HW 2

Bryan Mui - UID 506021334 - 28 April 2025

Loaded packages: ggplot2, tidyverse (include = false for this chunk)

Reading the dataset:

```r
data <- read_csv("dataset-logistic-regression.csv")
```

```
Rows: 10000 Columns: 101
-- Column specification --------------------------------------------------------
Delimiter: ","
dbl (101): y, X1, X2, X3, X4, X5, X6, X7, X8, X9, X10, X11, X12, X13, X14, X...

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```r
head(data, n = 25)
```

```
# A tibble: 25 x 101
       y      X1      X2     X3     X4     X5     X6      X7     X8      X9
   <dbl>   <dbl>   <dbl>  <dbl>  <dbl>  <dbl>  <dbl>   <dbl>  <dbl>   <dbl>
 1     1 -0.0895  0.450   1.71   0.657 -0.392  1.24   0.895   1.13  -0.0117
 2     1 -0.0943  0.281  -0.147 -0.701  0.400 -0.210  0.677  -0.440  0.458
 3     0 -0.431  -0.445  -0.777 -0.832 -2.26  -1.62  -1.98   -1.67  -1.15
 4     0  0.644   0.0817 -0.448  0.852 -1.02   0.671  0.299   0.145 -0.205
 5     1 -0.919  -0.0241  0.807 -0.612 -0.498  0.350  1.12    0.242 -0.947
 6     0 -1.89   -1.11   -0.210  0.161 -1.34  -2.04  -0.0135 -1.39  -1.31
 7     0 -1.34   -0.804   0.322 -0.110  0.624 -0.329 -0.432  -0.191  0.171
 8     1  0.329   0.468   0.719  0.588  1.71   1.39   0.603   0.650  0.161
 9     0  0.332   1.42   -0.431  1.02   0.484  0.348  0.474   1.26  -0.479
10     0 -0.311   0.0193  0.168 -0.346  0.626 -0.704 -0.290   0.680 -0.0453
# i 15 more rows
```

```
# i 91 more variables: X10 <dbl>, X11 <dbl>, X12 <dbl>, X13 <dbl>, X14 <dbl>,
#   X15 <dbl>, X16 <dbl>, X17 <dbl>, X18 <dbl>, X19 <dbl>, X20 <dbl>,
#   X21 <dbl>, X22 <dbl>, X23 <dbl>, X24 <dbl>, X25 <dbl>, X26 <dbl>,
#   X27 <dbl>, X28 <dbl>, X29 <dbl>, X30 <dbl>, X31 <dbl>, X32 <dbl>,
#   X33 <dbl>, X34 <dbl>, X35 <dbl>, X36 <dbl>, X37 <dbl>, X38 <dbl>,
#   X39 <dbl>, X40 <dbl>, X41 <dbl>, X42 <dbl>, X43 <dbl>, X44 <dbl>, ...
```

```r
set.seed(777)
```

Our data set has 10000 observations, 1 binary outcome variable y, and 100 predictor variables X1-X100

Separating into X matrix and y vector:

```r
X <- data %>%
  select(-y)
y <- data %>%
  select(y)
```

Problem 1

Part (α)

The optimization problem is to minimize the log-likelihood function. From there we will get the objective function and gradient function

From the slides in class we have:

$$\min_{\beta}(-\ell(\beta)) = \frac{1}{m}\sum_{i=1}^{m} f_i(\beta)$$

and the equation for $f_i(\beta)$:

$$f_i(\beta) = -y_i(x_i^{\intercal}\beta) + log(1 + exp(x_i^{\intercal}\beta))$$

For the objective function, we get:

$$f(\beta) = \frac{1}{m}\sum_{i=1}^{m}[-y_i(x_i^{\intercal}\beta) + log(1 + exp(x_i^{\intercal}\beta))]$$

We can matricize the objective function to

$$\boxed{f(\beta) = \frac{1}{m}[-y^{\mathsf{T}}(X\beta) + \mathbf{1}^{\mathsf{T}} log(1 + exp(X\beta))]}$$

We also have the gradient function:

$$\nabla f(x) = \frac{1}{m}\sum_{i=1}^{m} \nabla f_i(x)$$

and

$$\nabla_{\beta} f_i(\beta) = [\sigma(x_i^{\mathsf{T}}\beta) - y_i] \cdot x_i$$

where $\sigma(z) = \frac{1}{1+exp(-z)}$ as the logistic sigmoid function, therefore:

$$\nabla f(x) = \frac{1}{m}\sum_{i=1}^{m} \nabla f_i(x),\ \nabla_{\beta} f_i(\beta) = [\sigma(x_i^{\mathsf{T}}\beta) - y_i] \cdot x_i$$

$$\nabla f(\beta) = \frac{1}{m}\sum_{i=1}^{m}[\sigma(x_i^{\mathsf{T}}\beta) - y_i] \cdot x_i$$

And we can also matricize this:

$$\boxed{\nabla f(\beta) = \frac{1}{m} X^{\mathsf{T}}[\sigma(X\beta) - y], \quad \sigma(z) = \frac{1}{1 + exp(-z)}}$$

Therefore our gradient descent update step is(for constant step size):

$$\boxed{\beta_{k+1} = \beta_k - \eta \nabla f(\beta_k)}$$

Implement the following algorithms to obtain estimates of the regression coefficients $\boldsymbol{\beta}$:

(1) Gradient descent with backtracking line search

Algorithm; Backtracking Line Search:

Params:

- Set $\eta^0 > 0$(usually a large value ~1),
- Set $\eta_1 = \eta^0$
- Set $\epsilon \in (0,1), \tau \in (0,1)$, where ϵ and τ are used to modify step size

Repeat:

- At iteration k, set $\eta_k < -\eta_{k-1}$

    1. Check whether the Armijo Condition holds:

    $$h(\eta_k) \leq h(0) + \epsilon \eta_k h'(0)$$

    where $h(\eta_k) = f(x_k) - \eta_k \nabla f(x_k)$,
    and $h(0) = f(x_k)$,
    and $h'(0) = -||\nabla(x_k)||^2$

    2.

