Homework on optimization algorithms.

P8160 Advanced Statistical Computing

Problem 1:

Design an optmization algorithm to find the minimum of the continuously differentiable function

$$f(x) = -e^{-x}\sin(x)$$

on the closed interval $[0, 1.5]$. Write out your algorithm and implement it into **R**.

Answer: your answer starts here...

To find the minimum of a continuously function,we first make some changes to the function let

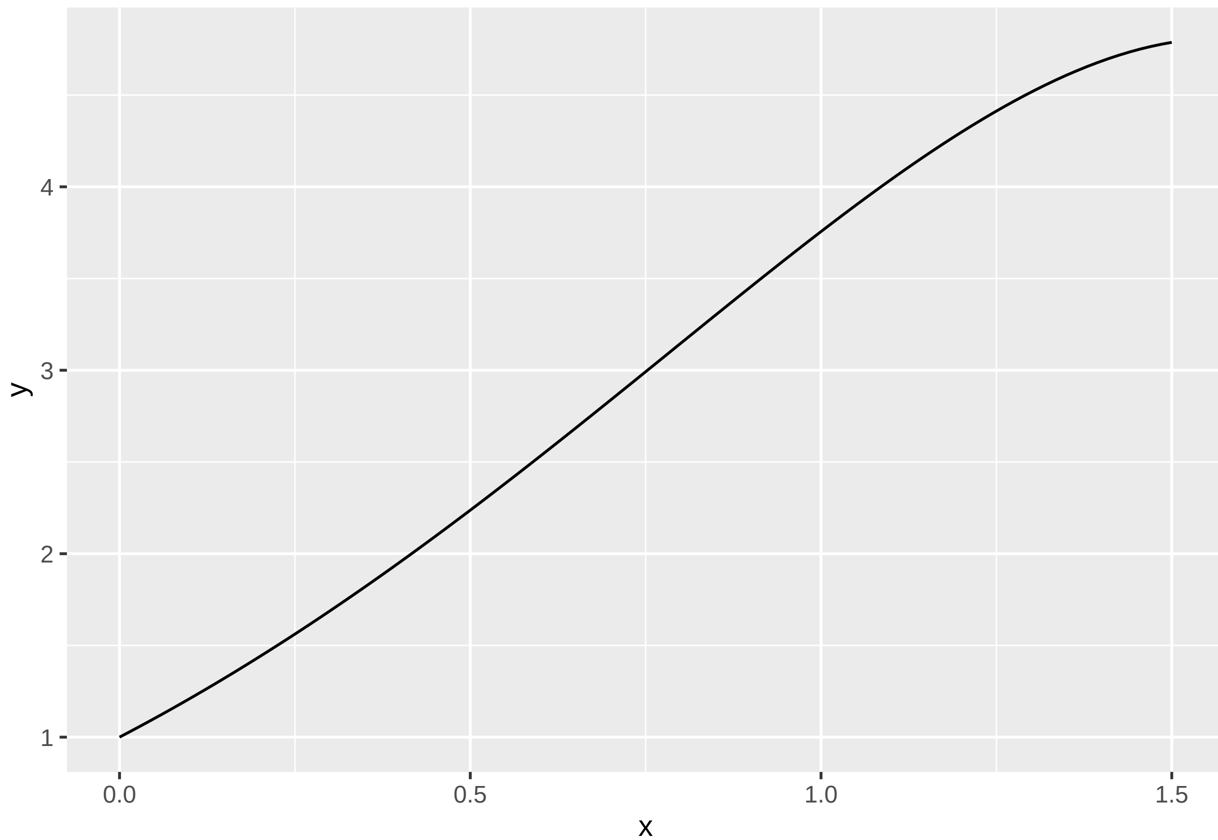
$$g(x) = e^{x}\sin(x)$$

and instead find the maximum of g(x).

