Exercises 1: Preliminaries

Su Chen

September 8, 2016

1 LINEAR REGRESSION

Consider the simple linear regression model

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

where $y = (y_1, \ldots, y_N)$ is an N-vector of responses, X is an $N \times P$ matrix of features whose ith row is x_i, and e is a vector of model residuals. The goal is to estimate β, the unknown P-vector of regression coefficients.
Let's say you trust the precision of some observations more than others, and therefore decide to estimate β by the principle of weighted least squares (WLS):

$$\hat{\beta} = \arg\min_{\beta \in \mathscr{R}^P} \sum_{i=1}^{N} \frac{w_i}{2}(y_i - x_i^T\beta)^2,$$

where w_i is the weight for observation i. (Higher weight means more influence on the answer; the factor of $1/2$ is just for convenience, as you'll see later.)

(A) Rewrite the WLS objective function[1] above in terms of vectors and matrices, and show that $\hat{\beta}$ is the solution to the following linear system of P equations in P unknowns:

$$(X^T W X)\hat{\beta} = X^T W y,$$

where W is the diagonal matrix of weights.

[1] That is, the thing to be minimized.

Sol:

$$\hat{\beta} = \arg\min_{\beta \in \mathscr{R}^P} \sum_{i=1}^{N} \frac{w_i}{2}(y_i - x_i^T\beta)^2$$
$$= \arg\min_{\beta \in \mathscr{R}^P} (Y - X\beta)^T \frac{W}{2}(Y - X\beta)$$

We notice that the target function $f(\beta) = (Y - X\beta)^T \frac{W}{2}(Y - X\beta)$ to minimize here is convex, so to find the argmin, we just need to find the stationary point, i.e. to take the gradient and solve for 0:

$$\hat{\beta} = \arg\min_{\beta \in \mathscr{R}^P} \sum_{i=1}^{N} \frac{w_i}{2}(y_i - x_i^T\beta)^2$$
$$\iff \nabla f(\hat{\beta}) = 2X^T \frac{W}{2}(Y - X\hat{\beta}) = 0$$
$$\iff (X^T W X)\hat{\beta} = X^T W Y$$

(B) One way to calculate $\hat{\beta}$ is to: (1) recognize that, trivially, the solution to the above linear system must satisfy $\hat{\beta} = (X^T W X)^{-1} X^T W y$; and (2) to calculate this directly, i.e. by inverting $X^T W X$. Let's call this the "inversion method" for calculating the WLS solution.

Numerically speaking, is the inversion method the fastest and most stable way to actually solve the above linear system? Do some independent sleuthing on this question. Summarize what you find, and provide pseudo-code for at least one alternate method based on matrix factorizations—call it "your method" for short. (Note: our linear system is not a special flower; whatever you discover about general linear systems should apply here.)

Sol: The inversion method has the following disadvantages:

1. the time complexity to invert a $N \times N$ matrix is N^3, so the time complexity to invert $(X^T W X)$ is P^3, and this can be very computationally expensive when P is large.
2. the inversion method can also be unstable because if one of the weight is numerically 0, then the matrix $X^T W X$ will become singular and cannot be inverted.
3. also matrix inversion is known to be unstable in numerical analysis.

pseudo-code for alternate method based on Cholesky Decomposition:

- Cholesky Decomposition: $(X^T W X) = LL^T$ where L is a lower triangular matrix.
- Solve linear system: $L\gamma = X^T W Y$ where $\gamma = L^T\hat{\beta}$ to get the solution $\hat{\gamma}$.
- Solve linear system: $L^T\hat{\beta} = \hat{\gamma}$ to get the solution $\hat{\beta}$.

Advantages of alternate method over the inversion method:

1. the time complexity to solve a linear system with upper/lower triangular matrix is N^2. So we have decreased the time complexity from P^3 to P^2.
2. If one or more weights are numerically 0, then $X^T W X$ is not positive definite, and the Cholesky Decomposition may not be unique. We can use LDL decomposition instead which does not require the matrix to be positive definite.