    - If yes(condition holds), terminate and keep η_k
    - If no, set $\eta_k = \tau \eta_k$ and go to Step 1

Stopping criteria: Stop if $||x_k - x_{k+1}|| \leq tol$ (change in parameters is small)

Implement BLS

```r
# logistic gradient descent w/ bls
log_bls <- function(X, y, tol = 1e-6, max_iter = 10000, epsilon = 0.5, tau =
↪  0.5) {
  # Initialize
  n <- nrow(X)
  p <- ncol(X)
  x <- as.matrix(X)
  y <- as.matrix(y)
  beta <- as.matrix(rep(0, p))
  obj_values <- numeric(max_iter)
  eta_values <- numeric(max_iter)  # To store eta values used each iteration
  beta_values <- list() # To store beta values used each iteration
  eta_bt <- 1  # Initial step size for backtracking

  # Objective function: negative log-likelihood
```

```r
# input: Beta vector, x matrix, y matrix
# output: scalar objective func value
# comments: We want to minimize this function for logit regression
obj_function <- function(beta, x, y) {
  m <- nrow(x)
  z <- x %*% beta
  (1 / m) * (-(t(y) %*% z) + sum(log(1 + exp(z))))
}

# Gradient function
# input: Beta vector, x matrix, y matrix
# output: gradient vector in the dimension of nrow(Beta) x 1
# comments: We use this for gradient descent
gradient <- function(beta, x, y) {
  m <- nrow(x)                          # define m
  sig <- function(z) 1 / (1 + exp(-z))  # sigmoid function
  (1 / m) * (t(x) %*% (sig(x %*% beta) - y))
}

# Algorithm:
for (iter in 1:max_iter) {
  grad <- gradient(beta, x, y)

  #cat("iter ", iter, "\n")

  # backtracking step
  current_obj <- obj_function(beta, x, y)
  grad_norm_sq <- sum(grad^2)

  beta_new <- beta - eta_bt * grad

  while (obj_function(beta_new, x, y) > current_obj - epsilon * eta_bt *
   ↪  grad_norm_sq) {
    eta_bt <- tau * eta_bt
    beta_new <- beta - eta_bt * grad
  }

  # save values to the matrix
  eta_values[iter] <- eta_bt
  obj_values[iter] <- obj_function(beta_new, x, y)
  beta_values[[iter]] <- beta_new

  if (sqrt(sum((beta_new - beta)^2)) < tol) {
```

```
    # set the vector ranges and break
    beta <- beta_new
    obj_values <- obj_values[1:iter]
    eta_values <- eta_values[1:iter]
    beta_values <- beta_values[1:iter]
    break
  }

  beta <- beta_new
}

return(list(beta = beta, obj_values = obj_values, eta_values = eta_values,
  ↪ beta_values = beta_values))
}
```

TESTING: BLS

```
log_reg_bls <- log_bls(X, y, tol=1e-6, max_iter=10000, epsilon=0.5, tau=0.5)
```

```
cat("betas \n")
```

```
betas
```

```
print(log_reg_bls$beta)
```

```
                  y
X1   -0.1418188273
X2   -0.0601340162
X3    0.1588169528
X4    0.1328223189
X5   -0.0480437781
X6    0.0992481092
X7    0.1189707785
X8    0.1165560855
X9    0.0121222291
X10   0.0002641372
X11   0.0440526577
X12  -0.1793886158
X13  -0.0107332284
```

X14	-0.1230510680
X15	0.0724799230
X16	0.0571868940
X17	0.1299458439
X18	0.1249113906
X19	-0.0018170795
X20	0.1248825007
X21	-0.0107845610
X22	-0.1431801553
X23	-0.1094846603
X24	0.0576435159
X25	-0.1190174922
X26	0.0164879978
X27	-0.0977482724
X28	0.1544632196
X29	-0.0276524076
X30	0.0164226883
X31	-0.0589010945
X32	0.0205242099
X33	0.1352153619
X34	-0.0301792708
X35	-0.0097106467
X36	0.0631274232
X37	0.1972595891
X38	0.0932479560
X39	0.1242393813
X40	0.1466042152
X41	0.1112967707
X42	-0.1226544766
X43	-0.0374866338
X44	-0.0155583465
X45	-0.0103256878
X46	-0.1807311531
X47	0.0122916067
X48	0.0309436582
X49	0.0257891274
X50	0.1230837280
X51	-0.0237134869
X52	-0.0136672407
X53	0.0802510780
X54	0.1695795679
X55	0.1711403640
X56	-0.0447703054

X57	-0.0407325139
X58	-0.0768578382
X59	0.0786448045
X60	-0.1192193182
X61	-0.0080431756
X62	0.0701535429
X63	0.0295238798
X64	-0.1090225592
X65	0.0633967271
X66	-0.1450871355
X67	0.1404424947
X68	0.0649021774
X69	-0.1595801011
X70	0.1128079446
X71	0.1888668197
X72	0.0920649207
X73	-0.0647758044
X74	-0.0684344716
X75	0.2306707321
X76	-0.1312078759
X77	0.0301767178
X78	-0.0742090881
X79	0.0695790861
X80	-0.0273839196
X81	0.0183730389
X82	0.0555339156
X83	-0.0196159895
X84	-0.0119020076
X85	0.0981161430
X86	0.1724354285
X87	0.0832570899
X88	-0.0070115810
X89	0.0720539875
X90	0.0779093972
X91	0.0026928031
X92	-0.1223692130
X93	0.0073627318
X94	-0.0996425700
X95	-0.0485788118
X96	0.0338587696
X97	0.1496954257
X98	0.1702285222
X99	0.0197714549

```
X100   0.0070161693
```

```r
cat("The function converged after", length(log_reg_bls$obj_values), "
↪  iterations \n")
```

```
The function converged after 1909  iterations
```

```r
cat("Eta Vals: \n")
```

```
Eta Vals:
```

```r
print(log_reg_bls$eta_values[1:50])
```

```
 [1] 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625
[11] 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625
[21] 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625
[31] 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625
[41] 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625 0.0625
```

```r
cat("Objective Function vals \n")
```

```
Objective Function vals
```

```r
print(log_reg_bls$obj_values[1:50])
```

```
 [1] 0.5642463 0.5491551 0.5446388 0.5427888 0.5418291 0.5412092 0.5407301
 [8] 0.5403132 0.5399257 0.5395537 0.5391908 0.5388343 0.5384829 0.5381361
[15] 0.5377934 0.5374547 0.5371200 0.5367892 0.5364622 0.5361389 0.5358194
[22] 0.5355035 0.5351913 0.5348826 0.5345775 0.5342759 0.5339777 0.5336830
[29] 0.5333916 0.5331036 0.5328188 0.5325373 0.5322591 0.5319840 0.5317120
[36] 0.5314432 0.5311774 0.5309146 0.5306549 0.5303981 0.5301442 0.5298932
[43] 0.5296451 0.5293997 0.5291572 0.5289174 0.5286804 0.5284460 0.5282142
[50] 0.5279851
```

