# Amin Anvari

SDS 385D - Statistical Models for Big Data
Assignment 1

## 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 $i$th row is $x_i$, and $e$ is a vector of model residuals. The goal is to estimate $\beta$, 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 $\beta$ by the principle of weighted least squares (WLS):

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

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.

Let's first write the (WLS) objective function in terms of vectors and matrices:

$$\hat{\beta} =_{\beta \in \mathbb{R}^p} \sum_{i=1}^{N} \frac{w_i}{2} \left(y_i - x_i^T \beta\right)^2 \tag{1}$$

$$=_{\beta \in \mathbb{R}^p} \frac{1}{2}(Y - X\beta)^T W (Y - X\beta). \tag{2}$$

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

Expanding the term in the objective function yielda

$$\begin{aligned}
\frac{1}{2}(Y - X\beta)^T W(Y - X\beta) &= \frac{1}{2}(Y^T - \beta^T X^T)W(Y - X\beta) && (3)\\
&= \frac{1}{2}(Y^T W - \beta^T X^T W)(Y - X\beta) && (4)\\
&= \frac{1}{2}(Y^T WY - \beta^T X^T WY - Y^T WX\beta + \beta^T X^T WX\beta) && (5)\\
&= \frac{1}{2}(Y^T WY - 2(X\beta)^T WY + \beta^T X^T WX\beta) && (6)\\
&= \frac{1}{2}Y^T WY - (X\beta)^T WY + \frac{1}{2}\beta^T X^T WX\beta, && (7)
\end{aligned}$$

To minimize the objective function we take the gradient with respect to $\beta$ and set it equal to zero.

(a)

$$\nabla_\beta(\frac{1}{2}Y^T WY) = 0 \tag{8}$$

(b)

$$\nabla_\beta(-(X\beta)^T WY) = -X^T WY \tag{9}$$

(c)

$$\begin{aligned}
\nabla_\beta(\frac{1}{2}\beta^T X^T WX\beta) &= \frac{1}{2}\beta^T(X^T WX + (X^T WX)^T) && (10)\\
&= X^T WX\beta. && (11)
\end{aligned}$$

We have to sum all these three terms and set it equal to zero:

$$X^T WX\beta - X^T WY = 0 \implies (X^T WX)\hat{\beta} = X^T WY. \tag{12}$$

(B) One way to calculate $\hat{\beta}$ is to: (1) recognize that, trivially, the solution to the above linear system must satisfy $\hat{\beta} = (X^T WX)^{-1}X^T Wy$; and (2) to calculate this directly, i.e. by inverting $X^T WX$. 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.)

solving Eqn. (12) using the inversion method yields:

$$\hat{\beta} = (X^T W X)^{-1} X^T W y. \tag{13}$$

Inverting a matrix in general is a no no! it is very expensive to invert a matrix in general. It is also not a stable method of solving linear systems. One alternate method based on matrix factorization is Cholesky Decomposition.

**Cholesky Decomposition Method:**
Let

$$C = X^T W X, \quad D = X^T W y \tag{14}$$

so

$$C\hat{\beta} = D. \tag{15}$$

We decompose matrix $C$ into a product of a lower-triangular matrix and an upper-triangular matrix, such that $U = L^T$ so

$$C = LU = LL^T \implies LL^T\hat{\beta} = D. \tag{16}$$

Furthermore we define matrix $A = L^T\hat{\beta}$. Thus we are left with two matrix equations to solve.

$$LA = D \tag{17}$$
$$L^T\hat{\beta} = A \tag{18}$$

Here both L and U are triangular and therefore solving these two equations are much less expensive. Therefore, this method is a much better way of solving the system by taking advantage of the symmetry of the matrix.

Cholesky decomposition method:

```
{
Compute $D = X^T W y$ ;
Compute $C = X^T W X$;
Compute $U$ as the upper triangular matrix from Cholesky decomposition
  of $C$;
Compute $L = U^T$;
Solve for $A$ in $LA = D$;
Solve for $\hat{\beta}$ in $L^T \hat{\beta} = U\hat{\beta} = A$;
return($\hat{\beta}$);
}
```
**Algorithm 1:** Cholesky Decomposition

(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.)

