```
---
title: "AMS 274 - Assignment 1"
output: pdf_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```

\begin{center}
Sam Leonard
\end{center}

In this paper I'll explore a four different algorithms for obtaining estimates
for the parameters of a probit regression: Newton-Raphson, Fisher scoring,
coordinate descent, and stochastic gradient.

Recall that probit regression estimates a categorical outcome:

$$y_{i}|\theta_{i}\ \sim\ \textrm{Ber}(\theta_{i})$$

$$\theta_{i}\ =\ \Phi(x_{i}^{T}\beta)$$

Section one is the tough part: the math.

In the second section I will show the code I wrote for executing each of these
algorithms.

And finally, section three will be a brief comparison of the results against a
simulated dataset.

Section 1 - the Math

\textbf{Newton method}:

$$\log(p(\underline{y}|\beta))=\sum_{i=1}^{n}\big(y_{i}\log(\Phi(x_{i}^{T}\beta))+
(1-y_{i})\log(1-\Phi(x_{i}^{T}\beta)\big)$$

For some r $\in$ $\{1, ..., p\}$:

$$\frac{\partial l}{\partial \beta_{r}} =
\sum_{i=1}^{n}\bigg(\frac{y_{i}\phi(x_{i}^{T}\beta)}{\Phi(x_{i}^{T}\beta)}-
\frac{(1-y_{i})\phi(x_{i}^{T}\beta)}{1-\Phi(x_{i}^{T}\beta)}\bigg)x_{i, r}$$

This expression appears for each β_{r} in the matrix:

$$\nabla l(\beta) =
\begin{bmatrix}
\frac{\partial l}{\partial \beta_{1}} \\
\frac{\partial l}{\partial \beta_{2}} \\
. \\
. \\
. \\
\frac{\partial l}{\partial \beta_{p}} \\
```

\end{bmatrix} $$

For some s where s $\in$ $\{1, ..., p\}$:

$$\frac{\partial l^{2}}{\partial\beta_{r}\partial\beta_{s}} =
\sum_{i=1}^{n}\bigg(\frac{y_{i}\phi(x_{i}^{T}\beta)(-2*\frac{1}
{2}*x_{i}^{T}\beta)}{\Phi(x_{i}^{T}\beta)} -
\frac{y_{i}\phi(x_{i}^{T}\beta)\phi(x_{i}^{T}\beta)}{\Phi(x_{i}^{T}\beta)^{2}} -
\frac{(1-y_{i})\phi(x^{T}_{i}\beta)(-x_{i}^{T}\beta)}{1-\Phi(x_{i}^{T}\beta)} -
\frac{(1-y_{i})\phi(x_{i}^{T}\beta)(-\phi(x_{i}^{T}\beta))}{(1-
\Phi(x_{i}^{T}\beta))^{2}}\bigg)x_{i, r}x_{i,s}$$

So

$$\frac{\partial l^{2}}{\partial\beta_{r}\partial\beta_{s}}=\sum_{i=1}^{n}\bigg(-
\frac{y_{i}\phi(x_{i}^{T}\beta)x_{i}^{T}\beta}{\Phi(x_{i}^{T}\beta} -
\frac{y_{i}\phi(x_{i}^{T}\beta)^{2}}{\Phi(x_{i}^{T}\beta)^{2}}+\frac{(1-
y_{i})\phi(x_{i}^{T}\beta)x_{i}^{T}\beta}{1-\Phi(x_{i}^{T}\beta)}-\frac{(1-
y_{i})\phi(x_{i}^{T}\beta)^{2}}{(1-\Phi(x_{i}^{T}\beta))^{2}}\bigg)x_{i, r}x_{i,
s}$$

This expression for each r and s completes the Hessian matrix:

$$Hes(\beta) =
\begin{bmatrix}
\frac{\partial l^{2}}{\partial\beta_{1}\partial\beta_{1}} & \frac{\partial l^{2}}
{\partial\beta_{1}\partial\beta_{2}} & ... &
\frac{\partial l^{2}}{\partial\beta_{1}\partial\beta_{p}}\\
\frac{\partial l^{2}}{\partial\beta_{2}\partial\beta_{1}} & \frac{\partial l^{2}}
{\partial\beta_{2}\partial\beta_{2}} & ... &
\frac{\partial l^{2}}{\partial\beta_{2}\partial\beta_{p}} \\
... & ... & ... & ... \\
\frac{\partial l^{p}}{\partial\beta_{p}\partial\beta_{1}} & \frac{\partial l^{2}}
{\partial\beta_{p}\partial\beta_{2}} & ... &
\frac{\partial l^{2}}{\partial\beta_{p}\partial\beta_{p}} \\
\end{bmatrix} $$

