Fast Penalized Regression and Cross Validation for Tall Data with the oem Package

Jared D. Huling
University of Minnesota, Twin Cities

Peter Chien
University of Wisconsin-Madison

Abstract

A large body of research has focused on theory and computation for variable selection techniques for high dimensional data. There has been substantially less work in the big “tall” data paradigm, where the number of variables may be large, but the number of observations is much larger. The orthogonalizing expectation maximization (OEM) algorithm is one approach for computation of penalized models which excels in the big tall data regime. The oem package is an efficient implementation of the OEM algorithm which provides a multitude of computation routines with a focus on big tall data, such as a function for out-of-memory computation, for large-scale parallel computation of penalized regression models. Furthermore, in this paper we propose a specialized implementation of the OEM algorithm for cross validation, dramatically reducing the computing time for cross validation over a naive implementation.

Keywords: lasso, MCP, optimization, expectation maximization, C++, OpenMP, parallel computing, out-of-memory computing.

1. Introduction

Penalized regression has been a widely used technique for variable selection for many decades. Computation for such models has been a challenge due to the non-smooth properties of penalties such as the lasso (Tibshirani 1996). A plethora of algorithms now exist such as the least angle regression (LARS) algorithm (Efron, Hastie, Johnstone, and Tibshirani 2004) and coordinate descent (Tseng 2001; Friedman, Hastie, and Tibshirani 2010), among many others. There exist for these algorithms even more R (R Core Team 2022) packages for various penalized regression routines such as glmnet (Friedman, Hastie, Simon, and Tibshirani 2022), lars (Hastie and Efron 2022), ncvreg (Breheny 2021), grrp (Breheny and Zeng 2021), and gglasso (Yang, Zou, and Bhatnagar 2020), among countless others. Each of the above packages
focuses on a narrow class of penalties, such as group regularization, non-convex penalties, or the lasso. The existence of many options often makes it hard to choose between packages, and furthermore difficult to develop a consistent workflow when it is not clear which type of penalty is suitable for a given problem. There has been much focus on algorithms for scenarios where the number of variables $p$ is much larger than the number of observations $n$, such as the LARS algorithm, yet many applications of “internet scale” typically involve an extraordinarily large number of observations and a moderate number of variables. In these applications, speed is crucial. The **oem** package (Dai and Huling 2022) is intended to provide a highly efficient framework for penalized regression in these tall data settings. It provides computation for a comprehensive selection of penalties, including the lasso and elastic net, group-wise penalties, and non-convex penalties and allows for simultaneous computation of these penalties. Most of the algorithms and packages listed above, however, are more efficient than the **oem** package for scenarios when the number of variables is larger than the number of observations. Roughly speaking, the **oem** package is most effective when the ratio of the number of variables to the number of observations is less than $1/10$. This is ideal for data settings, such as in internet applications, where a large number of variables are available, but the number of observations grows rapidly over time.

Centered around the orthogonalizing expectation maximization (OEM) algorithm of Xiong, Dai, Huling, and Qian (2016), the **oem** package provides a unified framework for computation for penalized regression problems with a focus on big *tall* data scenarios. The OEM algorithm is particularly useful for regression scenarios when the number of observations is significantly larger than the number of variables. It is efficient even when the number of variables is large (in the thousands) as long as the number of observations is yet larger (hundreds of thousands or more). The OEM algorithm is particularly well-suited for penalized linear regression scenarios when the practitioner must choose between a large number of potential penalties, as the OEM algorithm can compute full tuning parameter paths for multiple penalties nearly as efficiently as for just one penalty.

The **oem** package also places an explicit emphasis on practical aspects of penalized regression model fitting, such as tuning parameter selection, in contrast to the vast majority of penalized regression packages. The most common approach for tuning parameter selection for penalized regression is cross validation. Cross validation is computationally demanding, yet there has been little focus on efficient implementations of cross validation. Here we present a modification of the OEM algorithm to dramatically reduce the computational load for cross validation. Additionally, the **oem** package provides some extra unique features for very large-scale problems. The **oem** package provides functionality for out-of-memory computation, allowing for fitting penalized regression models on data which are too large to fit into memory. Feasibly, one could use these routines to fit models on datasets hundreds of gigabytes in size on just a laptop. The **biglasso** package (Zeng and Breheny 2017a,b) also provides functionality for out-of-memory computation, however its emphasis is on ultrahigh-dimensional data scenarios and is limited to the lasso, elastic-net, and ridge penalties. Also provided are OEM routines based off of the quantities $X^\top X$ and $X^\top y$ that may already be available to researchers from exploratory analyses. This can be especially useful for scenarios when data are stored across a large cluster, yet the sufficient quantities can be computed easily on the cluster, making penalized regression computation very simple and quick for datasets with an arbitrarily large number of observations.
The core computation for the `oem` package is in C++ using the `Eigen` numerical linear algebra library (Guennebaud, Jacob, and others 2010) with an R interface via the `RcppEigen` (Bates and Eddelbuettel 2013) package. Out-of-memory computation capability is provided by interfacing to special C++ objects for referencing objects stored on disk using the `bigmemory` package (Kane, Emerson, and Weston 2013; Kane, Emerson, Haverty, and Determan Jr. 2022).

In Section 2, we provide a review of the OEM algorithm. In Section 3 we present a new efficient approach for cross validation based on the OEM algorithm. In Section 4 we show how the OEM algorithm can be extended to logistic regression using a proximal Newton algorithm. Section 5 provides an introduction to the package, highlighting useful features. Finally, Section 6 demonstrates the computational efficiency of the package with some numerical examples.

2. The orthogonalizing EM algorithm

2.1. Review of the OEM algorithm

The OEM algorithm is centered around the linear regression model:

\[ y = X\beta + \varepsilon, \]

where \( X = (x_{ij}) \) is an \( n \times p \) design matrix, \( y \in \mathbb{R}^n \) is a vector of responses, \( \beta = (\beta_1, \ldots, \beta_p)^\top \) is a vector of regression coefficients, and \( \varepsilon \) is a vector of random error terms with mean zero. When the number of covariates is large, researchers often want or need to perform variable selection to reduce variability or to select important covariates. A sparse estimate \( \hat{\beta} \) with some estimated components exactly zero can be obtained by minimizing a penalized least squares criterion:

\[ \hat{\beta} = \arg\min_\beta \| y - X\beta \|^2 + P_\lambda(\beta), \]

where the penalty term \( P_\lambda \) has a singularity at the zero point of its argument. Widely used examples of penalties include the lasso (Tibshirani 1996), the group lasso (Yuan and Lin 2006), the smoothly clipped absolute deviation (SCAD) penalty (Fan and Li 2001), and the minimax concave penalty (MCP; Zhang 2010), among many others.