(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).

## 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 $i$th 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 $\beta$ 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.

We have $y_i \sim \text{Binomial}(m_i, w_i)$, where

$$w_i = \frac{1}{1 + \exp(-x_i^T \beta)}, \quad 1 - w_i = \frac{\exp(-x_i^T \beta)}{1 + \exp(-x_i^T \beta)}, \tag{19}$$

so the negative log likelihood is

$$\ell(\beta) = -\log\left\{\prod_{i=1}^{N} p(y_i|\beta)\right\} \tag{20}$$

$$= -\log\left\{\prod_{i=1}^{N} \binom{m_i}{y_i}(w_i)^{y_i}(1-w_i)^{m_i-y_i}\right\} \tag{21}$$

$$= -\left\{\sum_{i=1}^{N}\left(\log\binom{m_i}{y_i} + y_i\log(w_i) + (m_i-y_i)\log(1-w_i)\right)\right\} \tag{22}$$

$$= -\left\{\sum_{i=1}^{N}\left(\log\binom{m_i}{y_i} + y_i\log\left(\frac{1}{1+\exp(-x_i^T\beta)}\right) + (m_i-y_i)\log\left(\frac{\exp(-x_i^T\beta)}{1+\exp(-x_i^T\beta)}\right)\right)\right\} \tag{23}$$

$$= -\left\{\sum_{i=1}^{N}\left(\log\binom{m_i}{y_i} - y_i\log(1+\exp(-x_i^T\beta)) - (m_i-y_i)x_i^T\beta\right.\right. \tag{24}$$

$$- m_i\log(1+\exp(-x_i^T\beta)) + y_i\log(1+\exp(-x_i^T\beta)) \tag{25}$$

$$= -\left\{\sum_{i=1}^{N}\left(\log\binom{m_i}{y_i} - (m_i-y_i)x_i^T\beta - m_i\log(1+\exp(-x_i^T\beta))\right)\right\} \tag{26}$$

$$= \sum_{i=1}^{N}\left((m_i-y_i)x_i^T\beta + m_i\log(1+\exp(-x_i^T\beta)) - \log\binom{m_i}{y_i}\right) \tag{27}$$

The gradient for this expression is,

$$\nabla\ell(\beta) = \sum_{i=1}^{N}\left((m_i-y_i)x_i - m_i\frac{\exp(-x_i^T\beta)}{1+\exp(-x_i^T\beta)}x_i\right) \tag{28}$$

$$= \sum_{i=1}^{N}((m_i-y_i)x_i - m_i(1-w_i)x_i) \tag{29}$$

$$= \sum_{i=1}^{N}(m_iw_i - y_i)x_i \tag{30}$$

$$= -X^T(y-mw) \tag{31}$$

where $y$ is the $n\times 1$ vector of responses and $mw$ is the element-wise product of the two $n\times 1$ vectors $m$ and $w$.

(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:

(a) 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.

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

(b) I strongly recommend that you write a self-contained function that, for given values of $\beta$, $y$, $X$, 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.

(c) 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.

(d) Be sensitive to the numerical consequences of an estimated success probability that is either very near 0, or very near 1.

(e) 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.

(C) Now consider a point $\beta_0 \in \mathcal{R}^P$, 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 $\beta_0$, 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 $\beta$. Give explicit expressions for the diagonal elements $W_{ii}$ and for $z_i$ (which will necessarily involve the point $\beta_0$, around which you're doing the expansion).[2]

[2] Remember the trick of completing the square, e.g. `https://justindomke.wordpress.com/completing-the-square-in-n-dimensions/`.