(C) Code up functions that implement both the inversion method and your method for an arbitrary X, y, and weights W. Obviously you shouldn't write your own linear algebra routines for doing things like multiplying or decomposing matrices, but don't use a direct model-fitting function like R's "lm" either. Your actual code should look a lot like the pseudo-code you wrote for the previous part. Note: be attentive to how you multiply a matrix by a diagonal matrix, or you'll waste a lot of time multiplying stuff by zero.

Now simulate some silly data from the linear model for a range of values of N and P. (Feel free to assume that the weights w_i are all 1.) It doesn't matter how you do this—e.g. everything can be Gaussian if you want. (We're not concerned with statistical principles yet, just with algorithms, and using least squares is a pretty terrible idea for enormous linear models, anyway.) Just make sure that you explore values of P up into the thousands, and that $N > P$. Benchmark the performance of the inversion solver and your solver across a range of scenarios. (In R, a simple library for this purpose is microbenchmark.)

Sol: I tested the code across a range of scenarios for different N and P, see the results below:

- $N = 200, P = 50$

```
Unit: microseconds
                   expr     min      lq       mean  median      uq        max neval
 inversion_solver(y, X) 458.201 459.667 33230.2320 468.465 541.778 164223.049     5
   default_solver(y, X) 385.622 389.288   492.2186 391.488 404.684    890.011     5
      chol_solver(y, X) 354.831 383.423   526.0886 454.536 458.935    978.718     5
```

- $N = 2000, P = 500$

```
Unit: milliseconds
                   expr      min       lq     mean   median       uq      max neval
 inversion_solver(y, X) 394.3374 396.7200 418.0693 402.6898 446.5422 450.0568     5
   default_solver(y, X) 318.3955 318.8082 324.6732 321.2158 327.4239 337.5226     5
      chol_solver(y, X) 313.8208 313.9190 315.3659 314.2277 316.3164 318.5458     5
```

- $N = 2000, P = 1000$

```
Unit: seconds
                   expr      min       lq     mean   median       uq      max neval
 inversion_solver(y, X) 3.235092 3.272444 3.373972 3.393764 3.451564 3.516995     5
   default_solver(y, X) 2.125599 2.164097 2.284623 2.164479 2.260316 2.708626     5
      chol_solver(y, X) 2.096494 2.101471 2.161027 2.102805 2.189339 2.315026     5
```

- $N = 4000, P = 1000$

```
Unit: seconds
                   expr      min       lq     mean   median       uq      max neval
 inversion_solver(y, X) 3.293999 3.327009 4.048204 4.521505 4.522532 4.575976     5
   default_solver(y, X) 2.564091 3.522809 3.353392 3.531033 3.572115 3.576911     5
      chol_solver(y, X) 2.534099 2.539857 3.004037 2.950430 3.494235 3.501564     5
```

(D) Now what happens if X is a highly sparse matrix, in the sense that most entries are zero? Ideally we'd realize some savings by not doing a whole bunch of needless multiplication by zero in our code.

It's easy to simulate an X matrix that looks like this. A quick-and-dirty way is to simulate a mask of zeros and ones (but mostly zeros), and then do pointwise multiplication with your original feature matrix. For example:

```
N = 2000
P = 500
X = matrix(rnorm(N*P), nrow=N)
mask = matrix(rbinom(N*P,1,0.05), nrow=N)
X = mask*X
X[1:10, 1:10]  # quick visual check
```

Again assume that the weights w_i are all 1. Repeat the previous benchmarking exercise with this new recipe for simulating a sparse X, except add another solver to the mix: one that can solve a linear system $Ax = b$ in a way that exploits the sparsity of A. To do this, you'll need to actually represent the feature matrix X in a sparse format, and then call the appropriate routines for that format. (Again, do some sleuthing; in R, the Matrix library has data structures and functions that can do this; SciPy will have an equivalent.)

Benchmark the inversion method, your method, and the sparse method across some different scenarios (including different sparsity levels in X, e.g. 5% dense in my code above).