The OEM algorithm can solve (2) under a broad class of penalties, including all of those previously mentioned. The basic motivation of the OEM algorithm is the triviality of minimizing this loss when the design matrix \( X \) is orthogonal. However, the majority of design matrices from observational data are not orthogonal. Instead, we seek to augment the design matrix with extra rows such that the augmented matrix is orthogonal. If the non-existent responses of the augmented rows are treated as missing, then we can embed our original minimization problem inside a missing data problem and use the EM algorithm. Let \( \Delta \) be a matrix of pseudo observations whose response values \( z \) are missing. If \( \Delta \) is designed such that the augmented regression matrix

\[ X_\times = \begin{pmatrix} X \\ \Delta \end{pmatrix} \]

is column orthogonal, an EM algorithm can be used to solve the augmented problem efficiently similar to Healy and Westmacott (1956). The OEM algorithm achieves this with two steps:
Step 1. Construct an augmentation matrix $\Delta$.

Step 2. Iteratively solve the orthogonal design with missing data by the EM algorithm.

Step 2.1. E-step: Impute the missing responses $z$ by $z = \Delta \beta^{(t)}$, where $\beta^{(t)}$ is the current estimate.

Step 2.2. M-step: Solve

$$
\beta^{(t+1)} = \text{argmin} \frac{1}{2} \|y - X\beta\|^2 + \frac{1}{2} \|z - \Delta \beta\|^2 + P_\lambda(\beta).
$$

An augmentation matrix $\Delta$ can be constructed using the active orthogonalization procedure. The procedure starts with any positive definite diagonal matrix $S$ and $\Delta$ can be constructed conceptually by ensuring that $\Delta^\top \Delta = dS^2 - X^\top X$ is positive semidefinite for some constant $d \geq \lambda_1(S^{-1}X^\top X S^{-1})$, where $\lambda_1(\cdot)$ is the largest eigenvalue. The term $\Delta$ need not be explicitly computed, as the EM iterations in the second step result in closed-form solutions which only depend on $\Delta^\top \Delta$. To be more specific, let $A = \Delta^\top \Delta$ and $u = X^\top y + A \beta^{(t)}$.

Furthermore, assume the regression matrix $X$ is standardized so that $\sum_{i=1}^n x_{ij}^2 = 1$, for $j = 1, \ldots, p$.

Then the update for the regression coefficients when $P_\lambda(\beta) = 0$ has the form $\beta^{(t+1)}_j = u_j/d_j$.

When $P_\lambda(\beta)$ is the $\ell_1$ norm, corresponding to the lasso penalty,

$$
\beta^{(t+1)}_j = \text{sign}(u_j) \left( \frac{|u_j| - \lambda}{d_j} \right)_+,
$$

where $(a)_+$ denotes $\max\{a, 0\}$.

Xiong et al. (2016) used a scalar value for $d$ and for simplicity chose $S$ to be the diagonal matrix. Furthermore, they showed that while the above algorithm converges for all $d \geq \lambda_1(X^\top X)$, using $d = \lambda_1(X^\top X)$ results in the fastest convergence. Such a $d$ can be computed efficiently using the Lanczos algorithm (Lanczos 1950).

2.2. Penalties

The oem package uses the OEM algorithm to solve penalized least squares problems with the penalties outlined in Table 1. For a vector $u$ of length $k$ and an index set $g \subseteq \{1, \ldots, k\}$ we define the length $|g|$ subvector $u_g$ of a vector $u$ as the elements of $u$ indexed by $g$. Furthermore, for a vector $u$ of length $k$ let $\|u\| = \sqrt{\sum_{j=1}^k u_j^2}$.

For $\lambda > 0$ let:

$$
P_{\lambda,\gamma}^{\text{SCAD}}(\beta) = \left\{ \begin{array}{ll}
\lambda|\beta| & |\beta| \leq \lambda; \\
\lambda|\beta| - \frac{\beta^2}{2\gamma} + \frac{\lambda^2}{2(\gamma-1)} & \lambda < |\beta| \leq \gamma \lambda; \\
\frac{\lambda^2}{2} & |\beta| > \gamma \lambda;
\end{array} \right.
$$

for $\gamma > 2$ and

$$
P_{\lambda,\gamma}^{\text{MCP}}(\beta) = \left\{ \begin{array}{ll}
\lambda|\beta| - \frac{\beta^2}{2\gamma} & |\beta| \leq \gamma \lambda; \\
\frac{\gamma \lambda^2}{2} & |\beta| > \gamma \lambda
\end{array} \right.
$$

for $\gamma > 1$. 

Penalty | Penalty form
--- | ---
Lasso | $\lambda \sum_{j=1}^{p} w_j |\beta_j|$
Elastic Net | $\alpha \lambda \sum_{j=1}^{p} w_j |\beta_j| + \frac{1}{2}(1-\alpha)\lambda \sum_{j=1}^{p} w_j \beta_j^2$
MCP | $\sum_{j=1}^{p} P_{\lambda w_j, \gamma}(\beta_j)$
SCAD | $\sum_{j=1}^{p} P_{\lambda, \gamma}(\beta_j)$
Group Lasso | $\lambda \sum_{k=1}^{G} c_k \|\beta_{g_k}\|$
Group MCP | $\lambda \sum_{k=1}^{G} P_{\lambda, \gamma}(\|\beta_{g_k}\|)$
Group SCAD | $\lambda \sum_{k=1}^{G} P_{\lambda, \gamma}(\|\beta_{g_k}\|)$
Sparse Group Lasso | $\lambda(1 - \tau) \sum_{k=1}^{G} c_k \|\beta_{g_k}\| + \lambda \tau \sum_{j=1}^{p} w_j |\beta_j|$

Table 1: The penalties available in the **oem** package. In the group lasso, $g_k$ refers to the index set of the $k$-th group. The vector $w \in \mathbb{R}^p$ is a set of variable-specific penalty weights and the vector $c \in \mathbb{R}^G$ is a set of group-specific penalty weights.

