Linear Regression

(A) $F(\beta) = \frac{1}{2}\sum_{i=1}^{N} \omega_i (y_i - x_i^T \beta)^2$

F is differentiable.

Set $F'(\beta) = \sum_{i=1}^{N} \omega_i (y_i - x_i^T \beta) = 0$

$\Leftrightarrow Wy = WX\hat{\beta}$

$\Leftrightarrow X^T Wy = X^T WX\hat{\beta}$

(B) Numerically speaking, the inversion method is not the fastest and most stable way to solve the linear system. Since $X^T WX$ is symmetric, it is much faster to solve this problem using triangular system. (Triangular systems are one of the simplest systems to solve.) *Cholesky Factorization*, *QR Factorization*, and *SDV* are all capable here.

- *Cholesky Factorization*: is much quicker than other algorithms but is in general more unstable. It is relatively more sensitive compared to the original Least Squares Problem $Ax = b$.
- *QR Factorization*: enhances both complexity and stability. It is more accurate and broadly applicable, but may fail when A is nearly rank-deficient
- *SDV*: it always exists and can be computed stably. The computed SVD will be well-conditioned because orthogonal matrices preserve the 2-norm. Any perturbation in A will not be amplified by the SVD since $\|\delta A\|_2 = \|\delta \Sigma\|_2$.

Algorithms:

(1) Cholesky Factorization:

(i) Calculate $LL^T = X^T WX$

(ii) Calculate $d = X^T sqrt(W)y$

(iii) Solve $Lz = d$ by forward substitution

(iv) Solve $L^T\hat{\beta} = z$ by back substitution

(2) QR Factorization:

(i) $X^T sqrt(W) = QR$, where $Q_{P\times N}$ has orthonormal columns, and $R_{N\times N}$ is upper triangular.

(ii) Form $d = Q^T y$

(iii) Solve $R\hat{\beta} = d$ by back substitution

Code:

```
Chol = function(y, X, weights) {
    sqrtW = sqrt(weights)
    C = crossprod(sqrtW*X)
    d = crossprod(sqrtW*X,sqrtW*y)
    L=chol(C)
    z=forwardsolve(L,d)
    betahat=backsolve(t(L),z)
    return(betahat)
}

QR = function(y, X, W) {
    sqrtW = sqrt(W)
    XtsqrtW=sqrtW %*%X
    sqrtWy=sqrtW%*%y
    qr = qr(XtsqrtW)
    qty = qr.qty(qr, sqrtWy)
    betahat = backsolve(qr$qr, qty)
    return(betahat)
}
```

(C) **Code**:

```
N = 2000
P = 500
X = matrix(rnorm(N*P), nrow=N)
```

```
beta = rnorm(P)
weights0 = runif(N, min = 0, max = 1)
weights=weights0/sum(weights0)
W=diag(weights)
eps = rnorm(N)
y = X %*% beta + eps

Chol = function(y, X, weights) {
    sqrtW = sqrt(weights)
    C = crossprod(sqrtW*X)
    d = crossprod(sqrtW*X,sqrtW*y)
    L=chol(C)
    z=forwardsolve(L,d)
    betahat=backsolve(t(L),z)
    return(betahat)
}

QR = function(y, X, W) {
    sqrtW = sqrt(W)
    XtsqrtW=sqrtW %*%X
    sqrtWy=sqrtW%*%y
    qr = qr(XtsqrtW)
    qty = qr.qty(qr, sqrtWy)
    betahat = backsolve(qr$qr, qty)
    return(betahat)
}

inv = function(y, X, W){
    betahat=solve(t(X)%*%W%*%X, t(X)%*%W%*%y  )
    return(betahat)
```

```
}


#beta_chol=Chol(y,X,weights)
#beta_QR=QR(y,X,W)
#beta_inv=inv(y,X,W)

microbenchmark(
    Chol(y,X,weights),
    QR(y,X, W),
    inv(y,X,W),
    times=5)
```