Sol: I tested the code across a range of scenarios for different sparse level, see the results below:

- $N = 2000, P = 1000$, sparse level = 5%

```
Unit: milliseconds
                 expr       min        lq      mean    median        uq       max neval
    chol_solver(y, X) 1392.5025 1521.6007 1723.5669 1862.6887 1893.534 1947.5086     5
 sparse_solver(y, Xs)  228.5724  298.1778  327.7172  318.7308  328.265  464.8403     5
```

- $N = 2000, P = 1000$, sparse level = 10%

```
Unit: milliseconds
                 expr       min        lq      mean    median        uq       max neval
    chol_solver(y, X) 1357.8523 1379.9926 1395.4460 1381.9588 1408.9545 1448.4719     5
 sparse_solver(y, Xs)  257.5167  262.7116  285.1333  263.4007  269.9196  372.1181     5
```

- $N = 2000, P = 1000$, sparse level = 25%

```
Unit: milliseconds
                 expr       min        lq      mean    median        uq      max neval
    chol_solver(y, X) 1364.2341 1386.2241 1428.6534 1411.3239 1416.6698 1564.815     5
 sparse_solver(y, Xs)  539.0973  540.9419  569.1402  551.5846  559.7251  654.352     5
```

- $N = 2000, P = 1000$, sparse level = 50%

```
Unit: seconds
                 expr      min       lq      mean   median       uq        max neval
    chol_solver(y, X) 1.361492 1.378877  1.644585 1.383409 2.014433   2.084712     5
 sparse_solver(y, Xs) 1.442227 1.448527 88.606565 2.245315 3.190310 434.706444     5
```

2 GENERALIZED LINEAR MODELS

As an archetypal case of a GLM, we'll consider the binomial logistic regression model: $y_i \sim \text{Binomial}(m_i, w_i)$, where y_i in an integer number of "successes," m_i is the number of trials for the ith case, and the success probability w_i is a regression on a feature vector x_i given by the inverse logit transform:

$$w_i = \frac{1}{1+\exp\{-x_i^T\beta\}}.$$

We want to estimate β by the principle of maximum likelihood. Note: for binary logistic regression, $m_i = 1$ and y_i is either 0 or 1.
As an aside, if you have a favorite data set or problem that involves a different GLM—say, a Poisson regression for count data—then feel free to work with that model instead throughout this entire section. The fact that we're working with a logistic regression isn't essential here; any GLM will do.

(A) Start by writing out the negative log likelihood,

$$l(\beta) = -\log\left\{\prod_{i=1}^{N} p(y_i \mid \beta)\right\}.$$

Simplify your expression as much as possible. This is the thing we want to minimize to compute the MLE. (By longstanding convention, we phrase optimization problems as minimization problems.)

Derive the gradient of this expression, $\nabla l(\beta)$. Note: your gradient will be a sum of terms $l_i(\beta)$, and it's OK to use the shorthand

$$w_i(\beta) = \frac{1}{1+\exp\{-x_i^T\beta\}}$$

in your expression.

Sol:

$$\begin{aligned}
l(\beta) &= -\log\left\{\prod_{i=1}^{N} p(y_i|\beta)\right\} \\
&\propto -\sum_{i=1}^{N} y_i \log\omega_i - \sum_{i=1}^{N}(m_i - y_i)\log(1-\omega_i) \quad \text{(ignored constanct in terms of } \beta\text{)} \\
&= -\sum_{i=1}^{N} y_i \log\left(\frac{1}{1+\exp\{-x_i^T\beta\}}\right) - \sum_{i=1}^{N}(m_i - y_i)\log\left(\frac{\exp\{-x_i^T\beta\}}{1+\exp\{-x_i^T\beta\}}\right) \\
&= \sum_{i=1}^{N} y_i \log\left(1+\exp\{-x_i^T\beta\}\right) - \sum_{i=1}^{N}(m_i - y_i)\left[-x_i^T\beta - \log\left(1+\exp\{-x_i^T\beta\}\right)\right] \\
&= \sum_{i=1}^{N}(m_i - y_i)x_i^T\beta + \sum_{i=1}^{N} m_i \log\left(1+\exp\{-x_i^T\beta\}\right)
\end{aligned}$$