The updates for the above penalties are given below:

1. **Lasso**

   $$\beta_j^{(t+1)} = S(u_j, w_j \lambda, d) = \text{sign}(u_j) \left( \frac{|u_j| - w_j \lambda}{d} \right)_+.$$  

2. **Elastic Net**

   $$\beta_j^{(t+1)} = \text{sign}(u_j) \left( \frac{|u_j| - w_j \alpha \lambda}{d + w_j (1-\alpha) \lambda} \right)_+.$$  

3. **MCP**

   $$\beta_j^{(t+1)} = M(u_j, w_j \lambda, \gamma, d) = \begin{cases} 
   \text{sign}(u_j) \frac{\gamma (|u_j| - w_j \lambda)}{(\gamma d - 1)} & \text{if } |u_j| \leq w_j \gamma \lambda d, \\
   u_j / d, & \text{if } |u_j| > w_j \gamma \lambda d.
\end{cases}$$  

   where $\gamma > 1$

4. **SCAD**

   $$\beta_j^{(t+1)} = C(u_j, w_j \lambda, \gamma, d) = \begin{cases} 
   \text{sign}(u_j) \frac{|u_j| - w_j \lambda}{d} & \text{if } |u_j| \leq (d + 1) w_j \lambda, \\
   \text{sign}(u_j) \frac{[(\gamma-1)|u_j| - w_j \gamma \lambda]}{[(\gamma-1)d - 1]} & \text{if } (d + 1) w_j \lambda < |u_j| \leq w_j \gamma \lambda d, \\
   u_j / d, & \text{if } |u_j| > w_j \gamma \lambda d. 
\end{cases}$$  

   where $\gamma > 2$.

5. **Group Lasso**

   The update for the $k$-th group is

   $$\beta_{g_k}^{(t+1)} = G(u_{g_k}, c_k \lambda, d) = \frac{u_{g_k}}{d} \left( 1 - \frac{c_k \lambda}{\|u_{g_k}\|_2} \right)_+.$$  


6. Group MCP

The update for the $k$-th group is

$$\beta_{gk}^{(t+1)} = M(\|u_{gk}\|, c_k\lambda, \gamma, d) \frac{u_{gk}}{\|u_{gk}\|}.$$  \hfill (9)

7. Group SCAD

The update for the $k$-th group is

$$\beta_{gk}^{(t+1)} = C(\|u_{gk}\|, c_k\lambda, \gamma, d) \frac{u_{gk}}{\|u_{gk}\|}.$$  \hfill (10)

8. Sparse Group Lasso

The update for the $k$-th group is

$$\beta_{gk}^{(t+1)} = G(v_{gk}, c_k\lambda(1 - \tau), 1),$$  \hfill (11)

where the $j$-th element of $v$ is $S(u_j, w_j\lambda\tau, d)$. This is true because the thresholding operator of two nested penalties is the composition of the thresholding operators where the innermost nested penalty’s thresholding operator is evaluated first. For more details and theoretical justification of the composition of proximal operators for nested penalties, see Jenatton, Mairal, Bach, and Obozinski (2010).

3. Parallelization and fast cross validation

The OEM algorithm lends itself naturally to efficient computation for cross validation for penalized linear regression models. When the number of variables is not too large (ideally $n \gg p$) relative to the number of observations, computation for cross validation using the OEM algorithm is on a similar order of computational complexity as fitting one model for the full data. To see why this is the case, note that the key computational step in OEM is in forming the matrix $A$. Recall that $A = dI_p - X^\top X$. In $K$-fold cross validation, the design matrix is randomly partitioned into $K$ submatrices as

$$X = \begin{pmatrix} X_1 \\ \vdots \\ X_K \end{pmatrix}.$$  

Then for the $k$th cross validation model, the OEM algorithm requires the quantities $A_{(k)} = d_{(k)}I_p - X_{(k)}^\top X_{(k)}$ and $X_{(k)}^\top y_{-k}$, where $X_{(k)}$ is the design matrix $X$ with the $k$th submatrix $X_k$ removed and $y_{-k}$ is the response vector with the elements from the $k$th fold removed. Then trivially, we have that $A = dI_p - \sum_{k=1}^K X_k^\top X_k$, $A_{(k)} = dI_p - \sum_{c=1,\ldots,K, c \neq k} X_c^\top X_c$, and $X_{(k)}^\top y_{-k} = \sum_{c=1,\ldots,K, c \neq k} X_c^\top y_c$. Then the main computational tasks for fitting a model on the entire training data and for fitting $k$ models for cross validation is in computing $X_k^\top X_k$ and $X_k^\top y_k$ for each $k$, which has a total computational complexity of $O(np^2 + np)$ for any $k$. Hence, we can precompute these quantities and the computation time for the entire cross validation procedure can be dramatically reduced from the naive procedure of fitting models.
for each cross validation fold individually. It is clear that $X_k^\top X_k$ and $X_k^\top y_k$ can be computed independently across all $k$ and hence we can reduce the computational load even further by computing them in parallel. The techniques presented here are not applicable to models beyond the linear model, such as logistic regression.

4. Extension to logistic regression

The logistic regression model is often used when the response of interest is a binary outcome. The OEM algorithm can be extended to handle logistic regression models by using a proximal Newton algorithm similar to that used in the glmnet package and described in Friedman et al. (2010). OEM can act as a replacement for coordinate descent in the inner loop in the algorithm described in Section 3 of Friedman et al. (2010). While we do not present any new algorithmic results here, for the sake of clarity we will outline the proximal Newton algorithm of Friedman et al. (2010) that we use.

The response variable $Y$ takes values in $\{0, 1\}$. The logistic regression model posits the following model for the probability of an event conditional on predictors:

$$
\mu(x) = \Pr(Y = 1|x) = \frac{1}{1 + \exp(-(x^\top \beta))}.
$$

Under this model, we compute penalized regression estimates by maximizing the following penalized log-likelihood with respect to $\beta$:

$$
\frac{1}{n} \sum_{i=1}^{n} \left\{ y_i x_i^\top \beta + \log(1 + \exp(x_i^\top \beta)) \right\} - P(\lambda(\beta)). \tag{12}
$$

Then, given a current estimate $\hat{\beta}$, we approximate (12) with the following weighted penalized linear regression:

$$
-\frac{1}{2n} \sum_{i=1}^{n} w_i \{ z_i - x_i^\top \beta \} - P(\lambda(\beta)), \tag{13}
$$

where

$$
z_i = x_i^\top \hat{\beta} + \frac{y_i - \hat{\mu}(x_i)}{\hat{\mu}(x_i)(1 - \hat{\mu}(x_i))},
$$

$$
w_i = \hat{\mu}(x_i)(1 - \hat{\mu}(x_i))
$$

and $\hat{\mu}(x_i)$ is evaluated at $\hat{\beta}$. Here, $z_i$ are the working responses and $w_i$ are the weights. For each iteration of the proximal Newton algorithm, we maximize (13) using the OEM algorithm. Similar to Krishnapuram, Carin, Figueiredo, and Hartemink (2005); Friedman et al. (2010), we optionally employ an approximation to the Hessian using an upper-bound of $w_i = 0.25$ for all $i$. This upper bound is often quite efficient for big tall data settings.