Comparison Result:

```
Unit: milliseconds
                expr      min       lq     mean   median       uq      max neval
 Chol(y, X, weights)  219.1969 220.1072 221.0336 220.7857 222.3032 222.7753     5
         QR(y, X, W) 1442.9885 1539.4898 1558.3377 1553.3521 1587.2561 1668.6018     5
        inv(y, X, W) 2085.7829 2096.2960 2138.7876 2152.5467 2170.3436 2188.9686     5
```

Figure 1: Compare Cholesky, QR, and Inverse Methods

(D)

Generalized linear models

(A) $l(\beta) = -log\{\prod_{i=1}^{N} p(y_i|\beta)\}$

$= -\sum_{i=1}^{N} log[p(y_i|\beta)]$

$= -\sum_{i=1}^{N} log[\binom{m_i}{y_i} w_i^{y_i}(1-w_i)^{m_i-y_i}]$

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

$\nabla_\beta l(\beta) = [-\sum_{i=1}^{N}[y_i \frac{1}{w_i} \cdot \frac{dw_i}{d\beta_1} - (m_i - y_i)\frac{1}{1-w_i} \cdot \frac{dw_i}{d\beta_1}], -\sum_{i=1}^{N}[y_i \frac{1}{w_i} \cdot \frac{dw_i}{d\beta_2} - (m_i - y_i)\frac{1}{1-w_i} \cdot$

$\frac{dw_i}{d\beta_2}], \cdots, -\sum_{i=1}^{N}[y_i \frac{1}{w_i} \cdot \frac{dw_i}{d\beta_p} - (m_i - y_i)\frac{1}{1-w_i} \cdot \frac{dw_i}{d\beta_p}]]$

$\frac{dw_i}{d\beta_j} = w_i^2 \cdot e^{x_i^T \beta} x_{ij}$

$\nabla_\beta l(\beta) = [-\sum_{i=1}^{N}[y_i \frac{1}{w_i} \cdot w_i^2 \cdot e^{x_i^T\beta} x_{i1} - (m_i - y_i)\frac{1}{1-w_i} \cdot w_i^2 \cdot e^{x_i^T\beta} x_{i1}], \cdots, -\sum_{i=1}^{N}[y_i \frac{1}{w_i} \cdot w_i^2 \cdot e^{x_i^T\beta} x_{ip} - (m_i - y_i)\frac{1}{1-w_i} \cdot w_i^2 \cdot e^{x_i^T\beta} x_{ip}]]$

$= [-\sum_{i=1}^{N}[y_i(1-w_i)x_{i1} - (m_i - y_i)w_i x_{i1}], \cdots, -\sum_{i=1}^{N}[y_i(1-w_i)x_{ip} - (m_i - y_i)w_i x_{ip}]]$

$= [-\sum_{i=1}^{N}(y_i x_{i1} - m_i w_i x_{i1}), \cdots, -\sum_{i=1}^{N}(y_i x_{ip} - m_i w_i x_{ip})]$

$= -(y - mw)^T X$

(B)**Steepest descent**

Algorithm:

(i) Given β^0, set $k := 0$.

(ii) $d^k := \nabla f(\beta^k)$. If $\|d^k\| \leq \epsilon$, then stop.

(iii) Solve $min_\lambda h(\lambda) := f(\beta^k + \lambda d^k)$ for the step-length λ^k

(vi) Set $\beta^{k+1} \leftarrow \beta^k + \lambda^k d^k$, $k \leftarrow k + 1$. Go to Step 1.

Use **Bisection Line-Search Algorithm** to find proper $\boldsymbol{\lambda}$

(i). Set $k = 0$. Set $\lambda_L := 0$ and $\lambda_U := \hat{\lambda}$.

(k). Set $\tilde{\lambda} = \frac{\lambda_U + \lambda_L}{2}$ and compute $h'(\tilde{\lambda})$.