The gradient of $g(x)$ is :

$$\nabla g(x) = e^{x}(\sin(x) + \cos(x))$$

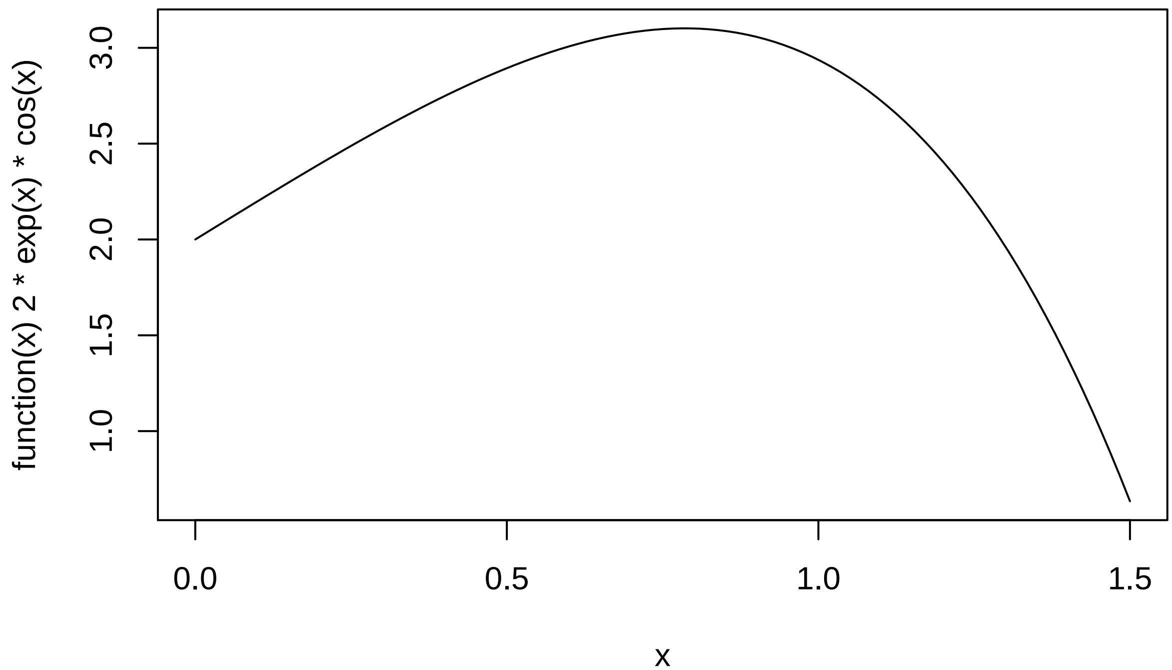
```
ggplot(tibble(x = seq(0,1.5,length = 10)),aes(x))+
  geom_function(fun = function(x) exp(x)*(sin(x)+cos(x)))
```



and the Hessian is:

$$\nabla^2 g(x) = 2e^x \cos(x)$$

```
plot(function(x) 2*exp(x)*cos(x),xlim = c(0,1.5))
```



the hessian is greater than 0 everywhere in [0,1.5], so we can't use Newton method.

```
goose_egg =
  function(
    fun,
    left = NULL,
    right = NULL,
    range = NULL,
    ratio = 0.618,
    tol = 10e-4,
    ...
  ){
    if (!any(left,right)){
      left = range[1]
      right = range[2]
    }

    mid_1 = left + ratio*(right - left)

    f_mid_1 = fun(mid_1)

    mid_2 = mid_1 + ratio*(right-mid_1)

    f_mid_2 = fun(mid_2)

    f_left = fun(left)
```

```r
    f_right = fun(right)

    i = 1

    while (abs(f_left - f_right)>tol && i<1000){
      i = i + 1
      if (f_mid_1 < f_mid_2) {
        f_left = f_mid_1
        left = mid_1
      } else {
        f_right = f_mid_2
        right = mid_2
      }
      mid_1 = left + ratio * (right - left)
      f_mid_1 = fun(mid_1)
      mid_2 = mid_1 + ratio * (right - mid_1)
      f_mid_2 = fun(mid_2)
    }
    return(mean(mid_1,mid_2))
  }
```

