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):

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

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,

$$\sum_{i=1}^{N} \frac{w_i}{2}(y_i - x_i^T\beta)^2 = \frac{1}{2}(y - X\beta)^T W (y - X\beta)$$

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.

First derivative to find the minimum:

$$\begin{aligned}
0 &= \frac{d}{d\beta}\frac{1}{2}(y - X\beta)^T W (y - X\beta) \\
&= \frac{d}{d\beta}\frac{1}{2}\left(y^T W y - 2\beta^T X^T W y + \beta^T X^T W X \beta\right) \\
&= \frac{1}{2}\left(0 - 2X^T W y + 2X^T W X \hat{\beta}\right) \\
&= -X^T W y + X^T W X \hat{\beta} \\
\Rightarrow X^T W X \hat{\beta} &= X^T W y
\end{aligned}$$

Second derivative to prove it's a minimum (and not a maximum or saddle point):

$$\begin{aligned}
\frac{d^2}{d\beta^2}\frac{1}{2}(y - X\beta)^T W (y - X\beta) &= \frac{d}{d\beta}\left(-X^T W y + X^T W X \beta\right) \\
&= X^T W X
\end{aligned}$$

This is positive definite, if $(W^{1/2}X)$ is full rank, positive semidefinite otherwise.

(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.

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

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,

Wikipedia (https://en.wikipedia.org/wiki/System_of_linear_equations) states that there are many methods, but a modification of Gaussian elimination (row reduction) using the LU decomposition is the common method used within computation. So no, inversion is not the fastest method, which makes sense considering that's the advice I've always received in my coding, to avoid inverting matrices if possible.

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.)

http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/INT-APP/CURVE-linear-system.html is really helpful here, though I'll say that I figured out the psuedocode for myself and didn't copy their's. They state the idea: to solve $B = AX$ for unknown X, instead of inverting A, take the LU decomposition instead to get $B = LUX$, where L is lower triangular and U is upper triangular. Then for unknown $Y = UX$, solving the linear system $B = LY$ for Y is straightforward with triangular L. Once Y is found, solving $Y = UX$ is likewise simple with triangular U.

Psuedocode for our problem using "my" method, which does require some nice behavior, namely $(W^{1/2}X)$ to be full rank:
1. LU decomposition of (X^TWX).
2. Let $U\hat{\beta} = \delta$ and $X^TWy = q$. Solve $q = L\delta$ for δ by looping over j from 1 to p:

$$\delta_j = \frac{q_j - \sum_{i<j} l_{ji}\delta_i}{l_{jj}}.$$

3. Solve $\delta = U\hat{\beta}$ for $\hat{\beta}$ by looping over j from p to 1:

$$\hat{\beta}_j = \frac{\delta_j - \sum_{i>j} u_{ji}\hat{\beta}_i}{u_{jj}}.$$

****

Apparently, the QR decomposition is preferable. The general idea is to decompose $W^{1/2}X = QR$, where Q has orthonormal columns and R is square and upper triangular. Then:

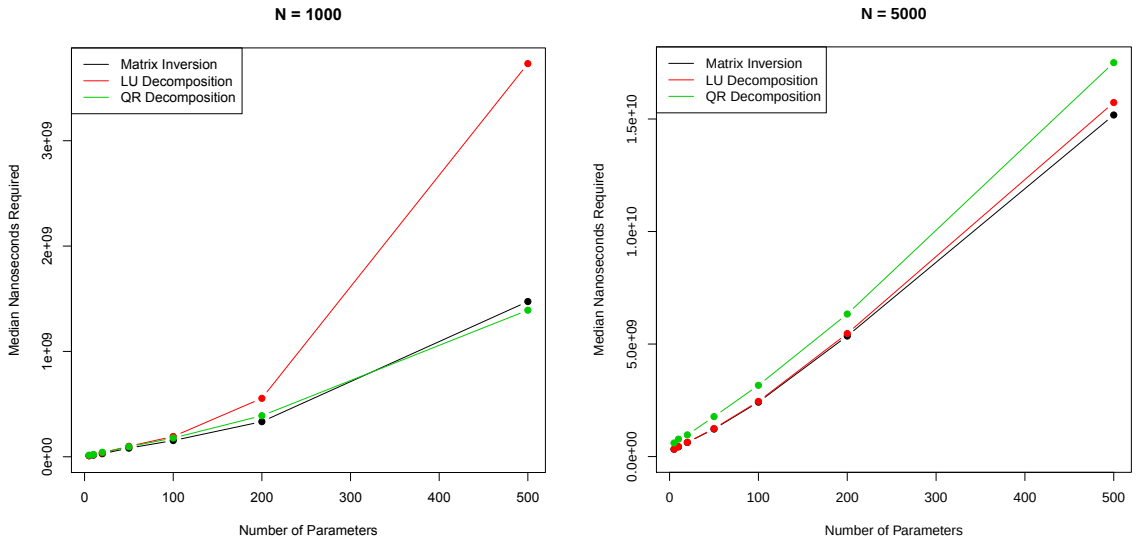
$$\begin{aligned}
X^TWX\hat{\beta} &= X^TWy \\
X^TW^{1/2}W^{1/2}X\hat{\beta} &= X^TW^{1/2}W^{1/2}y \\
(QR)^TQR\hat{\beta} &= (QR)^TW^{1/2}y \\
R^TQ^TQR\hat{\beta} &= R^TQ^TW^{1/2}y \\
R^TIR\hat{\beta} &= R^TQ^TW^{1/2}y \\
(R^T)^{-1}R^TR\hat{\beta} &= (R^T)^{-1}R^TQ^TW^{1/2}y \\
R\hat{\beta} &= Q^TW^{1/2}y
\end{aligned}$$

and this can easily be solved using Gaussian elimination, as R is triangular.

(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.

For coded functions, see accompanying R file. Note, I have not yet learned how to be "attentive" about diagonal matrix multiplication. My only ideas for such a workaround involve more loops, which would be worse, right?

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.)



It appears that my coding of the functions that use decompositions was subpar: they are defeated by the inversion method. However, as N increases the LU decomposition becomes comparable to inversion.

Notes from James:
High level understanding: don't invert a big data matrix unless you have to. In R, use solve(A,b) instead of $x = A^{-1}b$.
Intermediate: Factorize with discussed decomposition **this is our target
Low: details of how factorization work, what pivoting matrix to choose, etc.

(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:

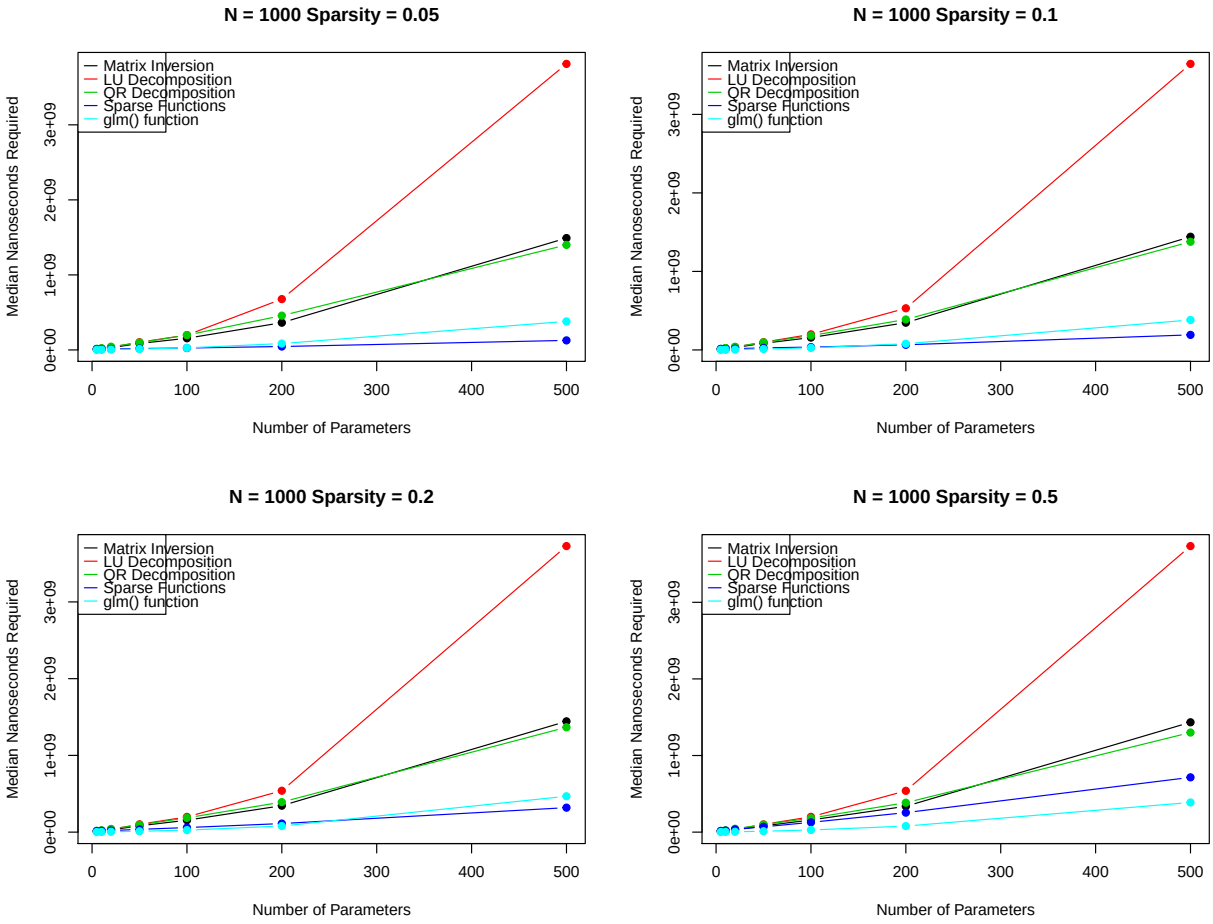
```
N = 2000
P = 500
X = matrix(rnorm(N*P), nrow=N)
mask = matrix(rbinom(N*P,1,0.05), nrow=N)
X = mask*X
X[1:10, 1:10]  # quick visual check
```

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).

Below are the run times required for varying methods, number of parameters, and data sparsity. Please forgive the legend box size, R would not cooperate.



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.)

$$\begin{aligned} l(\beta) &= -\log\left\{\prod_{i=1}^{N} p(y_i \mid \beta)\right\} \\ &= -\log\left\{\prod_{i=1}^{N} \binom{m_i}{y_i} w_i^{y_i}(1-w_i)^{m_i-y_i}\right\} \\ &= -\sum_{i=1}^{N}\left\{\log\binom{m_i}{y_i} + y_i\log(w_i) + (m_i-y_i)log(1-w_i)\right\} \end{aligned}$$

Note: thanks Spencer for the blackboard clarification.

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.

$$\begin{aligned} \nabla_\beta w_i &= \nabla_\beta\left(\frac{1}{1+\exp\{-x_i^T\beta\}}\right) \\ &= -\frac{\exp\{-x_i^T\beta\}(-x_i)}{(1+\exp\{-x_i^T\beta\})^2} \\ &= \frac{1}{(1+\exp\{-x_i^T\beta\})}\frac{\exp\{-x_i^T\beta\}}{(1+\exp\{-x_i^T\beta\})}(x_i) \\ &= w_i(1-w_i)x_i \end{aligned}$$