5. The oem package

5.1. The oem() function

The function oem() is the main workhorse of the oem package.
\texttt{R> nobs <- 1e4}
\texttt{R> nvars <- 25}
\texttt{R> rho <- 0.25}
\texttt{R> sigma <- matrix(rho, ncol = nvars, nrow = nvars)}
\texttt{R> diag(sigma) <- 1}
\texttt{R> x <- mvrnorm(n = nobs, mu = numeric(nvars), Sigma = sigma)}
\texttt{R> y <- drop(x %*% c(0.5, 0.5, -0.5, -0.5, rep(0, nvars - 4))) +}
\texttt{+ rnorm(nobs, sd = 3)}

The group membership indices for each covariate must be specified for the group lasso via the argument \texttt{groups}. The argument \texttt{gamma} specifies the $\gamma$ value for MCP. The \texttt{plot} method for ‘\texttt{oemfit}’ objects allows the user to plot the estimated coefficient paths. Its argument \texttt{which.model} allows the user to select the model to be plotted.

\texttt{R> fit <- oem(x = x, y = y, penalty = c("lasso", "mcp", "grp.lasso"),}
\texttt{+ gamma = 2, groups = rep(1:5, each = 5), lambda.min.ratio = 1e-3)}
\texttt{R> par(mar = c(5, 5, 5, 3) + 0.1)}
\texttt{R> layout(matrix(1:3, ncol = 3))}
\texttt{R> plot(fit, which.model = 1, xvar = "lambda", cex.main = 3,}
\texttt{+ cex.axis = 1.25, cex.lab = 2)}
\texttt{R> plot(fit, which.model = 2, xvar = "lambda", cex.main = 3,}
\texttt{+ cex.axis = 1.25, cex.lab = 2)}
\texttt{R> plot(fit, which.model = 3, xvar = "lambda", cex.main = 3,}
\texttt{+ cex.axis = 1.25, cex.lab = 2)}

To compute the loss function in addition to the estimated coefficients, the \texttt{compute.loss} argument must be set to \texttt{TRUE} like in the following code:

\texttt{R> fit <- oem(x = x, y = y, penalty = c("lasso", "mcp", "grp.lasso"),}
\texttt{+ gamma = 2, groups = rep(1:5, each = 5), lambda.min.ratio = 1e-3,}
\texttt{+ compute.loss = TRUE)}

By default, \texttt{compute.loss} is set to \texttt{FALSE} because it adds a large computational burden, especially when many penalties are input. The \texttt{logLik} method for ‘\texttt{oemfit}’ objects can be used for objects fitted using \texttt{oem()} with \texttt{compute.loss = TRUE}. The \texttt{logLik} method for ‘\texttt{oemfit}’ objects has a \texttt{which.model} argument which allows to specify the model:
R> logLik(fit, which.model = 2)[c(1, 25, 50, 100)]

[1] -14189.39 -13804.72 -13795.76 -13795.11

5.2. Fitting multiple penalties

The OEM algorithm is well-suited to quickly estimate a solution path for multiple penalties simultaneously for the linear model if the number of variables is not too large, often when the number of variables is several thousands or fewer, provided the number of observations is larger than the number of variables. Ideally the number of observations should be at least ten times larger than the number of variables for best performance. Once the quantities $A$ and $X^\top y$ are computed initially, then the remaining computational complexity for the OEM algorithm for a given model and tuning parameter is just $O(p^2)$ per iteration. To demonstrate the efficiency, consider the following simulated example:

```R
R> nobs <- 1e6
R> nvars <- 100
R> rho <- 0.25
R> sigma <- matrix(rho, ncol = nvars, nrow = nvars)
R> diag(sigma) <- 1
R> x2 <- mvrnorm(n = nobs, mu = numeric(nvars), Sigma = sigma)
R> y2 <- drop(x2 %*% c(0.5, 0.5, -0.5, -0.5, rep(0, nvars - 4))) +
+ rnorm(nobs, sd = 5)
R> mb <- microbenchmark(
+   "oem[lasso]" = oem(x = x2, y = y2, penalty = c("lasso"),
+   gamma = 3, groups = rep(1:20, each = 5)),
+   "oem[all]" = oem(x = x2, y = y2, penalty = c("lasso", "mcp",
+   "grp.lasso", "scad"), gamma = 3, groups = rep(1:20, each = 5)),
+   times = 10L)
R> print(mb, digits = 3)
```

Unit: seconds

<table>
<thead>
<tr>
<th>expr</th>
<th>min</th>
<th>lq</th>
<th>mean</th>
<th>median</th>
<th>uq</th>
<th>max</th>
<th>neval</th>
<th>cld</th>
</tr>
</thead>
<tbody>
<tr>
<td>oem[lasso]</td>
<td>2.38</td>
<td>2.39</td>
<td>2.41</td>
<td>2.42</td>
<td>2.44</td>
<td>2.46</td>
<td>10</td>
<td>a</td>
</tr>
<tr>
<td>oem[all]</td>
<td>2.88</td>
<td>2.89</td>
<td>2.92</td>
<td>2.91</td>
<td>2.93</td>
<td>3.00</td>
<td>10</td>
<td>b</td>
</tr>
</tbody>
</table>