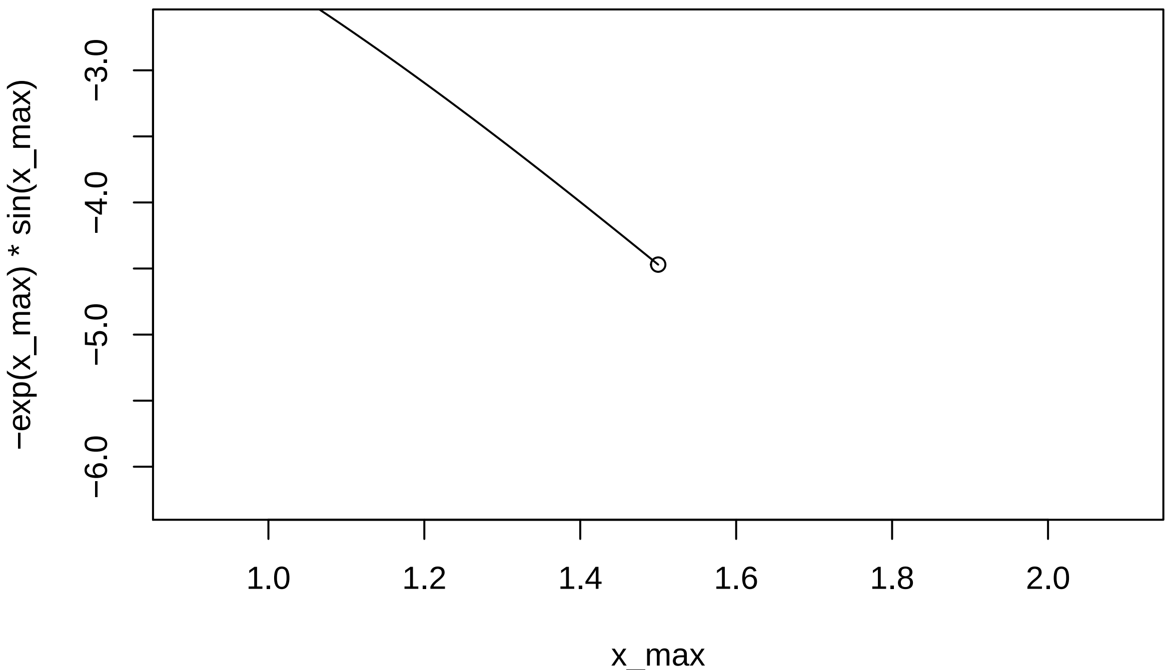
```r
x_max = goose_egg(function(x) exp(x)*sin(x),range = c(0,1.5))

print(x_max)
```

```
## [1] 1.499962
```

```r
plot(x_max,-exp(x_max)*sin(x_max))

plot(function(x) {-exp(x)*sin(x)}, xlim = c(0,1.5), add = T)
```



Problem 2:

The Poisson distribution, written as

$$P(Y = y) = \frac{\lambda^y e^{-\lambda}}{y!}$$

for $\lambda > 0$, is often used to model "count" data — e.g., the number of events in a given time period. A Poisson regression model states that

$$Y_i \sim \text{Poisson}(\lambda_i),$$

where

$$\log \lambda_i = \alpha + \beta x_i$$

for some explanatory variable x_i. The question is how to estimate α and β given a set of independent data $(x_1, Y_1), (x_2, Y_2), \ldots, (x_n, Y_n)$.

1. Generate a random sample (x_i, Y_i) with $n = 500$ from the Possion regression model above. You can choose the true parameters (α, β) and the distribution of X.
2. Write out the likelihood of your simulated data, and its Gradient and Hessian functions.
3. Develop a modify Newton-Raphson algorithm that allows the step-halving and re-direction steps to ensure ascent directions and monotone-increasing properties.
4. Write down your algorithm and implement it in R to estimate α and β from your simulated data.

Answer: your answer starts here...

