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

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

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

(B) Numerically speaking, the inversion method is not the fastest and most stable way to solve the linear system. Since $X^T W X$ 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 $LL^T = X^T W X$

(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) Sparse Matrix

Four different methods can be used to solve the linear problem: as before - *Cholesky Decomposition*, *QR decomposition*, *Inversion*; and a new method - *LU Decomposition*.

The algorithm of *LU Decomposition* is as follows:

i) Solve $Lz = PX^TWy$ for z, where L is a lower triangle matrix, and P is a diagonal one;

ii) Solve $U\hat{\beta} = z$, where U is a upper triangle matrix.

```
LU = function(y, X, W) {
```

```
    C=t(X)%*%W%*%X
    lu=lu(C)
    d=t(X)%*%W%*%y
    L=expand(lu)$L
    U=expand(lu)$U
    P=expand(lu)$P
    z=forwardsolve(L,d)
    betahat=backsolve(U,z)
    return(betahat)
}
```

Comparison Result:

```
+ times=5)
Unit: milliseconds
                 expr        min         lq       mean     median         uq        max neval
 Chol(y, Xs, weights)   75.20986   75.42453   76.08251   75.65003   77.03669   77.09143     5
        inv(y, Xs, W) 1820.34379 1901.27790 1886.43583 1902.70733 1902.91401 1904.93613     5
         LU(y, Xs, W) 1722.18160 1902.61211 1913.82428 1908.23031 1971.37958 2064.71781     5
>
```

Figure 2: Compare Cholesky, LU, and Inverse Methods

However, when running the *QR Decomposition*, it gives the following error.

```
> QR(y,Xs,W)
Error in qr.qty(qr, sqrtWy) :
  no method for coercing this S4 class to a vector
>
```

Figure 3: QR error

I'm still trying to dig out what causes this error. But for now, it seems like *QR Decomposition* is not very stable when dealing with sparse matrix.

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 $\|d^k\| \le \epsilon$, 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('C:/Users/Yuxin/Dropbox/Courses/2016 Fall/Stat Model for Big Data/Exerci

y0 = wdbc[,2]
y=as.numeric(y0=="M")
n=length(y0)
X0 = as.matrix(wdbc[,3:12])
X=cbind(rep(1,n),X0)
p=length(X[1,])
b0=rep(1,p)
tol=1e-2

ilogit=function(u) return( 1/(1+exp(-u)));
gradient.descent=function(X, y, maxit=100)
{
p=ncol(X)
X.t.y=crossprod(X, y)
b=rep(1,p)
k=0
add=tol+1
while( (k <= maxit) & (sum(abs(add)) > tol))
{
  k=k+1
  step.size=1/k^1
  pi.t=ilogit(as.numeric(X%*%b))
  W=diag(pi.t*(1-pi.t))
  ## compute the direction
  minusGrad=c(X.t.y-crossprod(X, pi.t))
```

```
  add=step.size*minusGrad
  b=b+add

  if( (sum(add^2) < tol) | (k >=maxit))
    iterating=FALSE

  b0=rbind(b0,b)
}
b=as.numeric(b)
return(list(b0=b0,b=b, total.iterations=k))
}

fit=gradient.descent(X=X, y=y,  maxit=1000)
```

Total iteration time: 1000

```
$total.iterations
[1] 1001
```

Figure 4: Total Iteration

Path of β

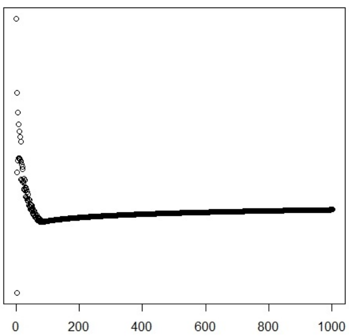

Figure 5: Sample path of beta

(C)

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

$$
\begin{aligned}
&\nabla^2 l(\beta_0) \\
&= \begin{bmatrix} -\sum_{i=1}^N m_i w_i(1-w_i)x_{i1}x_{i1} & -\sum_{i=1}^N m_i w_i(1-w_i)x_{i1}x_{i2} & \cdots & -\sum_{i=1}^N m_i w_i(1-w_i)x_{i1}x_{ip} \\ -\sum_{i=1}^N m_i w_i(1-w_i)x_{i2}x_{i1} & -\sum_{i=1}^N m_i w_i(1-w_i)x_{i2}x_{i2} & \cdots & -\sum_{i=1}^N m_i w_i(1-w_i)x_{i2}x_{ip} \\ \vdots & \vdots & \ddots & \vdots \\ -\sum_{i=1}^N m_i w_i(1-w_i)x_{ip}x_{i1} & -\sum_{i=1}^N m_i w_i(1-w_i)x_{ip}x_{i2} & \cdots & -\sum_{i=1}^N m_i w_i(1-w_i)x_{ip}x_{ip} \end{bmatrix} \\
&= \begin{bmatrix} x_{11} & \cdots & x_{N1} \\ \vdots & \ddots & \vdots \\ x_{1p} & \cdots & x_{Np} \end{bmatrix} \cdot \begin{bmatrix} m_1 w_1(1-w_1) & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & m_N w_N(1-w_N) \end{bmatrix} \cdot \begin{bmatrix} x_{11} & \cdots & x_{1p} \\ \vdots & \ddots & \vdots \\ x_{N1} & \cdots & x_{Np} \end{bmatrix} \\
&= X_T W X
\end{aligned}
$$

Then,

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

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 ############


Newton=function(X, y, maxit)
{
p=ncol(X)
X.t.y=crossprod(X, y)
b=rep(1,p)


k=0
add=tol+1
while( (k <= maxit) & (sum(abs(add)) > tol))
{
  k=k+1
  step.size=1
  pi.t=ilogit(as.numeric(X%*%b))
  W=diag(pi.t*(1-pi.t))
  ## compute the direction
  minusGrad=c(X.t.y-crossprod(X, pi.t))
  Hess=crossprod(X,W%*%X)
  dir=solve(Hess, minusGrad)
  add=step.size*dir
  b=b+add


  if( (sum(add^2) < tol) | (k >=maxit))
```

```
    iterating=FALSE

  b0=rbind(b0,b)
}
b=as.numeric(b)
return(list(b0=b0,b=b, total.iterations=k))
}

fit=Newton(X=X, y=y,  maxit=1000)
```

(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).