(2) Gradient descent with backtracking line search and Nesterov momentum

Nesterov is simply BLS with a special way to select the momentum ξ,

We set ξ to:

$$\frac{k-1}{k+2}$$

where k is the iteration index

Algorithm(Nesterov Momentum with BLS)

Params:

- Set $\eta^0 > 0$(usually a large value ~1),
- Set $\eta_1 = \eta^0$
- Set $\epsilon \in (0,1), \tau \in (0,1)$, where ϵ and τ are used to modify step size

Repeat:

- At iteration k, set $\eta_k < -\eta_{k-1}$, update with

$$\boxed{x_{k+1} = y_k - \eta_k \nabla(f(y_k)), \quad y_k = x_k + \xi(x_k - x_{k-1}), \quad \xi = \frac{k-1}{k+2}}$$

- Check the next setting of η:

  1. Check whether the Armijo Condition holds:

$$h(\eta_k) \leq h(0) + \epsilon \eta_k h'(0)$$

  where $h(\eta_k) = f(x_k) - \eta_k \nabla f(x_k)$,
  and $h(0) = f(x_k)$,
  and $h'(0) = -||\nabla(x_k)||^2$

  2.

     - If yes(condition holds), terminate and keep η_k
     - If no, set $\eta_k = \tau\eta_k$ and go to Step 1

Stopping criteria: Stop if $||x_k - x_{k+1}|| \leq tol$ (change in parameters is small)

Implement BLS Nesterov

```r
# logistic gradient descent w/ bls nesterov
log_bls_n <- function(X, y, tol = 1e-6, max_iter = 10000, epsilon = 0.5, tau
 ↪  = 0.8) {
  # Initialize
  n <- nrow(X)
  p <- ncol(X)
  x <- as.matrix(X)
  y <- as.matrix(y)
  beta <- as.matrix(rep(0, p))
  obj_values <- numeric(max_iter)
  eta_values <- numeric(max_iter)  # To store eta values used each iteration
  beta_values <- list() # To store beta values used each iteration
  eta_bt <- 1  # Initial step size for backtracking

  # Objective function: negative log-likelihood
  # input: Beta vector, x matrix, y matrix
  # output: scalar objective func value
  # comments: We want to minimize this function for logit regression
  obj_function <- function(beta, x, y) {
    m <- nrow(x)
    z <- x %*% beta
    (1 / m) * (-(t(y) %*% z) + sum(log(1 + exp(z))))
  }

  # Gradient function
  # input: Beta vector, x matrix, y matrix
  # output: gradient vector in the dimension of nrow(Beta) x 1
  # comments: We use this for gradient descent
  gradient <- function(beta, x, y) {
    m <- nrow(x)                          # define m
    sig <- function(z) 1 / (1 + exp(-z))  # sigmoid function
    (1 / m) * (t(x) %*% (sig(x %*% beta) - y))
  }

  # Algorithm:
  for (iter in 1:max_iter) {
    grad <- gradient(beta, x, y)

    #cat("iter ", iter, "\n")

    # backtracking step
    current_obj <- obj_function(beta, x, y)
    grad_norm_sq <- sum(grad^2)
```

```r
    if(iter == 1) {
      eta_bt <- 1
      y_k <- beta
    } else {
      beta_prev <- beta_values[[iter - 1]]
      xi <- (iter + 1) / (iter + 2)
      y_k <- beta + xi * ((beta - beta_prev))
    }

    beta_new <- y_k - eta_bt * grad

    while (obj_function(beta_new, x, y) > current_obj - epsilon * eta_bt *
     ↪  grad_norm_sq) {
      eta_bt <- tau * eta_bt
      beta_new <- beta - eta_bt * grad
    }

    # save values to the matrix
    eta_values[iter] <- eta_bt
    obj_values[iter] <- obj_function(beta_new, x, y)
    beta_values[[iter]] <- beta_new

    if (sqrt(sum((beta_new - beta)^2)) < tol) {
      # set the vector ranges and break
      beta <- beta_new
      obj_values <- obj_values[1:iter]
      eta_values <- eta_values[1:iter]
      beta_values <- beta_values[1:iter]
      break
    }

    beta <- beta_new
  }

  return(list(beta = beta, obj_values = obj_values, eta_values = eta_values,
   ↪  beta_values = beta_values))
}
```

TESTING: BLS

```r
log_reg_bls_n <- log_bls_n(X, y, tol=1e-6, max_iter=10000, epsilon=0.5,
  tau=0.8)
```