2.1

```
print("hello world")
```

```
X = rbind(rep(1,500),rnorm(500))
Beta = runif(2)
lambda = exp(t(X)%*%Beta)
Y = map(lambda,~rpois(1,.x)) %>% unlist()
dat = list(y = Y, x=X)
ans = glm(Y~0+t(X),family = poisson())
```

2.2

- The log-likelihood of Poisson distribution is

$$l(Y;\lambda) = \sum\{y * log(\lambda) - \lambda - log(y!)\}$$

OR

$$l(Y;\alpha,\beta) = \sum\{y * (\alpha + x\beta) - exp(\alpha + x\beta) - log(y!)\}$$

- The Score funtion is

$$\nabla(Y;\alpha,\beta) = \frac{\partial}{\partial\lambda} l(Y;\lambda) = (\sum\{y - exp(\alpha + x\beta)\}, \sum\{y * x - x * exp(\alpha + x\beta)\})$$

- The hessian is

$$\nabla^2(Y;\lambda) = \frac{\partial^2}{\partial\lambda^2} l(Y;\lambda)$$

$$= \begin{pmatrix} \sum -exp(\alpha + x\beta) & \sum -x * exp(\alpha + x\beta) \\ \sum -x * exp(\alpha + x\beta) & \sum -x^2 * exp(\alpha + x\beta) \end{pmatrix}$$

which is negative defined everywhere.

```
Poisson =
  function(Y,X,
           theta_vec) {
    lambda = exp(t(X) %*% theta_vec)
    loglink = sum(Y * log(lambda) - lambda - log(factorial(Y)))

    fisher = var(Y * log(lambda) - lambda - log(factorial(Y)))

    gradient = c(sum(Y - lambda), sum(Y * X[2,] - X[2,] * lambda))

    hessian = matrix(c(
      sum(-lambda),
      sum(-X[2,] * lambda),
      sum(-X[2,] * lambda),
      sum((-X[2,] ^ 2) * lambda)
    ), ncol = 2)
```

```r
    return(list(
      loglink = loglink,
      fisher = fisher,
      gradient = gradient,
      hessian = hessian
    ))
  }

Poisson(Y,X,c(7,2))
```

```
## $loglink
## [1] -3974589
##
## $fisher
##           [,1]
## [1,] 821143964
##
## $gradient
## [1] -3980250 -7302954
##
## $hessian
##          [,1]      [,2]
## [1,] -3981198  -7303206
## [2,] -7303206 -15987659
```

2.3

the Newton method updating is:

$$\nabla g(x_{k+1}) = \nabla g(x_k) + \eta * \nabla^2 g(x_k)(x_{k+1} - x_k)$$

where η is the step size that ensure $\nabla g(x_{k+1}) > \nabla g(x_k)$

```r
#Develop a modify Newton-Raphson algorithm that allows the
#step-halving and
#re-direction steps
#to ensure ascent directions and monotone-increasing properties.

newton_update =
  function(fun,
           previous_theta,
           y,x,
           step_size = 1,
           optimizer = F,
           backtracking = T,
           tol = 1e-8) {
    #take previous gradient and a updated hessian, return update gradient with
    #backtracking
    #if (abs(fun(y,x,previous_theta)$loglink) == Inf) stop("Check your log-likelihood")
    trial = 0

    gradient = fun(y,x,previous_theta)$gradient
```

```
    if (is.function(optimizer)) {
      hessian = optimizer(y,x, fun,) # get H
    } else{
      if (is.numeric(optimizer)) {
        H = optimizer # use H
      } else{
        hessian = fun(y,x, previous_theta)$hessian
        H = solve(hessian)
        while (all(eigen(H)$values > 0)) {# eigen decomposition
          P = eigen(hessian)
          lambda = max(P$values)
          hessian =
            P$vectors %*% (diag(P$values) - (lambda + tol) * diag(length(P$values))) %*%
            solve(P$vectors)

          H = solve(hessian)
        }
      }
    }

    #updating
    cur_theta = previous_theta - step_size * H %*% gradient

    #backtracking
    while (backtracking & fun(y,x,cur_theta)$loglink < fun(y,x,previous_theta)$loglink & trial < 2000)
      step_size = step_size / 2

      trial = trial + 1 # avoild dead loops

      cur_theta = previous_theta - step_size * H %*% gradient
    }

    return(cur_theta)
  }

newton_update(fun = Poisson,previous_theta = c(7,2), y=Y,x =X)
```

```
##          [,1]
## [1,] 6.001291
## [2,] 1.999426
```

```
naive_newton =
  function(fun,
           init_theta = 1,
           y,x,
           tol = 1e-8,
           maxtiter = 2000,
           optimizer = F,
           ...) {

    f = fun(y,x,init_theta)

    if (any(is.null(f$loglink),
```

```r
            is.null(f$gradient),
            is.null(f$hessian))) {
      stop("fun input must return both gradient and hessian")
    }
    result = tibble()

    i = 0

    cur_theta = init_theta

    prevlog = -Inf # \nabla g(x_{k})

    while (any(abs(f$loglink)==Inf,abs(f$loglink - prevlog) > tol) && i < maxtiter) {
      i = i + 1
      prev_theta = cur_theta
      prevlog = f$loglink
      cur_theta = newton_update(fun, prev_theta,y,x)
      f = fun(y,x,cur_theta)
      result =
        rbind(result, tibble(
          iter = i,
          x_i = list(prev_theta),
          `g(x_i)` = prevlog
        ))
    }
    return(list(theta = cur_theta,result = result))
  }
```

```r
Beta_hat = naive_newton(Poisson,init_theta = c(7,2),Y,X)$theta

tibble(
  term = c("alpha", "beta"),
  theta = Beta,
  theta_hat = Beta_hat
) %>%
  knitr::kable()
```