If $h'(\tilde{\lambda}) > 0$, re-set $\lambda_U := \tilde{\lambda}$. Set $k \leftarrow k + 1$.

If $h'(\tilde{\lambda}) < 0$, re-set $\lambda_L := \tilde{\lambda}$. Set $k \leftarrow k + 1$.

If $h'(\tilde{\lambda}) = 0$, stop.

Code:

```
wdbc = read.csv('F:/Courses/2016 Fall/Stat Model for Big Data/Exercise 01/wdbc.csv',
 header=FALSE)

y0 = wdbc[,2]
y=as.numeric(y0=="M")
```

```
X = wdbc[,3:12]
#X = as.matrix(wdbc[,-c(1,2)])
#scrub = which(1:ncol(X) %% 3 == 0)
#scrub = 11:30
#X = X[,-scrub]

glm1 = glm(y~X, family='binomial')

#### Preset ####################

N=569
m=rep(1,N)

iter=100
beta=rep(0,iter)
beta[1]=glm1
epsilon=0.01
lambda=rep(0,iter)
h=rep(0,iter)
w=matrix(1,nrow=iter,ncol=N)


#### Gradient ###################

grad=function(X, y, beta){
    w=1 / ( 1+exp( - X%*%beta ) )
    gradient= - t(X) %*% ( y - m*w  )
    return(gradient)
}
```

```
#### Loglikelihood ###############
loglikelihood=function(X, y, beta){
    w=1 / ( 1+exp( - X%*%beta ) )
    loglikelihood= - sum(  log(choose(m, y) ) + y*log(w) + (m-y) log(1-w)   )
    return(loglikelihood)
}

#### Steepest descent #############

steepest=function(X, y, beta){
    k=1
    h[k]=loglikelihood(X, y, beta)
    while ( sum(d^2 > epsilon || k<iter-1 ) ) {
        d=grad(X, y, beta)
        lambda[k]=bisec(X, y, beta)
        h[k+1]=loglikelihood(X, y, beta+ lambda[k]*d)
        beta[k+1]=beta[k]+ lambda[k]*d
        w[k+1,]=1 / ( 1+exp( - X%*%beta[k+1] ) )
        k = k+1
        }

}

#### Bisection ###################

bisec=function(X, y, beta){
    i=0
    lambda_L=0
    lambda_U=2
    lambda=(lambda_L+lambda_U)/2
```

```
    grad0=grad(X, y, beta)
    h_prime=t(grad(X, y, beta+lambda*grad0)) %*% grad0
    while ( h_prime!= 0) {
        if ( h_prime >0 ) {
            lambda_U=lambda
        }
        else
            lambda_L=lambda

        i=i+1
        lambda=(lambda_L+lambda_U)/2
        h_prime=t(grad(X, y, beta+lambda*grad0)) %*% grad0
    }
    return(lambda)
}
```

(C)

$l(\beta) = l(\beta_0) + (\nabla l(\beta_0))^T(\beta - \beta_0) + \frac{1}{2}(\beta - \beta_0)^T \nabla^2 l(\beta_0)(\beta - \beta_0)$

$\nabla^2 l(\beta_0)$

$$= \begin{bmatrix} -\sum_{i=1}^N m_i w_i(1-w_i)x_{i1}x_{i1} & -\sum_{i=1}^N m_i w_i(1-w_i)x_{i1}x_{i2} & \cdots & -\sum_{i=1}^N m_i w_i(1-w_i)x_{i1}x_{ip} \\ -\sum_{i=1}^N m_i w_i(1-w_i)x_{i2}x_{i1} & -\sum_{i=1}^N m_i w_i(1-w_i)x_{i2}x_{i2} & \cdots & -\sum_{i=1}^N m_i w_i(1-w_i)x_{i2}x_{ip} \\ \vdots & \vdots & \ddots & \vdots \\ -\sum_{i=1}^N m_i w_i(1-w_i)x_{ip}x_{i1} & -\sum_{i=1}^N m_i w_i(1-w_i)x_{ip}x_{i2} & \cdots & -\sum_{i=1}^N m_i w_i(1-w_i)x_{ip}x_{ip} \end{bmatrix}$$