To implement Newton-Raphson, we start with some guess $\beta^{(0)}$. Then

$$\beta^{(1)}=\beta^{0} - [Hes(\beta^{(0)})]^{-1}*(\nabla l(\beta^{(0)})$$

And generally

$$\beta^{(t+1)}=\beta^{t} - [Hes(\beta^{(t)})]^{-1}*(\nabla l(\beta^{(t)})$$

This sequence converges to the minimum of $\nabla l(\beta)$. The log of the probit likelihood is unimodal, therefore this minimum will be the MLE for β.

A reasonable stopping condition is when the relative change in log likelihood is very small: $\big|\frac{l(\beta^{(t+1)}-l(\beta^{(t)})}{l(\beta^{(t+1)})}\big| <\epsilon$, where ϵ is chosen in advance.

\textbf{Fisher Scoring}

Each y_{i} is Bernoulli with $p = \Phi(x_{i}^{T}\beta)$. Thus $E[y_{i}] = \Phi(x_{i}^{T}\beta)$. Entries of the expected information matrix take the form

$$E[\frac{\partial l^{2}}{\partial\beta_{r}\partial\beta_{s}}] = \sum_{i=1}^{n}\bigg(-\frac{\Phi(x_{i}^{T}\beta)\phi(x_{i}^{T}\beta)x_{i}^{T}\beta}{\Phi(x_{i}^{T}\beta)} - \frac{\Phi(x_{i}^{T}\beta)\phi(x_{i}^{T}\beta)^{2}}{\Phi(x_{i}^{T}\beta)^{2}} + \frac{(1-\Phi(x_{i}^{T}\beta))\phi(x_{i}^{T}\beta)x_{i}^{T}\beta}{1-\Phi(x_{i}^{T}\beta)} - \frac{(1-\Phi(x_{i}^{T}\beta))\phi(x_{i}^{T}\beta)^{2}}{(1-\Phi(x_{i}^{T}\beta))^{2}}\bigg)x_{i, r}x_{i, s}$$

So

$$E[\frac{\partial l^{2}}{\partial\beta_{r}\partial\beta_{s}}] = \sum_{i=1}^{n}\bigg(\frac{-\Phi(x_{i}^{T}\beta)(1-\Phi(x_{i}^{T}\beta))-\phi(x_{i}^{T}\beta)^{2}\Phi(x_{i}^{T}\beta)}{\Phi(x_{i}^{T}\beta)(1-\Phi(x_{i}^{T}\beta))}\bigg)x_{i, r}x_{i, s} = -\sum_{i=1}^{n}\bigg(\frac{\phi(x_{i}^{T}\beta)^{2}}{\Phi(x_{i}^{T}\beta)(1-\Phi(x_{i}^{T}\beta))}\bigg)x_{i, r}x_{i, s}$$

The negative of this expression completes the expected information matrix:

$$I_{E}(\beta) =
-\begin{bmatrix}
E[\frac{\partial l^{2}}{\partial\beta_{1}\partial\beta_{1}}] & E[\frac{\partial l^{2}}{\partial\beta_{1}\partial\beta_{2}}] & ... & E[\frac{\partial l^{2}}{\partial\beta_{1}\partial\beta_{p}}]\\
E[\frac{\partial l^{2}}{\partial\beta_{2}\partial\beta_{1}}] & E[\frac{\partial l^{2}}{\partial\beta_{2}\partial\beta_{2}}] & ... & E[\frac{\partial l^{2}}{\partial\beta_{2}\partial\beta_{p}}] \\
... & ... & ... & ... \\
E[\frac{\partial l^{p}}{\partial\beta_{p}\partial\beta_{1}}] & E[\frac{\partial l^{2}}{\partial\beta_{p}\partial\beta_{2}}] & ... & E[\frac{\partial l^{2}}{\partial\beta_{p}\partial\beta_{p}}] \\
\end{bmatrix} $$