term	theta	theta_hat
alpha	0.5854990	0.6064289
beta	0.2339566	0.2472194

Problem 3:

```
## Warning: Missing column names filled in: 'X33' [33]

## Parsed with column specification:
## cols(
##   .default = col_double(),
##   diagnosis = col_character(),
##   X33 = col_character()
## )
```

```
## See spec(...) for full column specifications.

## Warning: 569 parsing failures.
## row col   expected     actual                   file
##   1  -- 33 columns 32 columns './breast-cancer.csv'
##   2  -- 33 columns 32 columns './breast-cancer.csv'
##   3  -- 33 columns 32 columns './breast-cancer.csv'
##   4  -- 33 columns 32 columns './breast-cancer.csv'
##   5  -- 33 columns 32 columns './breast-cancer.csv'
## ... ... .......... .......... ......................
## See problems(...) for more details.
```

The data *breast-cancer.csv* have 569 row and 33 columns. The first column **ID** lables individual breast tissue images; The second column **Diagnonsis** indentifies if the image is coming from cancer tissue or benign cases (M=malignant, B = benign). There are 357 benign and 212 malignant cases. The other 30 columns correspond to mean, standard deviation and the largest values (points on the tails) of the distributions of the following 10 features computed for the cellnuclei;

- radius (mean of distances from center to points on the perimeter)
- texture (standard deviation of gray-scale values)
- perimeter
- area
- smoothness (local variation in radius lengths)
- compactness (perimeter$\hat{2}$ / area - 1.0)
- concavity (severity of concave portions of the contour)
- concave points (number of concave portions of the contour)
- symmetry
- fractal dimension ("coastline approximation" - 1)

The goal is to build a predictive model based on logistic regression to facilitate cancer diagnosis;

1. Build a logistic model to classify the images into malignant/benign, and write down your likelihood function, its gradient and Hessian matrix.
2. Build a logistic-LASSO model to select features, and implement a path-wise coordinate-wise optimization algorithm to obtain a path of solutions with a sequence of descending λ's.
3. Write a report to summarize your findings.

3

3.1

the data is a binomial outcome response, which follows a Bernoulli distribution, Using logit link, which

$$log(\frac{p}{1-p}) = X^T\beta$$

s.t

$$p = \frac{exp(X\beta)}{1 + exp(X\beta)}$$

the log-likelihood of Bernoulli is

$$l(Y;\beta) = \sum\{y * log(\frac{p}{1-p}) + log(1-p)\} = \sum\{y * X^T\beta - log(1 + exp(X^T\beta))\}$$

