SDS 385 Exercise Set 01

Kevin Song

September 12, 2016

A note to the reviewer: I discovered the ℓ symbol halfway through this document but really didn't want to go back and replace all the previous usages. As such, both l and ℓ appear in this document, and they're supposed to be the same symbol. Sorry, and I hope it isn't too distracting.

1 WLS Objective Function

The weighted least squares problem is described by the equation

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

where w_i is the weight of the i-th observation, y_i is the i-th response, and x_i is the i-th row of X, an $N \times P$ matrix.

We can rewrite the objective function in a matrix-vector form in a few steps. First, we note that this is a dot product between the vectors w and $y - X\beta$:

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

Unfortunately, this isn't quite valid, since the exponent denotes an element-wise square, which is something that cannot be done with standard vector operations.

Instead, we convert to quadratic form:

$$\frac{1}{2} w^T (y - X\beta)^2 = \frac{1}{2}(y - X\beta)^T W (y - X\beta)$$

where W is the diagonal matrix with the entries of w along its diagonal.

While I haven't made a formal proof of correctness of this transformation, I offer the following validity argument: define $p = y - X\beta$. If we say that the diagonal elements of W are the elements w, doing the second multiplication first yields the vector q, where $q_i = w_i p_i$. Then we get that the end result is $\sum_i p_i w_i p_i$, which is what we wanted.

Expanding this product, we find that

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

In this expansion, we find that the first term is useless to us: since we change the value the objective function takes by altering β, and β is not in this term, it does not affect the optimization. We throw it out.

The second and third terms are transposes of each other. As luck would have it, they are also scalar terms. Since $a = a^T$ when a is a scalar, we combine the two terms.

Finally, we recognize that the final term is the quadratic form of β with the matrix $X^T W X$.

This gives us the following objective function:

$$f(\beta) = (y^T W X)\beta + \frac{1}{2}\beta^T X^T W X \beta$$

Note that this is a quadratic form, $\frac{1}{2}x^T Ax + b^T x + c$, with $A = X^T W X$, $b = y^T W X$, and $c = 0$.

We know that quadratic forms are minimized by $Ax - b = 0$, or $Ax = b$. Thus, we conclude that $\hat{\beta}$ can be found by solving

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

or equivalently, by

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

2 Solutions to the minimization

The inversion method is a potentially awful way to solve the system. If the entries of X are all relatively similar to each other and the confidence values differ wildly, the computed inverse may not be the inverse at all! More generally, if the condition number of $X^T W X$ is large, then $X^{-1}X$ might not be equal to the identity matrix!

Since we cannot always use matrix inversion, we can instead use a matrix factorization method. In this case, we choose an LU method because I'm not sure if Choleky's conditions will hold here:

```
[L,U] = LU factor(X^T W X)
z = X^T W y
c = triangle solve(Lc = z)
x = triangle solve(Ux = b)
β̂ = x
```

3 Code

Code for solving with inversion:

```
## This solves (X^T W X) B = X^T W y by using matrix inversion on the LHS
## Notation: I treat this as an Ax = b problem, so
##    A = X^T W X
##    b = X^T W y
##    x = beta

import numpy as np
import scipy as sp
import random

def solve(A,b):

    solution = A.I * b
```

```
    return solution
```

inversionsolve.py

Code for solving with LU factorization:

```
## This solves (X^T W X) B = X^T W y by using matrix factorization on the LHS
## Notation: I treat this as an Ax = b problem, so
##   A = X^T W X
##   b = X^T W y
##   x = beta

import numpy as np
import scipy as sp
from scipy import linalg
import random

def solve(A,b):

    LUFact = linalg.lu_factor(A)
    solution = linalg.lu_solve(LUFact, b)

    return solution
```

factorsolve.py

Unfortunately, due to the limitations of my current system (a celeron with 8GB of memory), I was not able to do particularly large tests, since even relatively small timing tests took up a significant amount of time.

The results for the three tests I did manage to finish are shown below. These were done using iPython3's `%timeit` magic. The elements of the matrices are generated random uniform in $[0, 50)$.

Rows	Columns	Iterations	Factorization	Inversion
400	20	1000	2.44	2.54
2000	40	100	18.2	19.7
2000	200	100	33.3	36.3

Table 1: Timings for factorization and inverse-based solutions in python. The times are for a single solve, averaged over the number of iterations (e.g. 5s with 100 iterations means the run took 500 seconds total). All times are in seconds.

