1: The violin plots depict the distribution of running times obtained from optimizing the Rosenbrock function in the course a microbenchmark test. From top to bottom, the three experimental conditions were the following: `optim()` (L-BFGS-B method) with R-coded inputs; `lbfgs()` with R-coded inputs; and `lbfgs()` with inputs in C++.
> likelihood.include <- 'Rcpp::NumericVector lhood(SEXP xs, SEXP env){
arma::vec par = Rcpp::as<arma::vec>(xs);
Rcpp::Environment e = Rcpp::as<Rcpp::Environment>(env);
arma::mat X = Rcpp::as<arma::mat>(e["X"]);
arma::vec y = Rcpp::as<arma::vec>(e["y"]);
double prec = Rcpp::as<double>(e["prec"]);
arma::mat Xbeta = X * par;
double sum1 = sum(y % Xbeta - log(1 + exp(Xbeta)));
arma::mat sum2 = sum(pow(par, 2 * prec));
arma::vec out = -(sum1 - 0.5 * sum2);
Rcpp::NumericVector ret = Rcpp::as<Rcpp::NumericVector>(wrap(out));
return ret;
}

> gradient.include <- 'Rcpp::NumericVector grad(SEXP xs, SEXP env){
arma::vec par = Rcpp::as<arma::vec>(xs);
Rcpp::Environment e = Rcpp::as<Rcpp::Environment>(env);
arma::mat X = Rcpp::as<arma::mat>(e["X"]);
arma::vec y = Rcpp::as<arma::vec>(e["y"]);
double prec = Rcpp::as<double>(e["prec"]);
arma::vec p = 1 / (1 + exp(-(X * par)));
arma::vec grad = -((trans(X) * (y - p)) - par * prec);
Rcpp::NumericVector ret = Rcpp::as<Rcpp::NumericVector>(wrap(grad));
return ret;
}

Then we compile the functions and their pointer-assigners, taking care to map the functions’ signatures correctly:

> likelihood.body <- '
  typedef Rcpp::NumericVector (*funcPtr)(SEXP, SEXP);
  return(XPtr<funcPtr>(new funcPtr(&lhood)));'

> gradient.body <- '
  typedef Rcpp::NumericVector (*funcPtr)(SEXP, SEXP);
  return(XPtr<funcPtr>(new funcPtr(&grad)));'

> likelihood.CPP <- cxxfunction(signature(), body=likelihood.body,
  inc=likelihood.include, plugin="RcppArmadillo")

> gradient.CPP <- cxxfunction(signature(), body=gradient.body,
  inc=gradient.include, plugin="RcppArmadillo")
We then instantiate a new R environment with the required objects, and run the optimization routine:

```r
> data(Leukemia)
> X <- Leukemia$x
> y <- Leukemia$y
> X1 <- cbind(1, X)
> init <- rep(0, ncol(X1))

> env <- new.env()
> env[["X"]]<- X1
> env[["y"]]<- y
> env[["prec"]]<- 1

> output <- lbfgs(likelihood.CPP(), gradient.CPP(), init, environment=env)
```

A final microbenchmark test reveals performance improvements over the corresponding R implementation (neval = 100 for all runs):

```r
> microbenchmark(out.CPP <- lbfgs(likelihood.CPP(), gradient.CPP(), invisible=1, init, environment=env), out.R <- lbfgs(likelihood, gradient, init, invisible=1, X=X1, y=y, prec=1))
```

Unit: milliseconds

<table>
<thead>
<tr>
<th>expr</th>
<th>min</th>
<th>lq</th>
<th>median</th>
<th>uq</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>lbfgs(likelihood.CPP(), ...)</td>
<td>121.2342</td>
<td>130.6826</td>
<td>135.0989</td>
<td>140.2065</td>
<td>322.2660</td>
</tr>
<tr>
<td>lbfgs(likelihood, ...)</td>
<td>126.5353</td>
<td>142.9137</td>
<td>150.2248</td>
<td>156.0659</td>
<td>397.5917</td>
</tr>
</tbody>
</table>

While the speed of the optimization routines is satisfying with R implementations of the objective and gradient functions, the same interface can be used in tandem with C++ when high performance is important.

5. Conclusion

The lbfgs package provides a generic R interface for performing numerical optimization using the L-BFGS and OWL-QN algorithms. This vignette provides an overview of the package’s features and usage. More detailed documentation regarding the package’s functionality and API are available in the accompanying manual.
References


citation.cfm?id=192132.

jstatsoft.org/v43/i09/.


com/id/10228772.

Okazaki N (2010). libLBFGS: A Library of Limited-memory Broyden-Fletcher-Goldfarb-


Rosenbrock HH (1960). “An automatic method for finding the greatest or least value of a

C++, Fortran Function Calls from R. URL http://cran.r-project.org/web/packages/
inline/index.html.


Affiliation:
Antonio Coppola
Stanford Institute for Economic Policy Research
Stanford University
366 Galvez Street, Stanford, CA, USA
E-mail: acoppola@stanford.edu

Brandon M. Stewart
Department of Sociology
Princeton University
149 Wallace Hall, Princeton, NJ, USA
E-mail: bms4@princeton.edu
URL: https://scholar.princeton.edu/bstewart/home