$$
\begin{aligned}
\nabla_\beta l(\beta) &= -\nabla_\beta \sum_{i=1}^{N} \left\{ \log \binom{m_i}{y_i} + y_i \log(w_i) + (m_i - y_i) log(1 - w_i) \right\} \\
&= -\sum_{i=1}^{N} \{0 + y_i \nabla_\beta \log(w_i) + (m_i - y_i) \nabla_\beta log(1 - w_i)\} \\
&= -\sum_{i=1}^{N} \left\{ y_i \frac{1}{w_i} \nabla_\beta w_i + (m_i - y_i) \frac{1}{1 - w_i} \nabla_\beta (1 - w_i) \right\} \\
&= -\sum_{i=1}^{N} \left\{ y_i \frac{1}{w_i} \nabla_\beta w_i - (m_i - y_i) \frac{1}{1 - w_i} \nabla_\beta w_i \right\} \\
&= -\sum_{i=1}^{N} \left\{ y_i \frac{1}{w_i} w_i (1 - w_i) x_i - (m_i - y_i) \frac{1}{1 - w_i} w_i (1 - w_i) x_i \right\} \\
&= -\sum_{i=1}^{N} \{ y_i (1 - w_i) x_i - (m_i - y_i) w_i x_i \} \\
&= -\sum_{i=1}^{N} \{ y_i x_i - y_i w_i x_i - m_i w_i x_i + y_i w_i x_i \} \\
&= -\sum_{i=1}^{N} \{ (y_i - m_i w_i) x_i \} \qquad \text{or } (y_i - \hat{y}_i) \text{ as James pointed out} \\
&= -X^T (y - \hat{y}) \\
&= X^T (\hat{y} - y)
\end{aligned}
$$

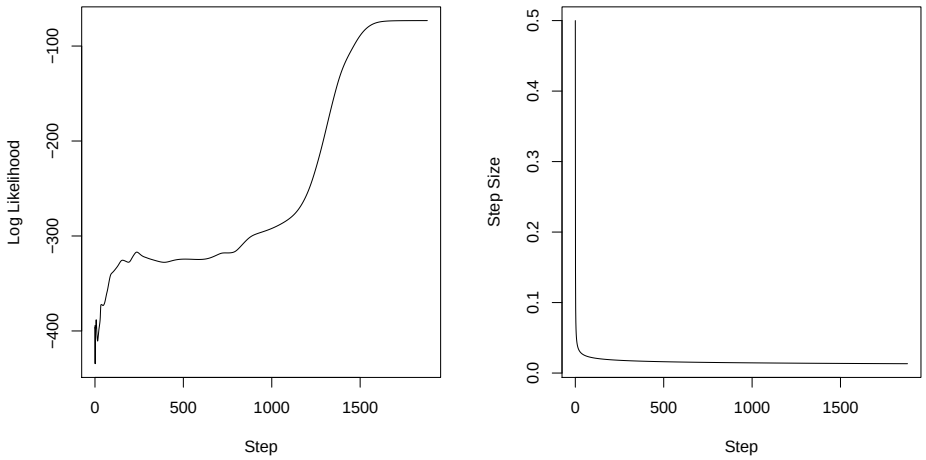
Note: thanks Spencer for the blackboard clarification.

(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.

	Intercept	β_1	β_2	β_3	β_4	β_5	β_6	β_7	β_8	β_9	β_{10}
Mine	0.234	-2.834	1.645	-3.747	11.120	1.015	-0.122	0.727	2.653	0.451	-0.389
R's glm()	0.487	-7.222	1.655	-1.738	14.005	1.075	-0.077	0.675	2.593	0.446	-0.482



(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 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).

The trick[2] for completing the square from a quadratic form is

$$a + b^T x + \frac{1}{2}x^T C x = \frac{1}{2}(x + C^{-1}b)^T C(x + C^{-1}b) + a - \frac{1}{2}b^T C^{-1}b. \tag{1}$$

In addition to these previous calculations, to compute the 2nd order Taylor approximation, we will need