4 Sparse Matrices

Converting the code to use sparse matrix representation is pretty simple to do with python. Simply use the sparse solver provided by `scipy.sparse.linalg` to solve the problem. Random matrices can be generated with given sparsity levels with `scipy.sparse.random` Stacking this solver up against the previous solver, we get the following values:

You may notice something interesting about these numbers: they make no sense. The sparse solver is supposed to be faster than the dense solver (which is indeed the case in R), but it's slower here by a huge margin. This can be partially explained by Issue #3831 [1], which notes that SuperLU is over 200 times slower than its competitor UMFPack (which is used by the R Matrix package).

Unfortunately, I have not had the time to recompile scipy against UMFPack to test the difference.

[1] https://github.com/scipy/scipy/issues/3831

Rows	Columns	Sparsity	Sparse Solver (ms)	Inversion (ms)	Factorization (ms)
4000	200	0.05	1.63	1.15	0.58
4000	400	0.05	13.3	4.33	2.05
4000	800	0.05	65.1	23.4	10.9
4000	1600	0.05	383	151	52.4
4000	200	0.15	1.77	1.15	0.58
4000	400	0.15	14	4.17	2.02
4000	800	0.15	68.4	23.5	10.2
4000	1600	0.15	394	157	50.9

Table 2: Timings for factorization and inverse-based, as well as sparse solver solutions in python. The non-sparse solvers were run by first converting the sparse matrix representation to a dense one, then using the routines provided by the previous test. All tests were done over 100 iterations.

5 MLE

The MLE form is:

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

Again, since the binomial coefficient is not important in the context of optimizing this function over β, we ignore it.

This gives us the final equation

$$-l(\beta) = y_i\sum_{i=1}^{N}\log w_i + (m_i-y_i)\sum_{i=1}^{N}\log(1-w_i)$$

If we define $l_i(\beta)$ as

$$-l_i(\beta) = \sum_{i=1}^{N}\log w_i - \sum_{i=1}^{N}\log(1-w_i)$$

we can express $l(\beta) = \sum l_i(\beta)$. Then to find the gradient of l, we need only find the gradient of each l_i.

$$-\nabla l_i(\beta) = \nabla_\beta \log w_i + \nabla_\beta \log(1-w_i)$$

We note that $\nabla_\beta l_i = (w_i)(1-w_i)x_i$. This allows us to apply the chain rule to the above form, yielding

$$\begin{aligned}
-\nabla l_i(\beta) &= y_i \nabla_\beta \log w_i + (m_i - y_i)\nabla_\beta \log(1-w_i) \\
&= y_i \frac{1}{w_i}\nabla_\beta w_i + (m_i - y_i)\frac{1}{1-w_i}\nabla_\beta(1-w_i) \\
&= y_i \frac{w_i(1-w_i)}{w_i}x_i - (m_i - y_i)\frac{w_i(1-w_i)}{1-w_i}x_i \\
&= y_i(1-w_i)x_i - (m_i - y_i)w_i x_i \\
&= (y_i - y_i w_i - m_i w_i + y_i w_i)x_i \\
&= (y_i - m_i w_i)x_i
\end{aligned}$$

Finally, this gives us that the gradient of the entire MLE function is

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

6 Steepest Descent Code

This code has one major quirk: instead of evaluating the gradient and log-likelihood at β_i in a single function, a generator is used to produce a gradient/likelihood function beforehand. This function is then used to compute the desired quantities.

Due to length issues, the steepest descent code is not listed in this document. It can be found at `steepestdescent.py`

7 Quadratic Forms Again

Note: This derivation follows the general framework of the derivation given by Matteo Vestrucci.

Earlier, we showed that the gradient of $\ell(\beta;\beta_0)$ could be written as

$$-\nabla \ell_i(\beta) = \sum_i (y_i - m_i w_i)x_i$$

Since the i,jth entry of H is given by $H_{i,j} = \frac{\partial}{\partial\beta_i}\frac{\partial}{\partial\beta_j}\ell(\beta)$, we can get the i,jth component by taking the derivative of the ith component of the gradient with respect to the jth variable.