5.3. Parallel support via OpenMP

As noted in Section 3, the key quantities necessary for the OEM algorithm can be computed in parallel. By specifying ncores to have a value greater than 1, the oem() function automatically employs OpenMP (OpenMP Architecture Review Board 2015) to compute $A$ and $X^\top y$ in parallel. Due to memory access inefficiencies in breaking up the computation of $A$ into pieces, using multiple cores does not speed up computation linearly. It is typical for OpenMP not to result in linear speedups, especially on Windows machines, due to its overhead costs. Furthermore, if a user does not have OpenMP on their machine, the oem package will still run normally on one core. In the following example, we can see a slight benefit from invoking...
the use of extra cores on a Windows laptop with a dual core multithreaded Intel i5 processor with 4 threads.

```r
R> nobs <- 1e5
R> nvars <- 500
R> rho <- 0.25
R> sigma <- rho ** abs(outer(1:nvars, 1:nvars, FUN = "-"))
R> x2 <- mvrnorm(n = nobs, mu = numeric(nvars), Sigma = sigma)
R> y2 <- drop(x2 %*% c(0.5, 0.5, -0.5, -0.5, rep(0, nvars - 4))) +
+     rnorm(nobs, sd = 5)
R> mb <- microbenchmark(
+     "oem" = oem(x = x2, y = y2, penalty = c("lasso", "mcp",
+     "grp.lasso", "scad"), gamma = 3, groups = rep(1:20, each = 25)),
+     "oem[parallel]" = oem(x = x2, y = y2, ncores = 2,
+     penalty = c("lasso", "mcp", "grp.lasso", "scad"), gamma = 3,
+     groups = rep(1:20, each = 25)), times = 10L)
R> print(mb, digits = 3)
```

```
Unit: seconds
  expr     min      lq    mean    median      uq     max neval  cld
oem        4.80    4.85    4.93    4.93    5.00    5.07    10    b
oem[parallel]  3.72    3.84    3.98    3.95    4.11    4.31    10    a
```

5.4. The `cv.oem()` function

The `cv.oem()` function is used for cross validation of penalized models fitted by the `oem()` function. It does not use the method described in Section 3 and hence can be used for models beyond the linear model. It can also benefit from parallelization using either OpenMP or the `foreach` package. For the former, one only needs to specify the `ncores` argument of `cv.oem()` and the key quantities for the OEM algorithm are computed in parallel for each cross validation fold. With the `foreach` package (Kane et al. 2013; Revolution Analytics and Weston 2022b), cores must be “registered” in advance using `doParallel` (Microsoft Corporation and Weston 2022), `doMC` (Revolution Analytics and Weston 2022a), or otherwise. Each cross validation fold is computed on a separate core, which may be more efficient depending on the user’s hardware.

```r
R> cvfit <- cv.oem(x = x2, y = y2, penalty = c("lasso", "mcp", "grp.lasso"),
+     gamma = 2, groups = rep(1:5, each = 5), nfolds = 10)
```

The best performing model and its corresponding best tuning parameter can be accessed via:

```r
R> cvfit$best.model
[1] "mcp"
R> cvfit$lambda.min
[1] 0.0739055
```
A summary method for ‘cv.oem’ objects is available, similar to the summary method for objects created with cv.ncvreg() of the ncvreg package, which prints output from all of the cross validated models. It can be used like the following:

```r
R> summary(cvfit)
```

lasso-penalized linear regression with n=10000, p=25
At minimum cross-validation error (lambda=0.0242):

- Nonzero coefficients: 13
- Cross-validation error (Mean-Squared Error): 9.11
- Scale estimate (sigma): 3.018

```r
cross-validation error (Mean-Squared Error): 9.10
Scale estimate (sigma): 3.016
```

```r
mcp-penalized linear regression with n=10000, p=25
At minimum cross-validation error (lambda=0.0739):

- Nonzero coefficients: 5
- Cross-validation error (Mean-Squared Error): 9.10
- Scale estimate (sigma): 3.016

```r
cross-validation error (Mean-Squared Error): 9.10
Scale estimate (sigma): 3.017
```  
Predictions from any model (which.model = 2) or the best of all models (which.model = "best.model") are obtained using the predict method for ‘cv.oem’ objects. The tuning parameter is specified via the argument s, which can take numeric values or "lambda.min" for the best tuning parameter or "lambda.1se" for a good but more conservative tuning parameter.

```r
R> predict(cvfit, newx = x[1:3, ], which.model = "best.model",
+       s = "lambda.min")
```

```r
[,1]
[1,]  -0.2233264
[2,]   0.2211386
[3,]   0.4760399
```

5.5. The xval.oem() function

The xval.oem() function is much like cv.oem() but is limited to use for linear models only. It is significantly faster than cv.oem(), as it uses the method described in Section 3. Whereas
cv.oem() performs repeated calls to oem(), all of the primary computation in xval.oem() is carried out in C++. We chose to keep the xval.oem() function separate from the cv.oem() function because the underlying code between the two methods is vastly different and furthermore because xval.oem() is not available for logistic regression models.

```r
R> xvalfit <- xval.oem(x = x, y = y, penalty = c("lasso", "mcp", + "grp.lasso"), gamma = 2, groups = rep(1:5, each = 5), nfolds = 10)
R> yrng <- range(c(unlist(xvalfit$cvup), unlist(xvalfit$cvlo)))
R> layout(matrix(1:3, ncol = 3))
R> par(mar = c(5, 5, 5, 3) + 0.1)
R> plot(xvalfit, which.model = 1, ylim = yrng, cex.main = 3, + cex.axis = 1.25, cex.lab = 2)
R> plot(xvalfit, which.model = 2, ylim = yrng, cex.main = 3, + cex.axis = 1.25, cex.lab = 2)
R> plot(xvalfit, which.model = 3, ylim = yrng, cex.main = 3, + cex.axis = 1.25, cex.lab = 2)
```

### 5.6. OEM with precomputation with the oem.xtx() function

The key quantities, $X^TX$ and $X^Ty$ can be computed in parallel, and when data are stored across a large cluster, their computation can be performed in a straightforward manner. When they are available, the oem.xtx() function can be used to carry out the OEM algorithm based on these quantities instead of on the full design matrix $X$ and the full response vector $y$. All methods available to objects fitted by oem() are also available to objects fitted by oem.xtx().