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

the Hessian matrix:

$$
\begin{aligned}
H\{l(\beta)\}_{jk} &= \frac{\partial^2}{\partial\beta_k\partial\beta_j} - \sum_{i=1}^{N}\left\{\log\binom{m_i}{y_i} + y_i\log(w_i) + (m_i - y_i)log(1-w_i)\right\} && \text{jk or kj?}\\
&= -\frac{\partial}{\partial\beta_k}\sum_{i=1}^{N}\{(y_i - m_i w_i)x_{ij}\} && \text{element from } \nabla_\beta l(\beta)\\
&= -\frac{\partial}{\partial\beta_k}\sum_{i=1}^{N}\{y_i x_{ij} - m_i w_i x_{ij}\}\\
&= -\sum_{i=1}^{N}\left\{0 - m_i x_{ij}\frac{\partial}{\partial\beta_k}w_i\right\}\\
&= \sum_{i=1}^{N}\{m_i x_{ij}\left[w_i(1-w_i)x_{ik}\right]\} && \text{element from } \nabla_\beta w_i\\
&= \sum_{i=1}^{N} x_{ik}x_{ij}\left[m_i w_i(1-w_i)\right]
\end{aligned}
$$

$$\Rightarrow H\{l(\beta)\} = X^T V X, \text{ where } V_{ii} = m_i w_i(1-w_i) \text{ are diagonal.}$$

Note that V is a diagonal matrix of binomial variances! Also, V_0, W_0 use β_0 in their formulation, where $W_{0ii} = w_i$. Let M be a diagonal matrix of all m_i. Then the approximation is

$$
\begin{aligned}
l(\beta) &\approx q(\beta;\beta_0)\\
&= l(\beta_0) + \nabla_\beta l(\beta_0)^T(\beta-\beta_0) + \frac{1}{2}(\beta-\beta_0)^T H\{l(\beta_0)\}(\beta-\beta_0)\\
&= l(\beta_0) - (y - (MW_0)\mathbf{1})^T X(\beta-\beta_0) + \frac{1}{2}(\beta-\beta_0)^T X^T V X(\beta-\beta_0)\\
&= l(\beta_0) - (y - MW_0\mathbf{1})^T(X\beta - X\beta_0) + \frac{1}{2}(X\beta - X\beta_0)^T V(X\beta - X\beta_0)
\end{aligned}
$$

and note this last expression fits the quadratic form from Equation 1 with:

$$
\begin{aligned}
x &= (X\beta - X\beta_0)\\
a &= l(\beta_0)\\
b &= -(y - MW_0\mathbf{1}) = -(y - \hat{y}_0)\\
C &= V = (I - W_0)MW_0.
\end{aligned}
$$

Substituting into said quatratic form,

$$
\begin{aligned}
&\Rightarrow l(\beta) \approx q(\beta;\beta_0)\\
&= l(\beta_0) - (y - (MW_0)\mathbf{1})^T(X\beta - X\beta_0) + \frac{1}{2}(X\beta - X\beta_0)^T(I-W_0)MW_0(X\beta - X\beta_0)\\
&= a + b^T x + \frac{1}{2}x^T C x\\
&= \frac{1}{2}(x + C^{-1}b)^T C(x + C^{-1}b) + a - \frac{1}{2}b^T C^{-1}b\\
&= \frac{1}{2}\left\{X\beta - X\beta_0 - V^{-1}(y-\hat{y}_0)\right\}^T V\left\{X\beta - X\beta_0 - V^{-1}(y-\hat{y}_0)\right\} + a - \frac{1}{2}(y-\hat{y}_0)^T V^{-1}(y-\hat{y}_0)\\
&= \frac{1}{2}\left\{X\beta - \left[X\beta_0 + V^{-1}(y-\hat{y}_0)\right]\right\}^T V\left\{X\beta - \left[X\beta_0 + V^{-1}(y-\hat{y}_0)\right]\right\} + a - \frac{1}{2}(y-\hat{y}_0)^T V^{-1}(y-\hat{y}_0)\\
&= \frac{1}{2}\left\{\left[X\beta_0 + V^{-1}(y-\hat{y}_0)\right] - X\beta\right\}^T V\left\{\left[X\beta_0 + V^{-1}(y-\hat{y}_0)\right] - X\beta\right\} + a - \frac{1}{2}(y-\hat{y}_0)^T V^{-1}(y-\hat{y}_0)\\
&= \frac{1}{2}(z - X\beta)^T W(z - X\beta) + c
\end{aligned}
$$

which is the desired form, with

$$z = X\beta_0 + V^{-1}(y - \hat{y}_0)$$
$$c = a - \frac{1}{2}(y - \hat{y}_0)^T V^{-1}(y - \hat{y}_0)$$
$$W = V$$