$$= \begin{bmatrix} x_{11} & \cdots & x_{N1} \\ \vdots & \ddots & \vdots \\ x_{1p} & \cdots & x_{Np} \end{bmatrix} \cdot \begin{bmatrix} m_1 w_1(1-w_1) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & m_N w_N(1-w_N) \end{bmatrix} \cdot \begin{bmatrix} x_{11} & \cdots & x_{1p} \\ \vdots & \ddots & \vdots \\ x_{N1} & \cdots & x_{Np} \end{bmatrix}$$

$= X_T W X$

Then,

$l(\beta) = \frac{1}{2}[(\beta - \beta_0) - m]^T \tilde{W}[(\beta - \beta_0) - m] + \tilde{v}$

$= \frac{1}{2}[\beta - m]^T \tilde{W}[\beta - m] + v$

Where,

$\tilde{W} = \nabla^2 l(\beta_0) = X^T W X$

$m = -(\nabla^2 l(\beta_0))^{-1} \nabla l(\beta_0)$

$v = l(\beta_0) - \frac{1}{2}\nabla l(\beta_0)(\nabla^2 l(\beta_0))^{-1}\nabla l(\beta_0) - \frac{1}{2}\beta_0^T \tilde{W}\beta_0$, **not relying on** β.

Therefore,

$l(\beta) = \frac{1}{2}(X\beta - z)^T W(X\beta - z) + v = \frac{1}{2}(z - X\beta)^T W(z - X\beta) + v$, where $z = Xm$

(D) **Newton Method**

The algorithm

(1) Given β^0, set $k := 0$.

(2) $d^k := -(\nabla^2 f_k)^{-1}\nabla f(\beta^k)$. If $\|d^k\| \leq \epsilon$, then stop.

(3) Solve $min_\lambda h(\lambda) := f(\beta^k + \lambda d^k)$ for the step-length λ^k

(4) Set $\beta^{k+1} \leftarrow \beta^k + \lambda^k d^k$, $k \leftarrow k + 1$. Go to Step 1.

Use **Bisection Line-Search Algorithm** to find proper $\boldsymbol{\lambda}$

(1). Set $k = 0$. Set $\lambda_L := 0$ and $\lambda_U := \hat{\lambda}$.

(k). Set $\tilde{\lambda} = \frac{\lambda_U + \lambda_L}{2}$ and compute $h'(\tilde{\lambda})$.

If $h'(\tilde{\lambda}) > 0$, re-set $\lambda_U := \tilde{\lambda}$. Set $k \leftarrow k + 1$.

If $h'(\tilde{\lambda}) < 0$, re-set $\lambda_L := \tilde{\lambda}$. Set $k \leftarrow k + 1$.

If $h'(\tilde{\lambda}) = 0$, stop.

Code:

```
#### Newton Method using Bisection #############


newton=function(X, y, beta){
    k=1
    h[k]=loglikelihood(X, y, beta)
    while ( sum(d[k]^2 > epsilon || k<iter-1 ) ) {
```

```
        d=grad(X, y, beta)
        w=1 / ( 1+exp( - X%*%beta ) )
        W=diag(m*w*(1-w))
        hess=t(X)%*%W%*%X
        p=solve(hess) %*% d
        lambda[k]=bisec(X, y, beta[k])
        h[k+1]=loglikelihood(X, y, beta[k]+ lambda[k]*d)
        beta[k+1]=beta[k]+ lambda[k]*d
        w[k+1,]=1 / ( 1+exp( - X%*%beta[k+1] ) )
        k = k+1
    }
}
```

(E) - Gradient descent generally requires more iterations, but each iteration is fast (we only need to compute 1st derivatives).
- Newton's method generally requires fewer iterations, but each iteration is slow (we need to compute 2nd derivatives too).