```r
R> xtx <- crossprod(x) / nrow(x)
R> xty <- crossprod(x, y) / nrow(x)
R> fitxtx <- oem.xtx(xtx, xty, penalty = c("lasso", "mcp", "grp.lasso"), + gamma = 2, groups = rep(1:5, each = 5))
```

### 5.7. Out-of-memory computation with the big.oem() function

Standard R objects are stored in memory and thus, when a design matrix is too large for memory it cannot be used for computation in the standard way. The bigmemory package
offers objects which point to data stored on disk and thus allows users to bypass memory
limitations. It also provides access to C++ objects which do the same. These objects are
highly efficient due to memory mapping, which is a method of mapping a data file to virtual
memory and allows for efficient moving of data in and out of memory from disk. For further
details on memory mapping, we refer readers to Bovet and Cesati (2005) and Kane et al.
(2013). The \texttt{big.oem()} function allows for out-of-memory computation by linking \texttt{Eigen}
matrix objects to data stored on disk via \texttt{bigmemory}.

The standard approach for loading the data from SEXP objects (\texttt{X_} in the example below) to
\texttt{Eigen} matrix objects (\texttt{X} in the example below) in C++ looks like:

```c++
using Eigen::Map;
using Eigen::MatrixXd;
const Map<MatrixXd> X(as<Map<MatrixXd> >(X_));
```

To instead map from an object which is a pointer to data on disk, we first need to load the
\texttt{bigmemory} headers:

```c++
#include <bigmemory/MatrixAccessor.hpp>
#include <bigmemory/BigMatrix.h>
```

Then we link the pointer (passed from R to C++ as \texttt{X_} and set as \texttt{bigPtr} below) to data to
an \texttt{Eigen} matrix object via:

```c++
XPtr<BigMatrix> bigPtr(X_);
const Map<MatrixXd> X = Map<MatrixXd>((double *)bigPtr->matrix(), bigPtr->nrow(), bigPtr->ncol());
```

The remaining computation for OEM is carried out similarly as for \texttt{oem()}, yet here the object
\texttt{X} is not stored in memory.

To test out \texttt{big.oem()} and demonstrate its memory usage profile we simulate a large dataset
and save it as a “filebacked” ‘big.matrix’ object from the \texttt{bigmemory} package.

```r
R> nobs <- 1e6
R> nvars <- 250
R> bkFile <- "big_matrix.bk"
R> descFile <- "big_matrix.desc"
R> big_mat <- filebacked.big.matrix(nrow = nobs, ncol = nvars,
+ type = "double", backingfil = bkFile, backingpath = ".",
+ descriptorfile = descFile, dimnames = c(NULL, NULL))
R> for (i in 1:nvars) {
+ big_mat[, i] = rnorm(nobs)
+ }
R> yb <- rnorm(nobs, sd = 5)
```

Using the \texttt{profvis()} function of the \texttt{profvis} package (Chang, Luraschi, and Mastny 2020),
we can see that no copies of the design matrix are made at any point. Furthermore, a maximum
of 173 Megabytes are used by the R session during this simulation, whereas the size of the design
matrix is 1.9 Gigabytes. The following code generates an interactive HTML visualization of
the memory usage of \texttt{big.oem()} line-by-line:
Here we save a copy of the design matrix in memory for use by `oem()`:

```r
R> xb <- big_mat[, ]
R> print(object.size(xb), units = "Mb")
```

1907.3 Mb

```r
R> print(object.size(big_mat), units = "Mb")
```

0 Mb

The following benchmark for on-disk computation is on a system with a hard drive with 7200 RPM, 16 MB Cache, and SATA 3.0 Gigabytes per second (a quite modest setup compared with a system with a solid state drive). Even without a solid state drive we pay little time penalty for computing on disk over computing in memory.

```r
R> mb <- microbenchmark(
+   "big.oem" = big.oem(x = big_mat, y = yb, penalty = c("lasso", 
+                     "grp.lasso", "mcp", "scad"), gamma = 3, groups = rep(1:50, each = 5)),
+   "oem" = oem(x = xb, y = yb, penalty = c("lasso", "grp.lasso", "mcp", 
+                     "scad"), gamma = 3, groups = rep(1:50, each = 5)), times = 10L)
R> print(mb, digits = 3)
```

Unit: seconds

<table>
<thead>
<tr>
<th>expr</th>
<th>min</th>
<th>lq</th>
<th>mean</th>
<th>median</th>
<th>uq</th>
<th>max</th>
<th>neval</th>
<th>cld</th>
</tr>
</thead>
<tbody>
<tr>
<td>big.oem</td>
<td>8.46</td>
<td>8.51</td>
<td>8.73</td>
<td>8.57</td>
<td>8.69</td>
<td>9.73</td>
<td>10</td>
<td>a</td>
</tr>
<tr>
<td>oem</td>
<td>9.81</td>
<td>9.85</td>
<td>10.36</td>
<td>10.08</td>
<td>10.65</td>
<td>12.03</td>
<td>10</td>
<td>b</td>
</tr>
</tbody>
</table>

5.8. Sparse matrix support

The `oem()` and `cv.oem()` functions accept sparse design matrices as provided by package `Matrix` (Bates, Maechler, and Jagan 2022) in the ‘CsparseMatrix’ class of objects. If the design matrix provided has a high degree of sparsity, using a ‘CsparseMatrix’ object can result in a substantial computational speedup and reduction in memory usage.