$$
\begin{aligned}
\nabla l(\beta) &= \sum_{i=1}^{N}(m_i - y_i)x_i^T + \sum_{i=1}^{N} m_i \frac{-x_i^T \exp\{-x_i^T\beta\}}{1+\exp\{-x_i^T\beta\}} \\
&= \sum_{i=1}^{N}(m_i - y_i)x_i^T - \sum_{i=1}^{N} m_i x_i^T \frac{\exp\{-x_i^T\beta\}}{1+\exp\{-x_i^T\beta\}} \\
&= \sum_{i=1}^{N}(m_i - y_i)x_i^T - \sum_{i=1}^{N} m_i x_i^T (1-\omega_i) \\
&= \sum_{i=1}^{N} m_i x_i^T - \sum_{i=1}^{N} y_i x_i^T - \sum_{i=1}^{N} m_i x_i^T + \sum_{i=1}^{N} m_i x_i^T \omega_i \\
&= \sum_{i=1}^{N}(m_i\omega_i - y_i)x_i^T \\
&= X^T(\Omega M - Y) \quad \text{(in matrix notation)}
\end{aligned}
$$

(B) Read up on the method of steepest descent, i.e. gradient descent, in Nocedal and Wright (see course website). Write your own function that will fit a logistic regression model by gradient descent. Grab the data "wdbc.csv" from the course website, or obtain some other real data that interests you, and test it out. The WDBC file has information on 569 breast-cancer patients from a study done in Wisconsin. The first column is a patient ID, the second column is a classification of a breast cell (Malignant or Benign), and the next 30 columns are measurements computed from a digitized image of the cell nucleus. These are things like radius, smoothness, etc. For this problem, use the first 10 features for X, i.e. columns 3-12 of the file. If you use all 30 features you'll run into trouble.

Some notes here:

1. You can handle the intercept/offset term by either adding a column of 1's to the feature matrix X, or by explicitly introducing an intercept into the linear predictor and handling the intercept and regression coefficients separately, i.e.

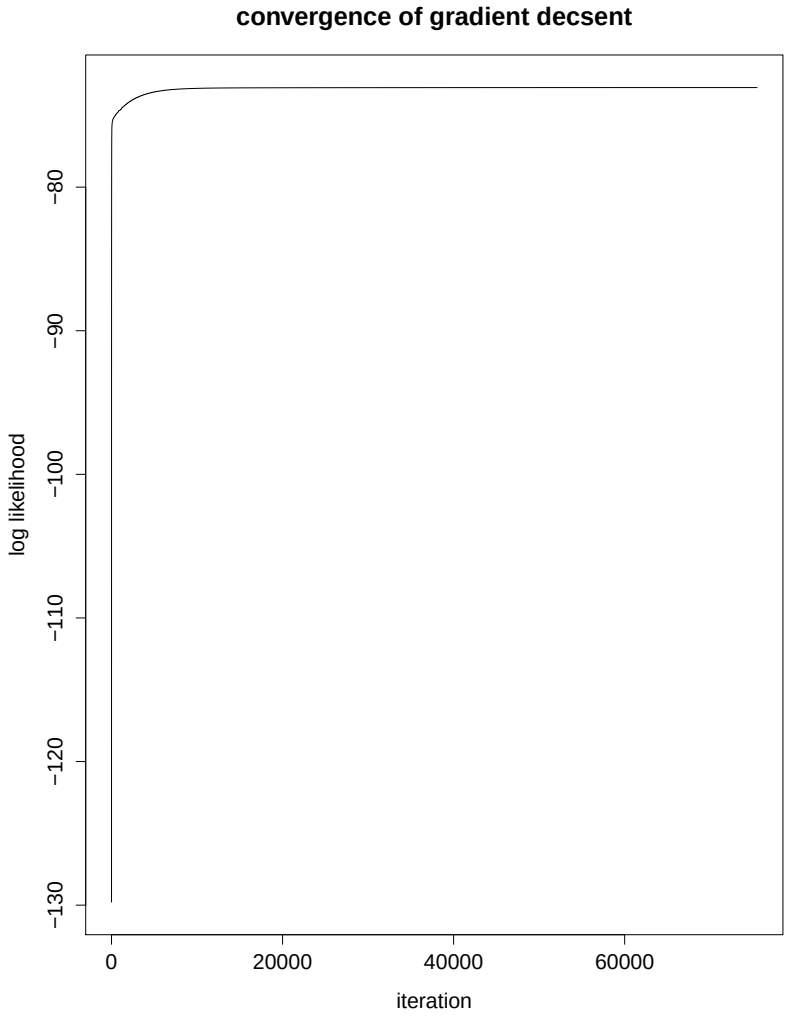
$$w_i(\beta) = \frac{1}{1+\exp\{-(\alpha + x_i^T\beta)\}}.$$