PRINTING OUTPUT

```r
cat("betas \n")
```

```
betas
```

```r
print(log_reg_bls_n$beta)
```

```
                y
X1   -0.1418263794
X2   -0.0601388591
X3    0.1588267497
X4    0.1328279392
X5   -0.0480506394
X6    0.0992548055
X7    0.1189794773
X8    0.1165617782
X9    0.0121210900
X10   0.0002652682
X11   0.0440525896
X12  -0.1794038315
X13  -0.0107339373
X14  -0.1230560308
X15   0.0724838710
X16   0.0571919899
X17   0.1299511146
X18   0.1249158303
X19  -0.0018154954
X20   0.1248931359
X21  -0.0107871202
X22  -0.1431957846
X23  -0.1094930269
X24   0.0576439569
X25  -0.1190248307
X26   0.0164864074
X27  -0.0977571692
X28   0.1544691126
```

```
X29  -0.0276528501
X30   0.0164184648
X31  -0.0589073210
X32   0.0205289418
X33   0.1352223830
X34  -0.0301846415
X35  -0.0097152706
X36   0.0631324820
X37   0.1972701788
X38   0.0932518671
X39   0.1242469201
X40   0.1466077067
X41   0.1113013302
X42  -0.1226607166
X43  -0.0374935885
X44  -0.0155599449
X45  -0.0103222178
X46  -0.1807432316
X47   0.0122929903
X48   0.0309476137
X49   0.0257875173
X50   0.1230898353
X51  -0.0237110270
X52  -0.0136673590
X53   0.0802556361
X54   0.1695903968
X55   0.1711505641
X56  -0.0447755635
X57  -0.0407377293
X58  -0.0768652234
X59   0.0786463310
X60  -0.1192273578
X61  -0.0080502803
X62   0.0701567008
X63   0.0295264284
X64  -0.1090289520
X65   0.0633998643
X66  -0.1450928496
X67   0.1404497219
X68   0.0649048905
X69  -0.1595896445
X70   0.1128140054
X71   0.1888821288
```

```
X72   0.0920697212
X73  -0.0647787849
X74  -0.0684414958
X75   0.2306799570
X76  -0.1312182054
X77   0.0301762811
X78  -0.0742102167
X79   0.0695810289
X80  -0.0273871826
X81   0.0183748140
X82   0.0555414970
X83  -0.0196157885
X84  -0.0119065906
X85   0.0981180432
X86   0.1724480190
X87   0.0832605298
X88  -0.0070189478
X89   0.0720560185
X90   0.0779116425
X91   0.0026900656
X92  -0.1223807711
X93   0.0073624203
X94  -0.0996496631
X95  -0.0485854423
X96   0.0338563238
X97   0.1497058189
X98   0.1702384964
X99   0.0197774565
X100  0.0070126286
```

```r
cat("The function converged after", length(log_reg_bls_n$obj_values), "
 ↪  iterations \n")
```

```
The function converged after 1443  iterations
```

```r
cat("Eta Vals: \n")
```

```
Eta Vals:
```

```r
print(log_reg_bls_n$eta_values[1:50])
```

```
 [1] 0.08589935 0.08589935 0.08589935 0.08589935 0.08589935 0.08589935
 [7] 0.08589935 0.08589935 0.08589935 0.08589935 0.08589935 0.08589935
[13] 0.08589935 0.08589935 0.08589935 0.08589935 0.08589935 0.08589935
[19] 0.08589935 0.08589935 0.08589935 0.08589935 0.08589935 0.08589935
[25] 0.08589935 0.08589935 0.08589935 0.08589935 0.08589935 0.08589935
[31] 0.08589935 0.08589935 0.08589935 0.08589935 0.08589935 0.08589935
[37] 0.08589935 0.08589935 0.08589935 0.08589935 0.08589935 0.08589935
[43] 0.08589935 0.08589935 0.08589935 0.08589935 0.08589935 0.08589935
[49] 0.08589935 0.08589935
```

```r
cat("Objective Function vals \n")
```

```
Objective Function vals
```

```r
print(log_reg_bls_n$obj_values[1:50])
```

```
 [1] 0.5475063 0.5433692 0.5420351 0.5412959 0.5407164 0.5401878 0.5396798
 [8] 0.5391836 0.5386963 0.5382171 0.5377455 0.5372814 0.5368247 0.5363752
[15] 0.5359328 0.5354973 0.5350688 0.5346469 0.5342317 0.5338231 0.5334209
[22] 0.5330250 0.5326353 0.5322517 0.5318741 0.5315025 0.5311367 0.5307766
[29] 0.5304221 0.5300732 0.5297297 0.5293916 0.5290587 0.5287310 0.5284085
[36] 0.5280909 0.5277783 0.5274705 0.5271675 0.5268692 0.5265755 0.5262864
[43] 0.5260017 0.5257214 0.5254454 0.5251737 0.5249062 0.5246428 0.5243834
[50] 0.5241280
```

(3) Gradient descent with AMSGrad-ADAM momentum

(no backtracking line search, since AMSGrad-ADAM adjusts step sizes per parameter using momentum and adaptive scaling)

AMSGrad-ADAM is a special way to adjust the step size intelligently:

$$
\begin{aligned}
m_k &= \beta_1 m_{k-1} + (1-\beta_1)G_k, \quad m_0 = 0, \quad G_k = \nabla f(x_k), \quad \beta_1 \in (0, \beta_2) \\
z_k &= \beta_2 z_{k-1} + (1-\beta_2)(G_k \odot G_k), \quad \beta_2 \in (0,1), \quad z_0 = 0 \\
\hat{m}_k &= \frac{m_k}{1-\beta_1^k} \quad \text{(exponentate at ktth iteration)} \\
\hat{z}_k &= \max(\hat{z}_{k-1}, z_k), \quad \hat{z}_0 = 0 \\
\tilde{z}_k(i) &= \frac{1}{\sqrt{\hat{z}_k(i)} + \epsilon} \\
\mathbf{x_{k+1}} &= \boxed{x_k - \eta(\tilde{z}_k \odot \hat{m}_k), \quad \eta > 0}
\end{aligned}
$$