To execute the Fisher Scoring algorithm start with some guess $\beta^{(0)}$. Then, because we defined expected information as the negative of the expectation of the Hessian, we have:

$$\beta^{(1)} = \beta^{(0)} + [I_{E}(\beta^{(0)})]^{-1}*\nabla l(\beta^{(0)})$$

And

$$\beta^{(t+1)} = \beta^{(t)} + [I_{E}(\beta^{(t)})]^{-1}*\nabla l(\beta^{(t)})$$

The algorithm stops again when the relative change in the log likelihood from $l(\beta^{t})$ to $l(\beta^{t+1})$ is smaller than some pre-selected ϵ.

\textbf{Coordinate Descent}

Start with some guess for the vector β, call it $\beta^{(0)}$. Assume that $l(\beta)$ is a function of only β_{r}. All other elements of the vector β are fixed. Let β_{-r} be β excluding β_{r}. To execute a coordinate descent algorithm we start by finding β_{r} that minimizes $\frac{dl}{d\beta_{r}}$ given β_{-r}.

The first derivative of $l(\beta)$ with respect to β_{r} is

$$\frac{d l}{d \beta_{r}} = \sum_{i=1}^{n}\bigg(\frac{y_{i}\phi(x_{i}^{T}\beta)}{\Phi(x_{i}^{T}\beta)}-\frac{(1-y_{i})\phi(x_{i}^{T}\beta)}{1-\Phi(x_{i}^{T}\beta)}\bigg)x_{i, r}$$

No closed form solution for $\frac{dl}{d\beta_{r}}=0$ is apparent, so we can use Newton-Raphson in one dimension to find the β_{r} that minimizes $\frac{dl}{d\beta_{r}}$. The second derivative is

$$\frac{d l^{2}}{d^{2}\beta_{r}}=\sum_{i=1}^{n}\bigg(-\frac{y_{i}\phi(x_{i}^{T}\beta)x_{i}^{T}\beta}{\Phi(x_{i}^{T}\beta} -\frac{y_{i}\phi(x_{i}^{T}\beta)^{2}}{\Phi(x_{i}^{T}\beta)^{2}}+\frac{(1-y_{i})\phi(x_{i}^{T}\beta)x_{i}^{T}\beta}{1-\Phi(x_{i}^{T}\beta)}-\frac{(1-y_{i})\phi(x_{i}^{T}\beta)^{2}}{(1-\Phi(x_{i}^{T}\beta))^{2}}\bigg)x_{i, r}x_{i, r}$$

The one-dimensional Netwon-Raphson algorithm for finding β_{r} that minimizes $\frac{dl}{d\beta_{r}}$ given β_{-r} is

$$\beta_{r}^{(1)} = \beta_{r}^{(0)}-\frac{\frac{dl}{d_{\beta_{r}}}(\beta^{(0)})}{\frac{dl^{2}}{d^{2}\beta_{r}}(\beta^{(0)})}$$

and generally

$$\beta_{r}^{(t+1)} = \beta_{r}^{(t)}-\frac{\frac{dl}{d_{\beta_{r}}}(\beta^{(t)})}{\frac{dl^{2}}{d^{2}\beta_{r}}(\beta^{(t)})}$$

Continue this process until the relative change in $\beta_{r}^{(t)}$ is less than some pre-selected ϵ.

Once we have minimized $\frac{dl}{d\beta_{r}}$, the $r^{\textrm{th}}$ element of the vector β is now this minimizing value. Move to another element of the vector β, β_{s}. Use the same process to minimize $\frac{dl}{\beta_{s}}$ given β_{-s}.

Continue this process until each of the p Newton-Raphson sequences have converged. A reasonable stopping condition is when each of the p sequences stop after one iteration sequentially.

\textbf{Stochastic Gradient}

Again start with a reasonable guess for β, $\beta^{(0)}$.