2. I strongly recommend that you write a self-contained function that, for given values of β, y, X, and sample sizes m_i (which for the WDBC data are all 1), will calculate the gradient of $l(\beta)$. Your gradient-descent optimizer will then call this function. Modular code is reusable code.

3. Make sure that, at every iteration of gradient descent, you compute and store the current value of the log likelihood, so that you can track and plot the convergence of the algorithm.

4. Be sensitive to the numerical consequences of an estimated success probability that is either very near 0, or very near 1.
5. Finally, you can be as clever as you want about the gradient-descent step size. Small step sizes will be more robust but slower; larger step sizes can be faster but may overshoot and diverge; step sizes based on line search (Chapter 3 of Nocedal and Wright) are cool but involve some extra work.

Sol: See graph of convergence for the log likelihood function:



(C) Now consider a point $\beta_0 \in \mathscr{R}^P$, which serves as an intermediate guess for our vector of regression coefficients. Show that the second-order Taylor approximation of $l(\beta)$, around the point β_0, takes the form

$$q(\beta;\beta_0) = \frac{1}{2}(z - X\beta)^T W(z - X\beta) + c$$

where z is a vector of "working responses" and W is a diagonal matrix of "working weights", and c is a constant that doesn't involve β. Give explicit expressions for the diagonal elements W_{ii} and for z_i (which will necessarily involve the point β_0, around which you're doing the expansion).

Sol:

$$\begin{aligned}\nabla l(\beta) &= \sum_{i=1}^{N}(m_i\omega_i - y_i)x_i^T = \sum_{i=1}^{N} m_i\omega_i x_i^T - y_i x_i^T \\ &= \sum_{i=1}^{N} m_i \frac{1}{1+\exp\{-x_i^T\beta\}} x_i^T\end{aligned}$$

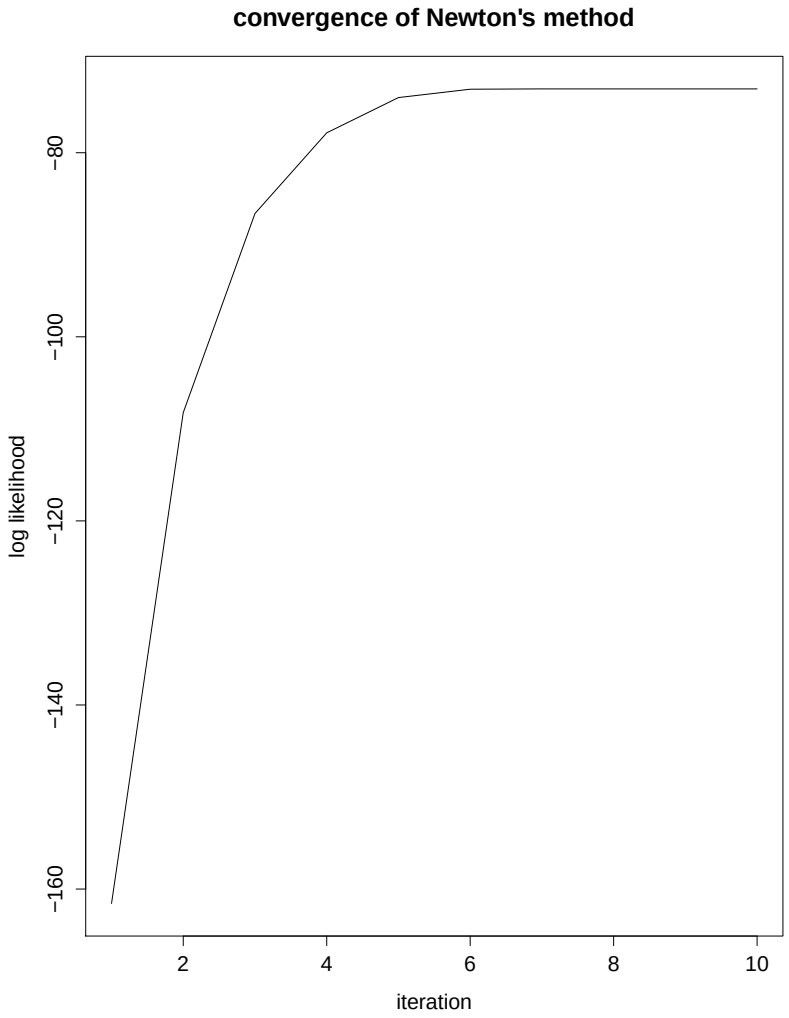
$$\begin{aligned}\mathrm{H}_{l(\beta)} &= \sum_{i=1}^{N} m_i \frac{x_i^T \exp\{-x_i^T\beta\}}{1+\exp\{-x_i^T\beta\}} x_i^T \\ &= \sum_{i=1}^{N} x_i \frac{m_i \exp\{-x_i^T\beta\}}{1+\exp\{-x_i^T\beta\}} x_i^T \\ &= \sum_{i=1}^{N} x_i \left[m_i\omega_i(1-\omega_i)\right] x_i^T \\ &= X^T W X \quad \text{(in matrix notation where W = diag}\left[m_i\omega_i(1-\omega_i)\right]\text{)}\end{aligned}$$