The gradient is

$$\nabla l(Y;\beta) = (\frac{\partial}{\partial \beta_i} l(Y;\beta)) = (\sum(y * x_i - \frac{x_i exp(X^T\beta)}{1 + exp(X^T\beta)})$$

and the hessian is

$$\nabla^2 l(Y;\beta) = (\sum -\frac{x_i * x_j * exp(X^T\beta)}{(1 + exp(X^T\beta)^2)})$$

```
Bernoulli =
  function(y,x,
           theta_vec){
    if (length(theta_vec)!=ncol(x)) stop("length of theta_vec must match dim of x")
    if (is.factor(y)) y = y %>% as.numeric()-1
    Y = y
    X = x
    loglink = sum(Y * X%*%theta_vec - log(1+exp(X%*%theta_vec)))

    if (abs(loglink) == Inf){
     stop("Choose a better starting value")
    }

    fisher = var(Y * X%*%theta_vec - log(1+exp(X%*%theta_vec)))

    X = x*1e-0

    gradient = map(1:length(theta_vec),
                   ~ sum(Y * X[, .x] - X[, .x] * exp(X %*% theta_vec) /
                           (1 + exp(X %*% theta_vec)))) %>% unlist()

    hessian =
      expand.grid(i = seq(1,length(theta_vec)),
                  j = seq(1,length(theta_vec))) %>%
      summarise(beta = map2(i,j,~sum(-X[,.x]*X[,.y]*exp(X%*%theta_vec)/(1+exp(X%*%theta_vec))^2))) %>%
      unnest(beta) %>%
      pull(beta)

    hessian = matrix(hessian,ncol = length(theta_vec))

    return(
      list(loglink = loglink,
           fisher = fisher,
           gradient  = gradient*1e+0,
           hessian = hessian*1e+0)
    )
  }

 Bernoulli(Y,X,rep(0,10))
```

```
## $loglink
## [1] -394.4007
##
## $fisher
##      [,1]
## [1,]    0
##
## $gradient
##  [1]     317.094500     907.665000   1707.730000 -21099.850000      5.600020
##  [6]      -1.094800      -8.820835     -4.736383     10.643850      4.577740
##
## $hessian
##               [,1]          [,2]          [,3]          [,4]          [,5]
##  [1,]   -30153.7946   -39461.4941  -196955.1304  -1489946.535  -194.8468530
##  [2,]   -39461.4941   -55556.7243  -257249.1049  -1865995.711  -264.2071359
##  [3,]  -196955.1304  -257249.1049 -1287036.1645  -9765528.587 -1270.7012542
##  [4,] -1489946.5348 -1865995.7110 -9765528.5865 -78593927.463 -9101.1459605
##  [5,]     -194.8469     -264.2071    -1270.7013     -9101.146    -1.3489220
##  [6,]     -223.0604     -293.9416    -1466.5402    -11035.815    -1.4997475
##  [7,]     -205.4499     -258.3854    -1358.7127    -11005.144    -1.3002989
##  [8,]     -114.2797     -141.1865     -753.9245     -6153.567    -0.7134547
##  [9,]     -366.0910     -498.2950    -2387.3801    -17083.935    -2.5137687
## [10,]     -125.0975     -171.9841     -815.1857     -5750.212    -0.8690283
##               [,6]          [,7]          [,8]          [,9]         [,10]
##  [1,] -2.230604e+02 -2.054499e+02  -114.2797367   -366.090984  -125.0975057
##  [2,] -2.939416e+02 -2.583854e+02  -141.1865362   -498.295027  -171.9840540
##  [3,] -1.466540e+03 -1.358713e+03  -753.9245464  -2387.380147  -815.1857356
##  [4,] -1.103581e+04 -1.100514e+04 -6153.5666261 -17083.934913 -5750.2116238
##  [5,] -1.499747e+00 -1.300299e+00    -0.7134547     -2.513769    -0.8690283
##  [6,] -1.944746e+00 -1.845979e+00    -0.9679417     -2.812793    -0.9620093
##  [7,] -1.845979e+00 -2.024132e+00    -1.0226591     -2.443757    -0.8201581
##  [8,] -9.679417e-01 -1.022659e+00    -0.5542199     -1.330521    -0.4434863
##  [9,] -2.812793e+00 -2.443757e+00    -1.3305214     -4.775310    -1.6315022
## [10,] -9.620093e-01 -8.201581e-01    -0.4434863     -1.631502    -0.5680471
```

```r
# Bernoulli(dat,ans$coefficients[-1] %>% as.vector())
```