We need to calculate the Hessian matrix of the log likelihood function, $\nabla^2 \ell(\beta)$. The Hessian will be a $P \times P$ matrix, with the element in row $i$ and column $j$ being:

$$\begin{aligned}
\frac{\partial^2}{\partial \beta_i \partial \beta_j} \ell(\beta) &= \frac{\partial}{\partial \beta_i} \left( \frac{\partial}{\partial \beta_j} \ell(\beta) \right) && (32) \\
&= \frac{\partial}{\partial \beta_i} \left( \frac{\partial}{\partial \beta_j} \sum_{k=1}^{N} \left[ (m_k - y_k) x_i^T \beta + m_k \log(1 + \exp(-x_k^T \beta)) - \log \binom{m_k}{y_k} \right] \right) && (33) \\
&= \frac{\partial}{\partial \beta_i} \left( \nabla_j \ell(\beta) \right) && (34) \\
&= \frac{\partial}{\partial \beta_i} \left( \sum_{k=1}^{N} (m_k w_k - y_k) x_{kj} \right) && (35) \\
&= \sum_{k=1}^{N} x_{ki} x_{kj} m_k w_k (1 - w_k) && (36)
\end{aligned}$$

Note:

$$\begin{aligned}
\frac{\partial}{\partial \beta_i} w_k &= x_{ki} \frac{\exp(-x_k^T \beta)}{(1 + \exp(-x_k^T \beta))^2} && (37) \\
&= x_{ki} w_k (1 - w_k) && (38)
\end{aligned}$$

This matrix is equivalent to $X^T W X$ where $W = \text{diag}(m_1 w_1 (1 - w_1), \ldots, m_N w_N (1 - w_N))$

Let $a = (y - mw)$. We have already shown that $\nabla \ell(\beta) = -X^T (y - mw) = -X^T a$ and $\nabla^2 \ell(\beta) = X^T W X$. The second-order Taylor approximation for $\ell(\beta)$ around the point $\beta_0$ is,

$$\begin{aligned}
\hat{\ell}(\beta) &= \ell(\beta_0) + (\nabla\ell(\beta))^T(\beta - \beta_0) + \frac{1}{2}(\beta - \beta_0)^T\nabla^2\ell(\beta)(\beta - \beta_0) && (39)\\
&= \ell(\beta_0) + (-X^Ta)^T(\beta - \beta_0) + \frac{1}{2}(\beta - \beta_0)^TX^TWX(\beta - \beta_0) && (40)\\
&= \frac{1}{2}([\beta - \beta_0] - (X^TWX)^{-1}X^Ta)^TX^TWX([\beta - \beta_0] - (X^TWX)^{-1}X^Ta) + c && (41)\\
&= \frac{1}{2}(\beta - \beta_0 + X^{-1}W^{-1}(X^T)^{-1}X^Ta)^TX^TWX(\beta - \beta_0 + X^{-1}W^{-1}(X^T)^{-1}X^Ta) + c && (42)\\
&= \frac{1}{2}(\beta - \beta_0 + X^{-1}W^{-1}a)^TX^TWX(\beta - \beta_0X^{-1}W^{-1}a) + c && (43)\\
&= \frac{1}{2}(X\beta - X\beta_0 + XX^{-1}W^{-1}a)^TW(X\beta - X\beta_0 + XX^{-1}W^{-1}a) + c && (44)\\
&= \frac{1}{2}(X\beta - X\beta_0 + W^{-1}a)^TW(X\beta - X\beta_0 + W^{-1}a) + \ldots && (45)\\
&= \frac{1}{2}(z - X\beta)^TW(z - X\beta) + c, && (46)
\end{aligned}$$

where $c$ is some constant, $z = X\beta_0 + W^{-1}a = X\beta_0 + W^{-1}(y - mw)$

(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.[3] Note: while you could do line search, there is a "natural" step size of 1 in Newton's method.