```r
R> library("Matrix")
R> n.obs <- 1e5
R> n.vars <- 200
R> true.beta <- c(runif(15, -0.25, 0.25), rep(0, n.vars - 15))
R> xs <- rsparsematrix(n.obs, n.vars, density = 0.01)
R> ys <- rnorm(n.obs, sd = 3) + as.vector(xs %%*% true.beta)
```
R> x.dense <- as.matrix(xs)
R> mb <- microbenchmark(
+   fit = oem(x = x.dense, y = ys, penalty = c("lasso", "grp.lasso"),
+              groups = rep(1:40, each = 5)),
+   fit.s = oem(x = xs, y = ys, penalty = c("lasso", "grp.lasso"),
+              groups = rep(1:40, each = 5)), times = 10L)
R> print(mb, digits = 3)

Unit: milliseconds
cld
  expr      min      lq     mean    median      uq     max neval
fit  669.9  672.5  679.3   680.3  682.2  690.6    10  b
fit.s  63.2  64.1  65.5    65.3  66.8  68.2    10  a

5.9. API comparison with glmnet

The application program interface (API) of the oem package was designed to be familiar to users of the glmnet package. Data ready for use by glmnet() can be used directly by oem(). Most of the arguments are the same, except the penalty argument and other arguments relevant to the various penalties available in oem().

Here we fit linear models with a lasso penalty using oem() and glmnet():

R> oem.fit <- oem(x = x, y = y, penalty = "lasso")
R> glmnet.fit <- glmnet(x = x, y = y)

Here we fit linear models with a lasso penalty using oem() and glmnet() with sparse design matrices:

R> oem.fit.sp <- oem(x = xs, y = ys, penalty = "lasso")
R> glmnet.fit.sp <- glmnet(x = xs, y = ys)

Now we make predictions using the fitted model objects from both packages:

R> preds.oem <- predict(oem.fit, newx = x)
R> preds glmnet <- predict(glmnet.fit, newx = x)

We now plot the coefficient paths using both fitted model objects:

R> plot(oem.fit, xvar = "norm")
R> plot(glmnet.fit, xvar = "norm")

We now fit linear models with a lasso penalty and select the tuning parameter with cross validation using cv.oem(), xval.oem(), and cv.glmnet():

R> oem.cv.fit <- cv.oem(x = x, y = y, penalty = "lasso")
R> oem.xv.fit <- xval.oem(x = x, y = y, penalty = "lasso")
R> glmnet.cv.fit <- cv.glmnet(x = x, y = y)

We now plot the cross validation errors using all fitted cross validation model objects:
R> plot(oem.cv.fit)
R> plot(oem.xv.fit)
R> plot(glmnet.cv.fit)

We now make predictions using the best tuning parameter according to the cross validation error using all fitted cross validation model objects:

R> preds.cv.oem <- predict(oem.cv.fit, newx = x, s = "lambda.min")
R> preds.xv.oem <- predict(oem.xv.fit, newx = x, s = "lambda.min")
R> preds.cv.glmnet <- predict(glmnet.cv.fit, newx = x, s = "lambda.min")

6. Timings

Extensive numerical studies are conducted in Xiong et al. (2016) regarding the OEM algorithm for computation time for paths of tuning parameters for various penalties. So in the following simulation studies, we will focus on computation time for the special features of the oem package such as cross validation and sparse matrix support. All simulations are run on a 64-bit machine with an Intel Xeon E5-2470 2.30 GHz CPU and 128 Gigabytes of main memory and a Linux operating system.

6.1. Cross validation

In this section we will compare the computation time of the cv.oem() and xval.oem() for various penalties, both individually and simultaneously, with the cross validation functions from various other packages, including glmnet, ncvreg, grpreg, gglasso, and the Python (van Rossum et al. 2011) package sklearn (Pedregosa et al. 2011). The model class we use from sklearn is ‘LassoCV’, which performs cross validation for lasso linear models. In particular, we focus on the comparison with glmnet, as it has been carefully developed with computation time in mind and has long been the gold standard for computational performance.

In the simulation setup, we generate the design matrix from a multivariate normal distribution with covariance matrix $(\sigma_{ij}) = 0.5^{|i-j|}$. Responses are generated from the following model:

$$ y = X\beta + \epsilon, $$

where the first five elements of $\beta$ are $(-0.5, -0.5, 0.5, 0.5, 1)$ and the remaining are zero and $\epsilon$ is an independent mean zero normal random variable with standard deviation 2. The total number of observations $n$ is set to $10^5$ and $10^6$, the number of variables $p$ is varied from 50 to 500, and the number of folds for cross validation is set to 10. For grouped regularization, groups of variables of size 25 are chosen contiguously. For the MCP regularization that has an additional tuning parameter $\gamma$, the tuning parameter $\gamma$ is chosen to be 3. Each method is fitted using the same sequence of 100 values of the tuning parameter $\lambda$ and convergence is specified to be the same level of precision for each method. The computation times for the glmnet and oem functions without cross validation are given as a reference point.

From the results in Figure 3, both the xval.oem() and cv.oem() functions are competitive with all other cross validation alternatives. Surprisingly, the xval.oem() function for the lasso penalty only is competitive with and in many scenarios is even faster than glmnet,
Figure 3: Average computing time over ten runs in log(seconds) for the functions that perform cross validation for selection of the tuning parameter for various penalized regression methods.

Figure 4: Average computing time over ten runs in log(seconds) for the `xval.oem()` function with a varying number of cores. Results are shown for just one penalty (lasso) and three simultaneously fitted penalties (lasso, group lasso, and MCP) with 100 values for the tuning parameter \( \lambda \) for each penalty.

which does not perform cross validation. The `xval.oem()` is clearly faster than `cv.oem()` in all scenarios. In many scenarios `cv.oem()` takes at least 6 times longer than `xval.oem()`. Both cross validation functions from the `oem` package are nearly as fast in computing for three penalties simultaneously as they are for just one. We have found in general that for any scenario where \( n \gg p \) and cross validation is required, it is worth considering `oem` as
**oem**: A Package for Fast Penalized Regression

![Figure 5: Average computing time over ten runs in log(seconds) for computation of lasso linear regression models using sparse matrices.](image)

a fast alternative to other packages. In particular, a rough rule of thumb is that `oem` is advantageous when \( n > 10p \), however, `oem` may be advantageous with fewer observations than this for penalties other than the lasso, such as MCP, SCAD, and group lasso.

Now we test the impact of parallelization on the computation time of `xval.oem()` using the same simulation setup as above except now we additionally vary the number of cores used. From the results in Figure 4, we can see that using parallelization through *OpenMP* helps to some degree. However, it is important to note that using more cores than the number of cross validation folds will unlikely result in better computation time than the same number of cores as the number of folds due to how parallelization is implemented.

### 6.2. Sparse matrices

In the following simulated example, sparse design matrices are generated using the function `rsparsematrix()` of the *Matrix* package with nonzero entries generated from an independent standard normal distribution. The proportion of zero entries in the design matrix is set to 0.99, 0.95, and 0.9. The total number of observations \( n \) is set to \( 10^5 \) and \( 10^6 \) and the number of variables \( p \) is varied from 250 to 1000. Responses are generated from the same model as in Section 6.1. Each method is fitted using the same sequence of 100 values of the tuning parameter \( \lambda \) and convergence is specified to be the same level of precision for each method. The authors are not aware of penalized regression packages other than *glmnet* that provide support for sparse matrices, so we just compare `oem` with *glmnet*. From the results in Figure 5, it can be seen that `oem()` is superior in computation time for very tall data settings with a high degree of sparsity of the design matrix.