To implement a stochastic gradient algorithm select m data points at random from your dataset. Then update $\beta^{(0)}$ by finding

$$\beta^{(1)} = \beta^{(0)}+\lambda_{0}*[\nabla l_{m}(\beta^{(0)})]$$

and generally

$$\beta^{(t+1)} = \beta^{(t)}+\lambda_{t}*[\nabla l_{m}(\beta^{(t)})]$$

Here $\nabla l_{m}(\beta)$ the gradient of $l(\beta)$ evaluated over the data points m. In our case this is:

$$\nabla l_{m}(\beta)=\sum_{i \in m}\bigg(\frac{y_{i}\phi(x_{i}^{T}\beta)}{\Phi(x_{i}^{T}\beta)} - \frac{(1-y_{i})\phi(x_{i}^{T}\beta)}{1-\Phi(x_{i}^{T}\beta)}\bigg)x_{i}$$

And $\lambda_{t}=\frac{\alpha}{1+\eta t}$ is a learning rate which influences the scale of movement from $\beta^{(t)}$ to $\beta^{(t+1)}$.

The m data points selected at each iteration are discarded after the completion of that iteration, so that no data point is randomly selected twice. The algorithm stops when a certain number of iterations has passed, or we see a stabilization in the log likelihood.

\vspace{5mm}

Section 2 - code

Here is the code for executing each algorithm.

First I'll simulate a dataset with

$$n = 100,000$$

$$x_{i, j}\ \sim\ \textrm{N}(0, 2)$$

$$\theta_{i} = \Phi(.2 + 2.4 x_{i, 1} - .5 x_{i, 3} + 1.2 x_{i, 7})$$

```{r Generate Data, echo=TRUE}
n = 100000
p = 7
x = matrix(rnorm(700000, 0, 2), ncol = 7)
x = cbind(1, x)
totals = .2*x[, 1]+2.4*x[, 2] - .5*x[, 4] + 1.2*x[, 8]
theta = pnorm(totals)
y = sapply(theta, function(t)rbinom(1, 1, prob = t))
true.beta = c(.2, 2.4, 0, -.5, 0, 0, 0, 1.2)

likelihood.beta <- function(beta){
  sums = apply(x, 1, function(t)sum(t*beta))
  l.term1 = pnorm(sums, log = TRUE)
  l.term2 =pnorm(-sums, log = TRUE)
```

```
  return(sum(y*l.term1+(1-y)*l.term2))
}
```


\textbf{Newton-Raphson}


```{r Newton Test, echo=TRUE}
loop = TRUE
beta = rep(0, 8)
l = 0


start <- Sys.time()
while (loop){
  info_matrix <- matrix(0, nrow = p+1, ncol = p+1)
  sums = apply(x, 1, function(t)sum(t*beta))
  dd.term1 = dnorm(sums)*sums/pnorm(sums)
  dd.term2 = (dnorm(sums)/pnorm(sums))**2
  dd.term3 = dnorm(sums)*sums/pnorm(-sums)
  dd.term4 = (dnorm(sums)/pnorm(-sums))**2
  d.term1 = dnorm(sums)/pnorm(sums)
  d.term2 = dnorm(sums)/pnorm(-sums)

  for (i in 1:(p+1)){
    for (j in i:(p+1)){
      x.1 = x[ , i]
      x.2 = x[ , j]
      info_matrix[i, j] = info_matrix[j, i] =
        sum( (-y*dd.term1 - y*dd.term2 + (1 - y)*dd.term3 - (1 - y)*dd.term4
)*x.1*x.2)
    }
  }

  info.inv = solve(info_matrix)
  dl.beta.vec = rep(0, p+1)
  for (i in 1:(p+1)){
    x.iter.1 = x[, i]
    dl.beta.vec[i] = sum((y*d.term1 - (1-y)*d.term2)*x.iter.1)
  }

  beta.t1 = beta - info.inv %*% dl.beta.vec
  l = l + 1
  if (abs((likelihood.beta(beta.t1)-
likelihood.beta(beta))/likelihood.beta(beta.t1)) < 10**(-4)){
    end <- Sys.time()
    cat('beta0: ', beta.t1[1], '\n')
    cat('beta1: ', beta.t1[2], '\n')
    cat('beta2: ', beta.t1[3], '\n')
    cat('beta3: ', beta.t1[4], '\n')
    cat('beta4: ', beta.t1[5], '\n')
    cat('beta5: ', beta.t1[6], '\n')
    cat('beta6: ', beta.t1[7], '\n')
    cat('beta7: ', beta.t1[8], '\n')
    cat('iterations: ', l, '\n')
```

```
    print(end - start)
    loop = FALSE
  }
  beta = beta.t1
}

beta.newton = beta.t1
time.newton = end - start
```