Implement AMSGRAD-ADAM

```
# logistic gradient descent AMSGRAD-ADAM
log_adam <- function(X, y, tol = 1e-6, max_iter = 10000, eta = 1, epsilon =
↪  1e-8, b_1 = 0.9, b_2 = 0.999) {
  # Initialize
  n <- nrow(X)
  p <- ncol(X)
  x <- as.matrix(X)
  y <- as.matrix(y)
  beta <- as.matrix(rep(0, p))
  obj_values <- numeric(max_iter)
  eta_values <- numeric(max_iter)  # To store eta values used each iteration
  beta_values <- list() # To store beta values used each iteration
  eta_bt <- 1  # Initial step size for backtracking

  # Objective function: negative log-likelihood
  # input: Beta vector, x matrix, y matrix
  # output: scalar objective func value
  # comments: We want to minimize this function for logit regression
  obj_function <- function(beta, x, y) {
    m <- nrow(x)
    z <- x %*% beta
    (1 / m) * (-(t(y) %*% z) + sum(log(1 + exp(z))))
  }

  # Gradient function
  # input: Beta vector, x matrix, y matrix
  # output: gradient vector in the dimension of nrow(Beta) x 1
  # comments: We use this for gradient descent
  gradient <- function(beta, x, y) {
    m <- nrow(x)                           # define m
```

```
  sig <- function(z) 1 / (1 + exp(-z))  # sigmoid function
  (1 / m) * (t(x) %*% (sig(x %*% beta) - y))
}

# Algorithm:
for (iter in 1:max_iter) {
  grad <- gradient(beta, x, y)

  #cat("iter ", iter, "\n")

  # ADAM step
  if (iter == 1) {
    m_k <- (1 - b_1) * grad
    z_k <- (1 - b_2) * grad^2
    m_hat_k <- m_k / (1 - b_1^iter)
    z_hat_k <- pmax(0, z_k)
    z_tild_k <- 1 / (sqrt(z_hat_k) + epsilon)
  } else {
    m_k <- b_1 * m_k_prev + (1 - b_1) * grad
    z_k <- b_2 * z_k_prev + (1 - b_2) * grad^2
    m_hat_k <- m_k / (1 - b_1^iter)
    z_hat_k <- pmax(z_hat_k_prev, z_k)
    z_tild_k <- 1 / (sqrt(z_hat_k) + epsilon)
  }

  beta_new <- beta - eta * (z_tild_k * m_hat_k)

  # current_obj <- obj_function(beta, x, y)
  # grad_norm_sq <- sum(grad^2)
  #
  # if(iter == 1) {
  #   eta_bt <- 1
  #   y_k <- beta
  # } else {
  #   beta_prev <- beta_values[[iter - 1]]
  #   xi <- (iter + 1) / (iter + 2)
  #   y_k <- beta + xi * (beta - beta_prev)
  # }
  #
  # beta_new <- y_k - eta_bt * grad
  #
  # while (obj_function(beta_new, x, y) > current_obj - epsilon * eta_bt *
  ↪  grad_norm_sq) {
```

```r
    #   eta_bt <- tau * eta_bt
    #   beta_new <- beta - eta_bt * grad
    # }

    # save values to the matrix
    eta_values[iter] <- eta_bt
    obj_values[iter] <- obj_function(beta_new, x, y)
    beta_values[[iter]] <- beta_new

    if (sqrt(sum((beta_new - beta)^2)) < tol) {
      # set the vector ranges and break
      beta <- beta_new
      obj_values <- obj_values[1:iter]
      eta_values <- eta_values[1:iter]
      beta_values <- beta_values[1:iter]
      break
    }

    beta <- beta_new
    z_k_prev <- z_k
    m_k_prev <- m_k
    z_hat_k_prev <- z_hat_k
  }

  return(list(beta = beta, obj_values = obj_values, eta_values = eta_values,
   ↪  beta_values = beta_values))
}
```

TESTING: AMSGRAD-ADAM

```r
log_reg_adam <- log_adam(X, y, tol = 1e-6, max_iter = 10000, eta = 0.1,
 ↪  epsilon = 1e-8, b_1 = 0.9, b_2 = 0.999)
```

PRINTING OUTPUT

```r
cat("betas \n")
```

```
betas
```

```r
print(log_reg_adam$beta)
```

```
                y
X1   -0.1418454849
X2   -0.0601519765
X3    0.1588541105
X4    0.1328432036
X5   -0.0480698927
X6    0.0992738274
X7    0.1190031009
X8    0.1165771060
X9    0.0121186568
X10   0.0002692862
X11   0.0440526339
X12  -0.1794460643
X13  -0.0107336589
X14  -0.1230683354
X15   0.0724951933
X16   0.0572069676
X17   0.1299658957
X18   0.1249286333
X19  -0.0018093352
X20   0.1249220832
X21  -0.0107937892
X22  -0.1432400037
X23  -0.1095158039
X24   0.0576442709
X25  -0.1190438042
X26   0.0164821239
X27  -0.0977823806
X28   0.1544852906
X29  -0.0276528153
X30   0.0164067155
X31  -0.0589237822
X32   0.0205429977
X33   0.1352414127
X34  -0.0301992700
X35  -0.0097282034
X36   0.0631470517
X37   0.1972976603
X38   0.0932627995
```

```
X39   0.1242672243
X40   0.1466160415
X41   0.1113132623
X42  -0.1226767443
X43  -0.0375124313
X44  -0.0155639906
X45  -0.0103119024
X46  -0.1807752324
X47   0.0122969426
X48   0.0309593047
X49   0.0257831520
X50   0.1231058283
X51  -0.0237022950
X52  -0.0136668437
X53   0.0802692451
X54   0.1696197561
X55   0.1711788101
X56  -0.0447894933
X57  -0.0407522017
X58  -0.0768854044
X59   0.0786508030
X60  -0.1192481480
X61  -0.0080695175
X62   0.0701660302
X63   0.0295340139
X64  -0.1090452901
X65   0.0634084702
X66  -0.1451071892
X67   0.1404697803
X68   0.0649124105
X69  -0.1596141322
X70   0.1128300914
X71   0.1889244806
X72   0.0920823098
X73  -0.0647856673
X74  -0.0684613155
X75   0.2307043702
X76  -0.1312463971
X77   0.0301756741
X78  -0.0742119633
X79   0.0695864468
X80  -0.0273958006
X81   0.0183803428
```

```
X82   0.0555629189
X83  -0.0196148832
X84  -0.0119195971
X85   0.0981231232
X86   0.1724823705
X87   0.0832704525
X88  -0.0070402199
X89   0.0720618891
X90   0.0779178976
X91   0.0026821896
X92  -0.1224121573
X93   0.0073619631
X94  -0.0996671370
X95  -0.0486034289
X96   0.0338488127
X97   0.1497352579
X98   0.1702661451
X99   0.0197954507
X100  0.0070025016
```

```r
cat("The function converged after", length(log_reg_adam$obj_values), "
↪  iterations \n")
```

```
The function converged after 279  iterations
```

```r
cat("Eta Vals: \n")
```

```
Eta Vals:
```

```r
print(log_reg_adam$eta_values[1:50])
```

```
 [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[39] 1 1 1 1 1 1 1 1 1 1 1 1
```

```r
cat("Objective Function vals \n")
```

```
Objective Function vals
```

```r
print(log_reg_adam$obj_values[1:50])
```

```
 [1]       Inf       Inf       Inf       Inf 22.2832809 14.8373097
 [7] 7.1284249 2.0996220 6.1137497 1.3895229  2.9819074  3.8056161
[13] 3.8809432 3.3404494 2.3347514 1.1762725  2.9227801  1.5778861
[19] 1.3114477 2.0080097 2.3413146 2.2664773  1.8437349  1.2165596
[25] 1.0948241 1.9858414 0.9086495 1.1820604  1.4516938  1.4476730
[31] 1.1868100 0.8328216 1.1328824 0.9287048  0.7908114  0.9833457
[37] 0.9977538 0.8109798 0.6450300 0.9912377  0.6135863  0.7809745
[43] 0.8118218 0.6527392 0.5829003 0.7306665  0.6084322  0.7789611
[49] 0.7352627 0.5369212
```