$$\begin{aligned}q(\beta;\beta_0) &= l(\beta_0) + \nabla l(\beta_0)^T(\beta-\beta_0) + \frac{1}{2}(\beta-\beta_0)^T \mathrm{H}_{l(\beta_0)}(\beta-\beta_0) \\ &= c + (\Omega(\beta_0)M - Y)^T X(\beta-\beta_0) + \frac{1}{2}(\beta-\beta_0)^T X^T W X(\beta-\beta_0) \\ &= c + (\Omega(\beta_0)M - Y)^T X\beta + \frac{1}{2}(X\beta)^T W(X\beta) - (X\beta_0)^T W(X\beta) \\ &= c + \frac{1}{2}(z - X\beta)^T W(z - X\beta) \\ \text{where } z &= \left[(\Omega(\beta_0)M - Y)^T - (X\beta_0)^T W\right] W^{-1}\end{aligned}$$

(D) Read up on Newton's method in Nocedal and Wright, Chapter 2. Implement it for the logit model and test it out on the same data set you just used to test out gradient descent.[2] Note: while you could do line search, there is a "natural" step size of 1 in Newton's method.

Sol: See graph of convergence for the log likelihood function:

[2] You should be able to use your own solver for linear systems from the first section.



Comparison of the Beta output for three different methods:

	glm	GD	NM
num of iter	0.00000000	75486.0000000	10.00000000
intercept	0.48701675	0.4798177	0.48701675
beta1	-7.22185053	-6.8540156	-7.22185053
beta2	1.65475615	1.6550242	1.65475615
beta3	-1.73763027	-2.0754897	-1.73763027
beta4	14.00484560	13.9468345	14.00484560
beta5	1.07495329	1.0715953	1.07495329
beta6	-0.07723455	-0.0575557	-0.07723455
beta7	0.67512313	0.6781015	0.67512313
beta8	2.59287426	2.5975101	2.59287426
beta9	0.44625631	0.4457612	0.44625631
beta10	-0.48248420	-0.4838200	-0.48248420

(E) Reflect broadly on the tradeoffs inherent in the decision of whether to use gradient descent or Newton's method for solving a logistic-regression problem.

Sol: Comparison of Gradient Descent verses Newton's Method:

- Gradient descent generally requires more iterations, but each iteration is fast because we only need to compute the 1st derivatives / gradient.
- Newton's method generally requires fewer iterations, but each iteration is slow because we need to compute 2nd derivatives / Hessian.
- Gradient descent needs one tuning parameter: the step size, while Newton's method has a natural step size of 1.
- Newton's method does not always work. It may diverge or the Hessian may be singular.
- Given real problems, no method is guaranteed to be faster than the others in every case. Advanced methods usually have larger overheads and computation costs in each iteration.