Now we use Newton's to estimate $\beta$. This is also an iterative process, though now we need far fewer iterations to achieve convergence because we are taking the curvature of our objective function ($\ell(\beta)$) into account. In fact, we only use 10 iterations and achieve estimates $\hat{\beta}$ which are *exactly* in line with estimates from `glm`.
**Newton's Method:**

$$\hat{\beta}_{t+1} = \hat{\beta}_t - (\nabla^2\ell(\hat{\beta}_t))^{-1}\nabla\ell(\hat{\beta}_t) \quad (47)$$

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

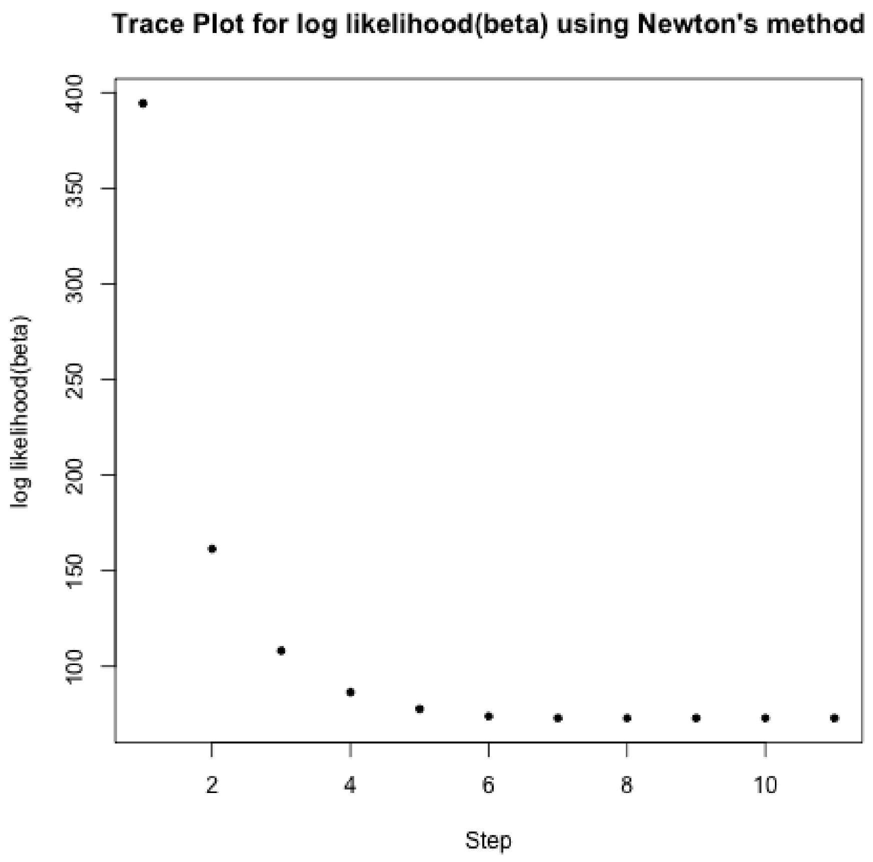

Figure 1: Log likelihood trace plot for Newton's method

| | N.'s method | R: `glm` |
|---|---:|---:|
| $\hat{\beta}_1$ | 0.48702 | 0.48702 |
| $\hat{\beta}_2$ | -7.22185 | -7.22185 |
| $\hat{\beta}_3$ | 1.65476 | 1.65476 |
| $\hat{\beta}_4$ | -1.73763 | -1.73763 |
| $\hat{\beta}_5$ | 14.00485 | 14.00485 |
| $\hat{\beta}_6$ | 1.07495 | 1.07495 |
| $\hat{\beta}_7$ | -0.07723 | -0.07723 |
| $\hat{\beta}_8$ | 0.67512 | 0.67512 |
| $\hat{\beta}_9$ | 2.59287 | 2.59287 |
| $\hat{\beta}_{10}$ | 0.44626 | 0.44626 |
| $\hat{\beta}_{11}$ | -0.48248 | -0.48248 |

Figure 2: Comparison of results from Newton's method and `glm`

(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.

Gradient descent requires many iterations, and there is no simple way to know what step size is wise, or even to know just how many iterations are sufficient to reach convergence. The only way to find out is to experiment with different numbers of iterations and step sizes. In contrast, Newton's method converges upon the MLE with far fewer iterations because we are using knowledge of the "curvature" of our function to be minimized. However, for Newton's method we must invert the Hessian matrix, which is computationally intensive for large matrices, and potentially unstable, leaving us vulnerable to floating point errors. It also possible that the Hessian matrix has no closed form solution or is so convoluted as to make its computation difficult.