\textbf{Coordinate Descent}

```{r Coord Descent, echo=TRUE}
beta.0 = rep(0, 8)
beta.1 = beta.0
error.vector = rep(1, p+1)
errors = c()
location = 1
l = 0
loop = TRUE

start <- Sys.time()
  while (loop){
    likelihood.0 = likelihood.beta(beta.0)
    sums = apply(x, 1, function(t)sum(t*beta.0))
    dd.term1 = dnorm(sums)*sums/pnorm(sums)
    dd.term2 = (dnorm(sums)/pnorm(sums))**2
    dd.term3 = dnorm(sums)*sums/pnorm(-sums)
    dd.term4 = (dnorm(sums)/pnorm(-sums))**2
    d.term1 = dnorm(sums)/pnorm(sums)
    d.term2 = dnorm(sums)/pnorm(-sums)

    beta.1[location] = beta.0[location] -
                       sum((y*d.term1 - (1-y)*d.term2)*x[ , location])/
                       sum( (-y*dd.term1 - y*dd.term2 + (1 - y)*dd.term3 - (1 -
y)*dd.term4 )*x[, location]*x[, location])

    likelihood.1 = likelihood.beta(beta.1)
    err = abs((likelihood.1 - likelihood.0)/likelihood.1)
    errors = append(errors, err)
    if(err < 10**-4){
      error.vector[location] = max(errors)
      errors = c()
      location = (location %% 8) + 1
    }
    if(all(error.vector < 10**-4)){
      l = l + 1
      end <- Sys.time()
      cat('beta0: ', beta.1[1], '\n')
      cat('beta1: ', beta.1[2], '\n')
      cat('beta2: ', beta.1[3], '\n')
      cat('beta3: ', beta.1[4], '\n')
      cat('beta4: ', beta.1[5], '\n')
      cat('beta5: ', beta.1[6], '\n')
      cat('beta6: ', beta.1[7], '\n')
      cat('beta7: ', beta.1[8], '\n')
```

```
      cat('iterations: ', l, '\n')
      print(end - start)
      loop = FALSE
    }
    beta.0 = beta.1
    l = l + 1
  }

beta.coord = beta.1
time.coord = end - start
```

\textbf{Stochastic Gradient}

I ran the stochastic gradient algorithms against standardized data.

In a simple case of one intercept and one covariate, against standardized data we have $\theta_{i} = \Phi(\beta_{0}+\beta_{1}(\frac{x_{i, 1}-\bar{x}_{1}}{s_{1}}))=\Phi((\beta_{0}-\frac{\bar{x}_{1}}{s_{1}})+\frac{\beta_{1}}{s_{1}}x_{i, 1})$.

Thus to compare scaled MLEs to the MLEs already generated, we simply need to divide by the standard deviation of the data column x, except in the case of the intercept. In the case of the intercept we simply subtract the mean of each column divided by its standard deviation.

This dataset was generated by sampling from a normal distribution with mean 0 and standard deviation 2. Therefore scaling the data will not change the intercept paramete, and the remaining parameters generated from standardized data will be twice the magnitude of those generated from unstandardized data.