3.2

```r
lasso_update_fit =
  function(fun, y,x, theta_vec = NaN,lambda = 1,maxiter = 200,tol = 1e-6,... ) {
    # data preprocessing
    if (is.factor(y)) y = y %>% as.numeric() -1
    y = y
    x = scale(x)
    xsd = attr(x,"scaled:scale") %>% as.vector()
    x = cbind(rep(1,length(y)),x) # add alpha
    xsd = append(1,xsd)
    # checking if intercept is include
    beta_0 = mean(y)/(1-mean(y)) %>% log()
    if (any(is.na(theta_vec))) theta_vec = rep(beta_0,ncol(x))
    if (length(theta_vec)<ncol(x)) theta_vec = append(beta_0,theta_vec)
```

```
    iter = 0

    soft_threshold =
      function(beta, lambda){
        beta = (abs(beta) > lambda) * (beta - sign(beta) * lambda) +(abs(beta) < lambda) * 0
        return(beta)}

    cur_result = fun(y,x, theta_vec)$loglink

    prev_result = -Inf

    result = tibble()
    while (any(abs(cur_result) == Inf,abs(cur_result - prev_result) > tol) && iter < maxiter) {
      prev_result = cur_result
      iter = iter + 1

      for (i in 1:length(theta_vec)) {
        #coordinate update beta not intercept
        theta_old = theta_vec[i]
        cur_f = fun(y,x, theta_vec)
        cur_result = cur_f$loglink
        gradient_i = cur_f$gradient[i]
        hessian_i = cur_f$hessian[i, i]
        H_i = solve(hessian_i)
        if (H_i>0) H_i = -1
        theta_new = theta_old - H_i %*% gradient_i
        if (i>1) theta_new = soft_threshold(theta_new, lambda)
        # don't penalize intercept
        theta_vec[[i]] = theta_new
      }

     # result = rbind(result,
     #                 tibble(
     #                   iteration = iter,
     #                   theta_id = list(1:length(theta_vec)),
     #                   theta = list(theta_vec*xsd),
     #                   L1 = fun(y,x, theta_vec)$loglink
     #                 ))
    }
    return(list(theta = theta_vec/xsd, result = result))
  }

lasso_update = function(fun, y,x, theta_vec = NaN,
                        lambda = exp(seq(from = 5, to = -10)), ...) {
  lambda = lambda %>% as.vector()
  result = tibble()
  for (lambda_i in lambda) {
    result = rbind(result,
                   tibble(
                     lamdba = lambda_i,
                     result = lasso_update_fit(fun, y, x, theta_vec = theta_vec, lambda = lambda_i)
                   ))
  }
```

```
  return(result)
}


b = glmnet(X,Y,family="binomial",lambda = 0.1)

tibble(
  lasso_01lambda = lasso_update_fit(Bernoulli,Y,X,lambda = 0.5)$theta,
  glmnet = coef(b) %>% as.vector(),
  lasso_0lambda = lasso_update_fit(Bernoulli,Y,X,lambda = 0)$theta,
  newton = naive_newton(Bernoulli, init_theta = rep(0, 11), Y, cbind(rep(1, length(
    Y
  )), X))$theta,
  glm = glm(diagnosis ~ ., data = breast, family = binomial())$coefficient %>% as.vector()
) %>%
  knitr::kable()
```

```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

lasso_01lambda	glmnet	lasso_0lambda	newton	glm
0.4088699	4.2816984	-0.2823106	7.35951761	7.3595176
0.0000000	0.0000000	2.0733985	2.04930490	2.0493049
-0.1386624	-0.0092337	-0.3810499	-0.38473434	-0.3847343
0.0000000	-0.0224573	-0.0338177	0.07151042	0.0715104
-0.0055739	0.0000000	-0.0323954	-0.03979620	-0.0397962
0.0000000	0.0000000	-78.4883724	-76.43227375	-76.4322738
0.0000000	0.0000000	7.2779110	1.46242225	1.4624223
0.0000000	0.0000000	-9.5658962	-8.46869976	-8.4686998
-72.0479395	-29.6555175	-62.6104777	-66.82175685	-66.8217568
0.0000000	0.0000000	-16.6576335	-16.27824232	-16.2782423
0.0000000	0.0000000	53.6963822	68.33702689	68.3370269