(4) Stochastic gradient descent with a fixed schedule of decreasing step sizes

Stochastic gradient descent happens is an implementation of gradient descent that adds randomness by calculating a gradient as a subset of the data points in order to try to get the algorithm to converge

Algorithm (SGD)

1. Select the cardinality s of index set I_k
2. Select $x_0 \in \mathbb{R}^n$
3. While stopping criterion > tol, do:

- $x_{k+1} = x_k - \eta_k \nabla f_{I_k}(x_k)$
- Calculate the value of the stopping criterion

Note that:

$$f_{I_k}(x_k) = \frac{1}{s} \sum_{i \in I_k} f_i(x_k), \quad \nabla [f_{I_k}(x_k)] = \frac{1}{s} \sum_{i \in I_k} \nabla f_i(x_k)$$

Implement SGD

```r
# stochastic gradient descent with fixed schedule of decreasing step size
log_sgd <- function(X, y, tol = 1e-6, max_iter = 10000, s = 32, eta = 1) {
  # Initialize
  n <- nrow(X)
  p <- ncol(X)
  x <- as.matrix(X)
  y <- as.matrix(y)
```

```r
beta <- as.matrix(rep(0, p))
obj_values <- numeric(max_iter)
eta_values <- numeric(max_iter)  # To store eta values used each iteration
beta_values <- list() # To store beta values used each iteration

# Objective function: negative log-likelihood
# input: Beta vector, x matrix, y matrix
# output: scalar objective func value
# comments: We want to minimize this function for logit regression
obj_function <- function(beta, x, y) {
  m <- nrow(x)
  z <- x %*% beta
  (1 / m) * (-(t(y) %*% z) + sum(log(1 + exp(z))))
}

obj_sum <- function(beta, x, y, subset) {
  x_sub <- x[subset, , drop = FALSE]   # subset of x
  y_sub <- y[subset, , drop = FALSE]   # subset of y
  obj_function(beta, x_sub, y_sub)
}

# Gradient function
# input: Beta vector, x matrix, y matrix
# output: gradient vector in the dimension of nrow(Beta) x 1
# comments: We use this for gradient descent
gradient <- function(beta, x, y) {
  m <- nrow(x)                          # define m
  sig <- function(z){
    z <- pmin(z, 20)  # Clip high values
    z <- pmax(z, -20) # Clip l
    1 / (1 + exp(-z))  # sigmoid function
  }
  (1 / m) * (t(x) %*% (sig(x %*% beta) - y))
}

grad_sum <- function(beta, x, y, subset) {
  x_sub <- x[subset, , drop = FALSE]   # subset of x
  y_sub <- y[subset, , drop = FALSE]   # subset of y
  gradient(beta, x_sub, y_sub)
}

# Algorithm:
for (iter in 1:max_iter) {
```

```r
    eta_k = eta / (1 + 0.001 * iter)

    # subset of data
    subset <- sample(1:n, s, replace=FALSE)
    obj_sub <- obj_sum(beta, x, y, subset)
    grad_sub <- grad_sum(beta, x, y, subset)

    beta_new <- beta - eta_k * grad_sub

    # save values to the matrix
    eta_values[iter] <- eta_k
    obj_values[iter] <- obj_sub
    beta_values[[iter]] <- beta_new

    if (sqrt(sum((beta_new - beta)^2)) < tol) {
      # set the vector ranges and break
      beta <- beta_new
      obj_values <- obj_values[1:iter]
      eta_values <- eta_values[1:iter]
      beta_values <- beta_values[1:iter]
      break
    }

    beta <- beta_new
    if (iter == 1 || iter %% 1000 == 0) cat("iter", iter, "eta:", eta_k,
     ↪  "obj:", obj_sub, "\n")
  }

  return(list(beta = beta, obj_values = obj_values, eta_values = eta_values,
   ↪  beta_values = beta_values))
}
```

TESTING: SGD(No ADAM)

```r
log_reg_sgd <- log_sgd(X, y, tol = 1e-4, max_iter = 10000, s = 256, eta =
 ↪  0.001)
```

```
iter 1 eta: 0.000999001 obj: 0.6931472
```