### 6.3. Penalized logistic regression

In this simulation, we generate the design matrix from a multivariate normal distribution...
with covariance matrix $(\sigma_{ij}) = 0.5|i-j|$. Responses are generated from the following model:

$$P(Y_i = 1|x_i) = \frac{1}{1 + \exp(-x_i^\top \beta)}, \text{ for } i = 1, \ldots, n,$$

where the first five elements of $\beta$ are $(-0.5, -0.5, 0.5, 0.5, 1)$ and the remaining are zero. The total number of observations $n$ is set to $10^4$ and $10^5$, the number of variables $p$ is varied from 50 to 500. For grouped regularization, groups of variables of size 25 are chosen contiguously. For the MCP regularization, the tuning parameter $\gamma$ is chosen to be 3. Each method is fitted using the same sequence of 25 values of the tuning parameter $\lambda$. Both \texttt{glmnet()} and \texttt{oem()} allow for a Hessian upper-bound approximation for logistic regression models, so in this simulation we compare both the vanilla versions of each in addition to the Hessian upper-bound versions. For logistic regression models, unlike linear regression, the various functions compared use vastly different convergence criteria, so care was taken to set the convergence thresholds for the different methods to achieve a similar level of precision.

The average computing times over 10 runs are displayed in Figure 6. The Hessian upper-bound versions of \texttt{glmnet()} and \texttt{oem()} are given by the dashed lines. The \texttt{glmnet()} function is the fastest for lasso-penalized logistic regression, but \texttt{oem()} without the use of a Hessian upper-bound is nearly as fast as \texttt{glmnet()} when the sample size gets larger. For the MCP and group lasso, \texttt{oem()} with the Hessian upper-bound option is the fastest across the vast majority of simulation settings. In general, the Hessian upper-bound is most advantageous when the number of variables is moderately large.

### 6.4. Quality of solutions

In this section we present information about the quality of solutions provided by each of the compared packages. Using the same simulation setup and precisions for convergence as the cross validation and binomial simulations, we investigate the numerical precision of the
Table 2: We can see that the solutions provided by the objective function value differences for the linear model simulations are presented in those of other functions except for the sequence of tuning parameter values we present the differences in objective functions averaged over all of the values of the tuning parameter for the linear regression simulations. Negative here means that the \texttt{oem()} function results in estimates with a lower objective function value. Since each simulation is run on a modified objective function, wherein the covariates are orthonormalized within groups as described in Section 2.1 of Breheny and Huang (2015). Due to this fact we do not compare \texttt{oem()} and other methods in objective function values averaged over all of the values of the tuning parameter for the linear regression simulations. Negative here means the \texttt{oem()} function results in estimates with a lower objective function value. The heading “\texttt{glmnet (ub)}” corresponds to \texttt{glmnet()} with the Hessian upper bound option.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$p$</th>
<th>\texttt{glmnet}</th>
<th>\texttt{glmnet (ub)}</th>
<th>\texttt{gglasso}</th>
<th>\texttt{ncvreg}</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^5$</td>
<td>50</td>
<td>$-4.04 \times 10^{-09}$</td>
<td>$-4.39 \times 10^{-06}$</td>
<td>$-1.45 \times 10^{-12}$</td>
<td>\</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>$-8.66 \times 10^{-09}$</td>
<td>$-3.45 \times 10^{-06}$</td>
<td>3.37 $\times 10^{-11}$</td>
<td>\</td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>$-1.87 \times 10^{-08}$</td>
<td>$-3.33 \times 10^{-06}$</td>
<td>$-1.51 \times 10^{-12}$</td>
<td>\</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>$-3.35 \times 10^{-08}$</td>
<td>$-3.37 \times 10^{-06}$</td>
<td>2.80 $\times 10^{-11}$</td>
<td>\</td>
</tr>
<tr>
<td>$10^6$</td>
<td>50</td>
<td>$-4.20 \times 10^{-09}$</td>
<td>$-3.35 \times 10^{-06}$</td>
<td>1.68 $\times 10^{-11}$</td>
<td>\</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>$-3.18 \times 10^{-09}$</td>
<td>$-3.40 \times 10^{-06}$</td>
<td>4.13 $\times 10^{-12}$</td>
<td>\</td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>$-7.51 \times 10^{-09}$</td>
<td>$-3.35 \times 10^{-06}$</td>
<td>6.62 $\times 10^{-12}$</td>
<td>\</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>$-1.57 \times 10^{-08}$</td>
<td>$-3.35 \times 10^{-06}$</td>
<td>5.99 $\times 10^{-15}$</td>
<td>\</td>
</tr>
</tbody>
</table>

Table 3: Averages differences between the \texttt{oem()} function and other methods in objective function values averaged over all of the values of the tuning parameter for the logistic regression simulations. Negative here means the \texttt{oem()} function results in estimates with a lower objective function value. The \texttt{gglasso} and \texttt{ncvreg} methods by presenting the objective function values corresponding to the given solutions. Specifically, we look at the difference in the objective function values between \texttt{oem()} and the comparative functions. A negative value here indicates that the \texttt{oem()} function results in solutions with a smaller value of the objective function. Since each simulation is run on a sequence of tuning parameter values we present the differences in objective functions averaged over the tuning parameters. Since the objective function values across the different tuning parameter values are on the same scale, this comparison is reasonable. While the \texttt{grpreg()} function from the \texttt{grpreg} package provides solutions for the group lasso, it minimizes a slightly modified objective function, wherein the covariates are orthonormalized within groups as described in Section 2.1 of Breheny and Huang (2015). Due to this fact we do not compare the objective function values associated with results returned by \texttt{grpreg()}.

The objective function value differences for the linear model simulations are presented in Table 2. We can see that the solutions provided by \texttt{oem()} are generally more precise than those of other functions except \texttt{ncvreg()} for the MCP. However, the differences between the
solutions provided by \texttt{oem()\textcolor{red}{\texttt{}} and \texttt{ncvreg()}} for the MCP are close to machine precision. The objective function value differences for the logistic model simulations are presented in Tables 3 and 4. Table 3 refers to comparisons between \texttt{oem()} using the full Hessian and other methods and Table 4 refers to comparisons between \texttt{oem()} using the Hessian upper-bound. We can see that \texttt{oem()} in both cases results in solutions which are generally more precise than other methods. Interestingly, for the logistic regression simulations unlike the linear regression simulations, we see that \texttt{oem()} provides solutions which are dramatically more precise than \texttt{ncvreg() for the MCP}. These results indicate that the nonconvexity of the MCP has more serious consequences for logistic regression in terms of algorithmic choices.

**Acknowledgments**

This material is based upon work supported by, or in part by, NSF Grants DMS 1055214 and DMS 1564376, and NIH grant T32HL083806.

**References**


Affiliation:
Jared D. Huling
Division of Biostatistics
University of Minnesota, Twin Cities
Minnesota 55455, United States of America
E-mail: huling@umn.edu
URL: https://jaredhuling.org/