# SDS 385 Exercise Set 01

Kevin Song

September 2, 2016

## 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 $\beta$, and $\beta$ 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 $\beta$ 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($Ux = b$)
$\hat{\beta}$ = 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:

```python
## 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. Stacking this solver up against the previous solver, we get the following values:

| 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 $\beta$, 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

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

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 $\beta_i$ 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.

```
import math
import numpy as np
import csv
import sys
from numpy import linalg as LA
import pdb

bump = 0.00000001 # A tiny bump for some values that really should not be zero
smallest_safe_exponent = math.log(sys.float_info.min) + 3
largest_safe_exponent = math.log(sys.float_info.max)   3

def safe_exp(val):
    """Calculates the "safe exponential" of a value. If the computed exponential would
    be too large, it replaces it with a safe value."""

    if val > largest_safe_exponent:
        return math.exp(largest_safe_exponent)
    elif val < smallest_safe_exponent:
        return math.exp(smallest_safe_exponent)
    else:
        return math.exp(val)

def calc_likelihood_function(X,y,m):
    """Gives a function to determine the likelihood in the inverse logit method.
    Returns a function which takes in beta and returns the likelihood as a float."""

    def likelihood(B):
        result = 0

        xvecs = [ xs for xs in X ]                          # Length Samples
        exponents = [ np.dot(x,B) for x in xvecs ]
        expterms = [ safe_exp(z) for z in exponents ]
        weights = [ 1.0 / (1.0 + e) for e in expterms ]

        for i in range(len(xvecs)):
            result  = np.asscalar(y[i]) * math.log(weights[i])
            result  = np.asscalar(m[i]   y[i]) * math.log(1   weights[i])

        return result
    return likelihood

def calc_grad_function(X,y,m):
    """Calculates the gradient of the inverse logit MLE given the parameters.
    Return is a lambda function which takes a single parameter and returns float."""

    # X is a feature matrix: columns are features, rows are entries. Dim: samples x features
    # y is a response vector: one column of responses                Dim: samples x 1
    # m is a trials vector                                           Dim: samples x 1

    def grad(B):
        # B should be a col vector with length = # features
        xvecs = [ xs for xs in X ]                          # Length Samples
        exponents = [ np.dot(x,B) for x in xvecs ]
        expterms = [ safe_exp(z) for z in exponents ]
        weights = [ 1.0 / (1.0 + e) for e in expterms ]

        W = np.matrix(np.diag(np.array(weights)))

        gradient = X.T * (y   W * m)
        return gradient

    return grad

def steepest_descent(params, initial_guess , converge_step):

    (X,y,m) = params

    # A function which calculates the gradient at a point
    grad_op = calc_grad_function(X,y,m)

    # A function which calculates the likelihood at a point
    llh_op = calc_likelihood_function(X,y,m)

    delta = sys.float_info.max
    guess = initial_guess

    # For storing likelihoods (for tracking convergence)
    likelihood_record = []

    ## Main Steepest Descent Loop
    while delta > converge_step:
        oldGuess = guess

        grad = grad_op(guess)
        step = 0.001

        guess = guess   grad * step

        delta = abs(llh_op(oldGuess)   llh_op(guess))

        likelihood_record.append(delta)

        print(delta)

    return (guess,likelihood_record)

def main(filename):
    """Driver for steepest descent code."""

    rawdata = []

    # Read the data in to rawdata
    with open(filename,'r') as ifile:
        r = csv.reader(ifile)
        for row in r:
            rawdata.append(row)

    # Get the data columns from the raw data and convert to matrix
    datalist = [ row[2:11] for row in rawdata ]
    predictors = np.array(datalist)
    predictors = predictors.astype(float)

    # Attach column of ones for intercept term
    onecol = np.matrix(np.ones(np.shape(predictors)[0])).T

    predictors = np.matrix(np.concatenate((predictors , onecol),axis=1))

    # Get the responses and convert to 0 1 matrix
    responselist = [ 1 if row[1] == "M" else 0 for row in rawdata ]
    response = np.matrix(responselist).T

    # Get the "trial number" for the MLE, in this case, a vectors of 1s
    trials = np.matrix(np.ones(np.shape(response)))

    # Recondition matrix to have order of magnitude ~1
    condWeights = np.average(predictors,axis=0)
    W = np.matrix(np.diagflat(condWeights))

    predictors = predictors * W.I

    initGuess = np.matrix(np.ones((np.shape(predictors)[1],1))) * 0.001

    solution,_ = steepest_descent((predictors, response, trials), initGuess, 0.01)

    # Since we changed the predictors, need to reverse transform the solution
    solution = W * solution

    print(solution)
```

steepestdescent.py

## 7 Quadratic Forms Again

The second-order Taylor approximation of $l(\beta)$ about $\beta_0$ is given by Taylor's Theorem:

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

We can get this into the desired quadratic form by completing the square. To derive the form for this, we write the form we have and the form we would like, and attempt to get the two to agree. Assuming that $M$ and $C$ are symmetric,

$$\begin{aligned} a + b^Tx + \frac{1}{2}x^TCx &= \frac{1}{2}(x-y)^TM(x-y) \\ &= \frac{1}{2}\left(x^TMx - x^TMy - y^TMx + y^TMy\right) + v \\ \frac{1}{2}x^TCx + b^Tx + a &= \left(\frac{1}{2}x^TMx\right) - y^TMx + \frac{1}{2}\left(y^TMy + v\right) \end{aligned}$$

Comparing the terms on the left and right, this suggests that

$$\begin{aligned} M &= C \\ -y^TM = b^T &\implies y = -M^{-1}b = -C^{-1}b \\ v = a - \frac{1}{2}y^TMy &= a - \frac{1}{2}b^TC^{-1}b \end{aligned}$$

Applying this to our problem, we get that

$$\begin{aligned} M &= H_l(\beta_0) \\ y &= H_l(\beta_0)^{-1}\nabla l(\beta_0) \\ v &= l(\beta_0) - \frac{1}{2}\nabla l(\beta_0)^T H_l(\beta_o)^{-1}\nabla l(\beta_0) \end{aligned}$$