PRINTING OUTPUT

```r
cat("betas \n")
```

```
betas
```

```r
print(log_reg_sgd$beta)
```

```
              y
X1   0.01135434
X2   0.01553247
X3   0.02547594
X4   0.02478308
X5   0.01649211
X6   0.02267313
X7   0.02279703
X8   0.02286618
X9   0.01939711
X10  0.01766395
X11  0.02058742
X12  0.01023013
X13  0.01838424
X14  0.01132511
X15  0.02220197
X16  0.01983699
X17  0.02474106
X18  0.02455106
X19  0.01809302
X20  0.02237795
X21  0.01786076
X22  0.01215984
X23  0.01263780
X24  0.02156132
X25  0.01241035
X26  0.01898106
X27  0.01457513
X28  0.02594599
X29  0.01643030
X30  0.01906561
X31  0.01544249
X32  0.01858988
X33  0.02373124
```

```
X34  0.01690271
X35  0.01768783
X36  0.02072269
X37  0.02700170
X38  0.02219808
X39  0.02400643
X40  0.02591264
X41  0.02360806
X42  0.01282970
X43  0.01740925
X44  0.01747289
X45  0.01663346
X46  0.01006871
X47  0.01872684
X48  0.01928490
X49  0.01917166
X50  0.02379881
X51  0.01596886
X52  0.01697237
X53  0.02194963
X54  0.02668542
X55  0.02570057
X56  0.01660433
X57  0.01652876
X58  0.01447817
X59  0.02157667
X60  0.01216898
X61  0.01935009
X62  0.02070880
X63  0.01855471
X64  0.01285274
X65  0.02124817
X66  0.01023270
X67  0.02506424
X68  0.02185364
X69  0.01003059
X70  0.02329080
X71  0.02609856
X72  0.02219396
X73  0.01473030
X74  0.01507792
X75  0.02978834
X76  0.01263462
```

```
X77  0.01944849
X78  0.01418123
X79  0.02073226
X80  0.01686592
X81  0.01860687
X82  0.01930083
X83  0.01695688
X84  0.01896319
X85  0.02295664
X86  0.02579406
X87  0.02208678
X88  0.01823207
X89  0.02214396
X90  0.02211423
X91  0.01753706
X92  0.01355594
X93  0.01862842
X94  0.01360701
X95  0.01685557
X96  0.02016261
X97  0.02535342
X98  0.02580509
X99  0.01852538
X100 0.01809922
```

```r
cat("The function converged after", length(log_reg_sgd$obj_values), "
↪  iterations \n")
```

```
The function converged after 774  iterations
```

```r
cat("Eta Vals: \n")
```

```
Eta Vals:
```

```r
print(log_reg_sgd$eta_values[1:50])
```

```
 [1] 0.0009990010 0.0009980040 0.0009970090 0.0009960159 0.0009950249
 [6] 0.0009940358 0.0009930487 0.0009920635 0.0009910803 0.0009900990
[11] 0.0009891197 0.0009881423 0.0009871668 0.0009861933 0.0009852217
```

```
[16] 0.0009842520 0.0009832842 0.0009823183 0.0009813543 0.0009803922
[21] 0.0009794319 0.0009784736 0.0009775171 0.0009765625 0.0009756098
[26] 0.0009746589 0.0009737098 0.0009727626 0.0009718173 0.0009708738
[31] 0.0009699321 0.0009689922 0.0009680542 0.0009671180 0.0009661836
[36] 0.0009652510 0.0009643202 0.0009633911 0.0009624639 0.0009615385
[41] 0.0009606148 0.0009596929 0.0009587728 0.0009578544 0.0009569378
[46] 0.0009560229 0.0009551098 0.0009541985 0.0009532888 0.0009523810
```

```r
cat("Objective Function vals \n")
```

```
Objective Function vals
```

```r
print(log_reg_sgd$obj_values[1:50])
```

```
 [1] 0.6931472 0.6899505 0.6883899 0.6831475 0.6787769 0.6762963 0.6759592
 [8] 0.6700340 0.6647162 0.6658464 0.6666402 0.6602883 0.6474681 0.6558720
[15] 0.6591510 0.6485301 0.6458368 0.6404269 0.6463510 0.6531543 0.6455916
[22] 0.6466601 0.6287876 0.6409876 0.6285182 0.6214071 0.6170738 0.6352378
[29] 0.6218606 0.6279371 0.6307798 0.6123240 0.6188174 0.6289519 0.6282660
[36] 0.6089209 0.6031430 0.6105668 0.5921086 0.6086099 0.5997608 0.6108823
[43] 0.6157268 0.6083084 0.6080561 0.6068213 0.5982542 0.5983205 0.5795877
[50] 0.6010424
```

(5) Stochastic gradient descent with AMSGrad-ADAM-W momentum

(no backtracking line search, since AMSGrad-ADAM adjusts step sizes per parameter using momentum and adaptive scaling)

We can apply the AMSGrad-ADAM update to the stochastic gradient algorithm shown previously, except multiplying $(1 - \quad)$ to x_k:

Implement SGD ADAM

```r
# stochastic gradient descent with fixed schedule of decreasing step size
log_sgd_adam <- function(X, y, tol = 1e-6, max_iter = 10000, lambda = 1e-4, s
↪  = 32, eta = 1, epsilon = 1e-8, b_1 = 0.9, b_2 = 0.999) {
  # Initialize
  n <- nrow(X)
  p <- ncol(X)
  x <- as.matrix(X)
```

```r
y <- as.matrix(y)
beta <- as.matrix(rep(0, p))
obj_values <- numeric(max_iter)
eta_values <- numeric(max_iter)  # To store eta values used each iteration
beta_values <- list() # To store beta values used each iteration

# Objective function: negative log-likelihood
# input: Beta vector, x matrix, y matrix
# output: scalar objective func value
# comments: We want to minimize this function for logit regression
 obj_function <- function(beta, x, y) {
  m <- nrow(x)
  z <- x %*% beta
  (1 / m) * (-(t(y) %*% z) + sum(log(1 + exp(z))))
}

obj_sum <- function(beta, x, y, subset) {
  x_sub <- x[subset, , drop = FALSE]   # subset of x
  y_sub <- y[subset, , drop = FALSE]   # subset of y
  obj_function(beta, x_sub, y_sub)
}

# Gradient function
# input: Beta vector, x matrix, y matrix
# output: gradient vector in the dimension of nrow(Beta) x 1
# comments: We use this for gradient descent
gradient <- function(beta, x, y) {
  m <- nrow(x)                          # define m
  sig <- function(z) 1 / (1 + exp(-z))  # sigmoid function
  (1 / m) * (t(x) %*% (sig(x %*% beta) - y))
}

grad_sum <- function(beta, x, y, subset) {
  x_sub <- x[subset, , drop = FALSE]   # subset of x
  y_sub <- y[subset, , drop = FALSE]   # subset of y
  gradient(beta, x_sub, y_sub)
}

# Algorithm:
for (iter in 1:max_iter) {

  # subset of data
  subset <- sample(1:n, s, replace=FALSE)
```

```r
    obj_sub <- obj_sum(beta, x, y, subset)
    grad_sub <- grad_sum(beta, x, y, subset)

    # ADAM step
    if (iter == 1) {
      m_k <- grad_sub
      z_k <- grad_sub^2
      m_hat_k <- m_k / (1 - b_1^iter)
      z_hat_k <- pmax(0, z_k)
      z_tild_k <- 1 / (sqrt(z_hat_k) + epsilon)
    } else {
      m_k <- b_1 * m_k_prev + (1 - b_1) * grad_sub
      z_k <- b_2 * z_k_prev + (1 - b_2) * grad_sub^2
      m_hat_k <- m_k / (1 - b_1^iter)
      z_hat_k <- pmax(z_hat_k_prev, z_k)
      z_tild_k <- 1 / (sqrt(z_hat_k) + epsilon)
    }

    beta_new <- (1 - eta * lambda) * beta - eta * (z_tild_k * m_hat_k)

    # save values to the matrix
    eta_values[iter] <- eta
    obj_values[iter] <- obj_function(beta_new, x, y)
    beta_values[[iter]] <- beta_new

    if (sqrt(sum((beta_new - beta)^2)) < tol) {
      # set the vector ranges and break
      beta <- beta_new
      obj_values <- obj_values[1:iter]
      eta_values <- eta_values[1:iter]
      beta_values <- beta_values[1:iter]
      break
    }

    beta <- beta_new
    z_k_prev <- z_k
    m_k_prev <- m_k
    z_hat_k_prev <- z_hat_k
    if (iter == 1 | iter %% 1000 == 0) cat("iter", iter, "obj:", obj_sub,
     ↪  "\n")
  }

  return(list(beta = beta, obj_values = obj_values, eta_values = eta_values,
   ↪  beta_values = beta_values))
```

```
}
```