For $m=1000$, $\alpha = .0045$ and $\eta = .05$ stabilized around good estimates of the β used to generate the data after just 100 iterations.

```{r Stochastic 1000 Test, echo=TRUE}
    m = .01*n
    x.scaled <- cbind(rep(1, n), scale(x[, -1]))
    location = 1
    beta.0 = rep(0, 8)
    beta.1 = beta.0
    alpha = .0083
    eta = .05
    niter = 0
    indices = seq(1:100000)
    likelihood.vec.1 <- rep(0, 100)

    start <- Sys.time()
    while(niter < 100){
      beta.0 = beta.1
      samp = sample(indices, m)
      indices = indices[!indices %in% samp]
      mat = x.scaled[samp, ]
      sums = apply(mat, 1, function(t)sum(t*beta.0))
      d.term1 = apply(mat, 1, function(t)exp(dnorm(sum(t*beta.0), log = TRUE) -
pnorm(sum(t*beta.0), log = TRUE)))
```

```
      d.term2 = apply(mat, 1, function(t)exp(dnorm(sum(t*beta.0), log = TRUE) -
(pnorm(-sum(t*beta.0), log = TRUE))))
      for (location in 1:(p+1)){
        beta.1[location] = beta.0[location] + alpha/(1+niter*eta) *
sum((y[samp]*d.term1 - (1-y[samp])*d.term2)*x.scaled[samp , location])
      }
      location = 1
      beta.scaled = .5*beta.1
      beta.scaled[1] = beta.1[1]
      likelihood.vec.1[niter+1] = likelihood.beta(beta.scaled)
      niter = niter + 1
    }

    test = TRUE
    if(test){
      end <- Sys.time()
      cat('beta0: ', beta.1[1], '\n')
      cat('beta1: ', beta.1[2], '\n')
      cat('beta2: ', beta.1[3], '\n')
      cat('beta3: ', beta.1[4], '\n')
      cat('beta4: ', beta.1[5], '\n')
      cat('beta5: ', beta.1[6], '\n')
      cat('beta6: ', beta.1[7], '\n')
      cat('beta7: ', beta.1[8], '\n')
      cat('iterations: ', niter, '\n')
      print(end - start)
      beta.m1000 = beta.1
      time.m1000 = end - start
    }
```

\vspace{5mm}

For $m=1$, I found $\eta=.0009$ and $\alpha = .086$ to yield reasonable
convergence.

```{r Stochastic 1 Test, echo=TRUE}
      m = 1
      location = 1
      beta.0 = rep(0, 8)
      beta.1 = beta.0
      alpha = .086
      eta = .0009
      niter = 0
      indices = seq(1:100000)
      beta.matrix = matrix(0, ncol = 8, nrow = 30000)

      start <- Sys.time()
      while(niter < 30000){
        beta.0 = beta.1
        samp = sample(indices, m)
        indices = indices[!indices %in% samp]
        total = sum(x.scaled[samp, ]*beta.0)
        d.term1 = exp(dnorm(total, log = TRUE) - pnorm(total, log = TRUE))
        d.term2 = exp(dnorm(total, log = TRUE) - pnorm(-total, log = TRUE))
```

```
        beta.1 = beta.0 + alpha/(1+eta*niter) * sum((y[samp]*d.term1 - (1-
y[samp])*d.term2))*x.scaled[samp , ]

        location = 1
        beta.scaled = .5*beta.1
        beta.scaled[1] = beta.1[1]
        beta.matrix[niter+1, ] = beta.scaled
        niter = niter + 1
      }
      test = TRUE
      end <- Sys.time()
      if(test){
      cat('beta0: ', beta.1[1], '\n')
      cat('beta1: ', beta.1[2], '\n')
      cat('beta2: ', beta.1[3], '\n')
      cat('beta3: ', beta.1[4], '\n')
      cat('beta4: ', beta.1[5], '\n')
      cat('beta5: ', beta.1[6], '\n')
      cat('beta6: ', beta.1[7], '\n')
      cat('beta7: ', beta.1[8], '\n')
      cat('iterations: ', niter, '\n')
      print(end - start)
      beta.m1 = beta.1
      time.m1 = end-start
      }
```

Here are plots of the log likelihood of $\beta^{(t)}$ at each point in the stochastic gradient algorithms. In the $m=1$ case I stored the likelihood at each $15^{\textrm{th}}$ iteration to save computation time.

As expected, we see an initial increase as the algorithm moves towards an MLE, and stabilization after the neighborhood of this MLE is discovered and the number of iterations increases.

```{r Likelihood graphs, echo=FALSE}
  beta.matrix.1 = beta.matrix[seq(from = 1, to = 30000, by = 15), ]
  likelihood.vec.2 = apply(beta.matrix.1, 1, likelihood.beta)
  plot(likelihood.vec.1, type = 'l', xlab = 'iteration', ylab = 'likelihood, m =
1000')
  plot(likelihood.vec.2, type = 'l', xlab = 'iteration', ylab = 'likelihood, m =
1')
```