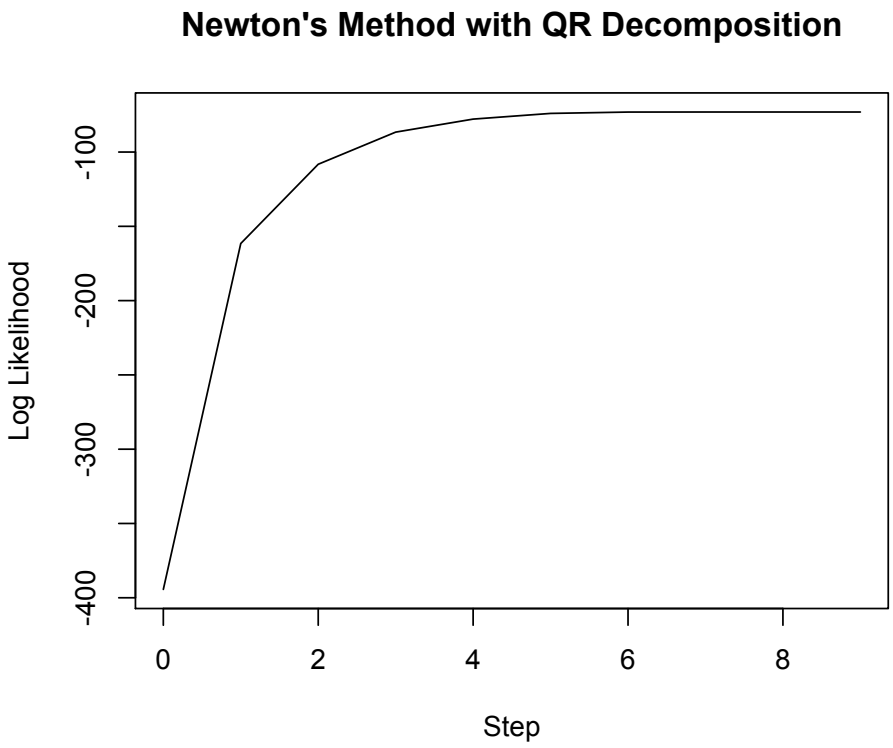
with $W_{ii} = V_{ii} = m_i w_i(1 - w_i)$.

(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. [4]

In part C we showed the math equating a step in Newton's method here is the same as iteratively resolving weighted least squares. However, here we use a QR decomposition of the Hessian matrix to solve the linear system for the step size of β, namely $\Delta\beta H = \nabla_\beta$.

	Intercept	β_1	β_2	β_3	β_4	β_5	β_6	β_7	β_8	β_9	β_{10}
My NM/QR	0.487	-7.222	1.655	-1.738	14.005	1.075	-0.077	0.675	2.593	0.446	-0.482
R's glm()	0.487	-7.222	1.655	-1.738	14.005	1.075	-0.077	0.675	2.593	0.446	-0.482

As seen above, my Newton's method is as precise as R's glm() when rounded up to the 3rd decimal place (it does diverge at the 4th decimal). The plot below shows Newton's method converging in 9 steps.



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

[4] http://ocw.mit.edu/courses/sloan-school-of-management/15-084j-nonlinear-programming-spring-2004/lecture-notes/lec3_newton_mthd.pdf

(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.

Newton's method steps much faster, but requires the calculation then inversion/decomposition of a matrix. Gradient descent can struggle with step size and thus take thousands of iterations, but is your step size is smart, it may be preferable to the calculations for inverting or decomposing the Hessian matrix.