```
Qlasso =
  function(y,
           x,
           lambda = 0,
           maxiter = 500,
           tol = 1e-6,
           ...) {
    # data preparation
    if (is.factor(y))
      y = y %>% as.numeric() - 1
    y = y
    x = scale(x)
    xsd = attr(x, "scaled:scale") %>% as.vector()
    x = cbind(rep(1, length(y)), x) # add alpha
    xsd = append(1, xsd)

    # Manually choosing starting value
    theta_vec = rep(0, ncol(x))
```

```r
  #update part
  iter = 0

  # quadratic function

  qfun =
    function(y, x, theta_vec) {
      p_prev = exp(x %*% theta_vec) / (1 + exp(x %*% theta_vec)) # 1 * col matrix

      w_prev = p_prev * (1 - p_prev) # 1 * col matrix

      z_prev = x %*% theta_vec + (y - p_prev) / w_prev # row * 1 matrix

      loglink = -sum(w_prev * (z_prev - x %*% theta_vec) ^ 2) / (2 * length(y))
      #gradient
      gradient_prev =
        map(1:ncol(x),
            ~ sum(w_prev * (z_prev - x %*% theta_vec) * x[, .x]) / length(y)) %>%
        unlist()# col * 1 data

      # Hessian
      hessian_prev =
        expand.grid(i = 1:ncol(x),
                    j = 1:ncol(x)) %>%
        summarise(theta =
                    map2(i, j,
                         ~ -sum(w_prev * (x[, .x] * x[, .y])) / length(y))) %>%
        unnest(theta) %>%
        pull(theta) %>%
        matrix(., ncol = ncol(x))

      return(
        list(
          loglink = loglink,
          p = p_prev,
          w = w_prev,
          z = z_prev,
          gradient = gradient_prev,
          hessian = hessian_prev
        )
      )
    }

  cur_result = qfun(y, x, theta_vec)$loglink

  if (abs(cur_result) == Inf)
    stop("Diverge at starting value")

  prev_result = -Inf

  while (abs(cur_result - prev_result) > tol
         && iter < maxiter) {
    iter = iter + 1
```

```r
      prev_result = cur_result

      for (i in 1:length(theta_vec)) {
        #weight
        fun_prev = qfun(y, x, theta_vec)
        p_prev = fun_prev$p
        z_prev = fun_prev$z
        w_prev = fun_prev$w
        gradient_prev = fun_prev$gradient
        hessian_prev = fun_prev$hessian
        H_prev = solve(hessian_prev[i, i])

        #update
        cur_theta =
          theta_vec[[i]] - H_prev * gradient_prev[[i]]

        #soft-threshod,skip penalize intercept
        if (i > 1)
          cur_theta =
          (abs(cur_theta) > lambda) * (cur_theta - sign(cur_theta) * lambda) +
          (abs(cur_theta) < lambda) * 0

        #update theta
        theta_vec[[i]] = cur_theta
      }

      cur_result = qfun(y, x, theta_vec)$loglink

    }

    return(theta_vec/xsd)
  }

tibble(glm = glm(Y~X,family = binomial())$coefficient,
       Qlasso_0lambda = Qlasso(Y,X),
       Qlasso_01lambda = Qlasso(Y,X,0.1),
       glmnet_01lambda = coef(glmnet(X,Y,"binomial",lambda = 0.1)) %>% as.vector()
       )
```

```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

## # A tibble: 11 x 4
##         glm Qlasso_0lambda Qlasso_01lambda glmnet_01lambda
##       <dbl>          <dbl>           <dbl>           <dbl>
##  1   7.36          -0.237           0.372           4.28
##  2   2.05           1.80            0               0
##  3  -0.385         -0.381          -0.308          -0.00923
##  4   0.0715        -0.0100          0              -0.0225
##  5  -0.0398        -0.0310         -0.00939         0
##  6 -76.4          -78.1           -35.9             0
##  7   1.46           7.03            4.06            0
##  8  -8.47          -9.81           -3.31            0
##  9 -66.8          -62.4           -70.4           -29.7
```

```
## 10 -16.3           -16.7           -11.5               0
## 11  68.3            52.4             0                 0
```