Tuning α and η was challenging because the stochastic gradient algorithm was sensitive to these parameters. Too large values for α led to movement away from the correct values of the parameters. I found small values for α and smaller decaying rate η yielded more accurate results.

```
\newpage
#### Section 3 - Comparison of Results


\vspace{5mm}


\begin{center}
\begin{tabular}{ l r r}
Method & Execute Time & $||\beta-\hat{\beta}||$ \\
\hline
Newton-Raphson & `r round(time.newton, 2)` secs & `r round(sqrt(sum((true.beta -
beta.newton)**2)), 3)` \\
Coordinate Descent & `r round(time.coord, 2)` mins & `r round(sqrt(sum((true.beta
- beta.coord)**2)), 3)` \\
Stochastic Gradient $m = 1000$ & `r round(time.m1000, 2)` secs & `r
round(sqrt(sum((2*true.beta - beta.m1000)**2)), 3)`  \\
Stochastic Gradient $m = 1$ & `r round(time.m1, 2)` mins & `r
round(sqrt(sum((2*true.beta - beta.m1)**2)), 3)`\\
\end{tabular}
\end{center}

\vspace{5mm}

The table above summarizes the results of the four algorithms.  I used Euclidean
distance between $\beta$, the vector used to generate the data, and
$\hat{\beta}$, the MLE produced by the algorithm, to measure the accuracy of each
method.


In this case Newton-Raphson was both the most accurate and speediest algorithm.
This is due to $p$ being small.  Because Newton-Raphson requires the computation
the inverse of a $p\times p$ matrix at each step, a growing $p$ would
significantly slow down this approach.

The stochastic gradient algorithms involve only the computation of the gradient.
This computation is easily vectorized and therefore the time required to complete
this computation will not increase quickly as $p$ increases.

The coordinate descent algorithm can benefit from a stricter stopping condition.
As the algorithm approaches the correct parameters, the magnitude of the steps
each parameter takes diminishes rapidly.  Thus the algorithm may still be
approaching a better solution, but slowly enough that the relative change in the
log likelihood is very small.

If the stopping condition is made to be a relative change in the log likelihood
that is less than $10^{-7}$, the coordinate descent algorithm yields an esimate
much closer to the parameters used to generate the data:

```{r Coord Descent Strict, echo=FALSE}
beta.0 = rep(0, 8)
beta.1 = beta.0
error.vector = rep(1, p+1)
errors = c()
location = 1
l = 0
```
```

```
loop = TRUE

start <- Sys.time()
  while (loop){
    likelihood.0 = likelihood.beta(beta.0)
    sums = apply(x, 1, function(t)sum(t*beta.0))
    dd.term1 = dnorm(sums)*sums/pnorm(sums)
    dd.term2 = (dnorm(sums)/pnorm(sums))**2
    dd.term3 = dnorm(sums)*sums/pnorm(-sums)
    dd.term4 = (dnorm(sums)/pnorm(-sums))**2
    d.term1 = dnorm(sums)/pnorm(sums)
    d.term2 = dnorm(sums)/pnorm(-sums)

    beta.1[location] = beta.0[location] -
                       sum((y*d.term1 - (1-y)*d.term2)*x[ , location])/
                       sum( (-y*dd.term1 - y*dd.term2 + (1 - y)*dd.term3 - (1 -
y)*dd.term4 )*x[, location]*x[, location])

    likelihood.1 = likelihood.beta(beta.1)
    err = abs((likelihood.1 - likelihood.0)/likelihood.1)
    errors = append(errors, err)
    if(err < 10**-4){
      error.vector[location] = max(errors)
      errors = c()
      location = (location %% 8) + 1
    }
    if(all(error.vector < 10**-7)){
      l = l + 1
      end <- Sys.time()
      cat('beta0: ', beta.1[1], '\n')
      cat('beta1: ', beta.1[2], '\n')
      cat('beta2: ', beta.1[3], '\n')
      cat('beta3: ', beta.1[4], '\n')
      cat('beta4: ', beta.1[5], '\n')
      cat('beta5: ', beta.1[6], '\n')
      cat('beta6: ', beta.1[7], '\n')
      cat('beta7: ', beta.1[8], '\n')
      cat('iterations: ', l, '\n')
      print(end - start)
      loop = FALSE
    }
    beta.0 = beta.1
    l = l + 1
  }

beta.coord = beta.1
time.coord = end - start
```