This, in turn, says that we can express the i,jth entry of the Hessian as

$$\begin{aligned}
H_{i,j} &= \frac{\partial}{\partial\beta_j}\sum_i (m_i w_i x_i - y_i x_i) \\
&= \frac{\partial}{\partial\beta_j}\sum_i \left(m_i \frac{1}{1+\exp(-x_i^T\beta)}x_i - y_i x_i\right) \\
&= m_i x_{i,j}\frac{\partial}{\partial\beta_j}\left(\frac{1}{1+\exp(-x_i^T\beta)}\right) \\
&= m_i x_{i,j}(-x_{j,i})\frac{-\exp(-x_i^T\beta)}{(1+\exp(-x_i^T\beta))^2} \\
&= m_i x_{i,j}x_{j,i}w_i(1-w_i)
\end{aligned}$$

This can be expressed as $H = X^T DX$ where $D_{i,i} = m_i w_i (1 - w_i)$.

Since the diagonal values are all positive, the Hessian is an SPD matrix. We proceed by using this form to complete the square.

The second-order Taylor approximation of $l(\beta)$ about β_0 is given by Taylor's Theorem:

$$\begin{aligned} q(\beta; \beta_0) &\approx l(\beta_0) + \nabla l(\beta_0)^T (\beta - \beta_0) + \frac{1}{2}(\beta - \beta_0)^T \nabla^2 l(\beta_0)(\beta - \beta_0) \\ &= l(\beta_0) + \nabla l(\beta_0)^T (\beta - \beta_0) + \frac{1}{2}(\beta - \beta_0)^T H_l(\beta_0)(\beta - \beta_0) \\ &= \left(l(\beta_0) - \nabla l(\beta_0)^T \beta_0 + \frac{1}{2}\beta_0^T H_l(\beta_0)\beta_0 \right) \\ &+ \nabla l(\beta_0)^T \beta - \beta_0^T H_l(\beta_0)\beta + \frac{1}{2}\beta^T H_l(\beta_0)\beta \end{aligned}$$

Since we seek to optimize this expression, we can roll all of the constant terms into one term c. This gives us

$$q(\beta; \beta_0) = c + \left(\nabla \ell(\beta_0)^T - \beta_0^T H_l(\beta_0) \right) \beta + \frac{1}{2}\beta^T H_l(\beta_0)\beta$$

Replacing this with our formulation for H, we get

$$\begin{aligned} q(\beta; \beta_0) &= c + \left(\nabla \ell(\beta_0)^T - \beta_0^T H_l(\beta_0) \right) \beta + \frac{1}{2}\beta^T H_l(\beta_0)\beta \\ &= c + \left(\nabla \ell(\beta_0)^T - \beta_0^T X^T DX \right) \beta + \frac{1}{2}\beta^T X^T DX\beta \end{aligned}$$

We have also expressed the gradient as $\nabla \ell_i(\beta) = \sum_i (y_i - m_i w_i) x_i$, or $\nabla \ell(\beta) = X^T s$ where $s_i = y_i - m_i w_i$. We can use this to further simplify the above:

$$\begin{aligned} q(\beta; \beta_0) &= c + \left(\nabla \ell(\beta_0)^T - \beta_0^T X^T DX \right) \beta + \frac{1}{2}\beta^T X^T DX\beta \\ &= c + \left(s^T X - \beta_0^T X^T DX \right) \beta + \frac{1}{2}\beta^T X^T DX\beta \\ &= c + \left(s^T - \beta_0^T X^T D \right) X\beta + \frac{1}{2}\beta^T X^T DX\beta \\ &= c + \left(s^T - \beta_0^T X^T D \right) \beta_X + \frac{1}{2}\beta_X^T D\beta_X \end{aligned}$$

where $\beta_X = X\beta$. We can get this into the desired quadratic form by completing the square. Given the formula from Domke's blog, we can add and subtract constants to get

$$\begin{aligned} q(\beta; \beta_0) &= c' + \frac{1}{2}[\beta_X + D^{-1}(s - DX\beta_0)]^T \; D \; [\beta_X + D^{-1}(s - DX\beta_0)] \\ &= c' + \frac{1}{2}[\beta_X + (D^{-1}s^T - X\beta_0)]^T \; D \; [\beta_X + (D^{-1}s - X\beta_0)] \\ &= c' + \frac{1}{2}[(-D^{-1}s + X\beta_0) - X\beta]^T \; D \; [(-D^{-1}s + X\beta_0) - X\beta]^T \end{aligned}$$

This gives us the final expression we're looking for, with $W_{i,i} = D_{i,i} = m_i w_i (1 - w_i)$ and $\mathrm{Z} = D^{-1}s + X\beta_0 =$