TESTING: SGD ADAM

```
log_reg_sgd_adam <- log_sgd_adam(X, y, tol = 1e-2, max_iter = 10000, lambda =
 ↪  1e-4, s = 256, eta = 0.01, epsilon = 1e-8, b_1 = 0.9, b_2 = 0.999)
```

```
iter 1 obj: 0.6931472
```

PRINTING OUTPUT: SGD ADAM

```
cat("betas \n")
```

```
betas
```

```
print(log_reg_sgd_adam$beta)
```

```
                 y
X1   -0.0846350487
X2   -0.0124010853
X3    0.0555628769
X4    0.0742999680
X5   -0.1182542697
X6    0.0558628759
X7    0.0293496739
X8    0.0442484052
X9    0.0077119300
X10  -0.1268388865
X11   0.0614244487
X12  -0.0745620848
X13  -0.0235281602
X14  -0.1118354964
X15  -0.0022481179
X16   0.0362880146
X17   0.0276927003
X18   0.0557341451
X19   0.0739836170
X20  -0.0447301030
```

```
X21   0.0528074479
X22   0.0162599817
X23  -0.0162785676
X24  -0.0954133547
X25  -0.0373110853
X26   0.0209924010
X27  -0.3112053982
X28   0.1055346118
X29   0.0190964421
X30  -0.0111074973
X31  -0.0812639465
X32   0.0674209252
X33   0.0778018010
X34   0.0110840244
X35   0.0474483632
X36   0.0766899453
X37   0.0889207977
X38   0.0569115340
X39   0.0059694992
X40   0.1064577032
X41   0.0848035758
X42  -0.0424669410
X43   0.0257388739
X44  -0.0567801372
X45  -0.0124715608
X46  -0.0189873300
X47  -0.0364654703
X48   0.0326952708
X49   0.0636014370
X50   0.0771965519
X51  -0.0197872936
X52  -0.1040558501
X53   0.0970089486
X54   0.0694368456
X55   0.0137015556
X56  -0.0155630584
X57  -0.0480951127
X58  -0.0892051386
X59   0.0719298726
X60  -0.1548228608
X61   0.0594450153
X62   0.0561337620
X63   0.0139001100
```

```
X64  -0.0551283259
X65  -0.0031146087
X66  -0.0620627266
X67  -0.0134451155
X68   0.0659925789
X69  -0.0828161146
X70   0.0671992452
X71   0.0634629106
X72   0.0064634833
X73  -0.0005192438
X74   0.0131385659
X75   0.0810782316
X76  -0.0357127540
X77   0.0109596326
X78  -0.0167523194
X79   0.0785550751
X80   0.0392163700
X81  -0.0466329074
X82  -0.0254647907
X83   0.0246743763
X84   0.0541367949
X85   0.0715176982
X86   0.0414844249
X87   0.0262101744
X88   0.0024564074
X89   0.0440199118
X90   0.0390924919
X91  -0.0161686477
X92  -0.0050758005
X93  -0.0229872255
X94  -0.0205879448
X95   0.0117065290
X96   0.0255148540
X97   0.0730530351
X98   0.0605628370
X99  -0.0051534570
X100  0.1013927258
```

```
cat("The function converged after", length(log_reg_sgd_adam$obj_values), "
 ↪  iterations \n")
```

```
The function converged after 56  iterations
```

```r
cat("Eta Vals: \n")
```

```
Eta Vals:
```

```r
print(log_reg_sgd_adam$eta_values[1:50])
```

```
 [1] 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01
[16] 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01
[31] 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01
[46] 0.01 0.01 0.01 0.01 0.01
```

```r
cat("Objective Function vals \n")
```

```
Objective Function vals
```

```r
print(log_reg_sgd_adam$obj_values[1:50])
```

```
 [1] 1.1161821 1.5373292 1.7827117 1.9423441 2.0494091 2.1201949 2.1661810
 [8] 2.1935314 2.2075204 2.2104329 2.2073275 2.1962513 2.1773307 2.1519954
[15] 2.1225932 2.0865053 2.0473293 2.0041901 1.9596392 1.9137775 1.8661258
[22] 1.8168482 1.7661386 1.7150601 1.6605808 1.6051586 1.5493324 1.4939364
[29] 1.4369382 1.3794203 1.3210101 1.2625243 1.2055373 1.1491443 1.0926230
[36] 1.0387216 0.9847929 0.9325850 0.8810229 0.8313522 0.7844547 0.7385439
[43] 0.6958187 0.6554445 0.6203532 0.5901666 0.5655170 0.5468710 0.5350448
[50] 0.5304285
```

Part (a) Hyperparameter Discussion

Discuss how you selected the various hyperparameters for each of the algorithms

For BLS, I selected tau and epsilon = 0.5, because they should be between 0 and 1 and 0.5 is relatively standard in order for it to converge. That is fairly standard for the Armijo condition.

For BLS with Nesterov, I kept the hyperparameters the same as BLS because it was standard from before, and then decided to set tau = 0.8 to keep the step size bigger and with faster convergence. The convergence was the same, likely the momentum not making a huge difference with this particular objective function

For SGD, The decreasing step size implemented was eta_k = eta / (1 + 0.001 * iter), ensuring that eta decreases with every iteration, as it is also a common algorithm used in literature to decrease eta. The multiplier 0.001 means that eta won't significantly drop after 1000 iterations, otherwise eta would get too small. I also set eta = 0.001 initially, otherwise it would not converge.

For AMSGRAD-ADAM, I selected Beta1 = 0.9 and Beta2 = 0.999, In order that Beta1 and Beta2 to not be to o small, and it is also a common step size eta = 1 that is used in ADAM. The step size allowed it to converge aggressively with few iterations

For AMSGRAD-ADAM-W with SGD, I selected the same coefficients as AMSGRAD, it's just that I selected lambda to be a very small value ~1e-4, I had to keep the step size and tolerance

Part (b) Metrics

```
g <- glm(y ~ ., data = data, family = binomial())
coefs <- g$coefficients[2:101]
print(sqrt(sum((log_reg_bls$beta - coefs)^2)))
```

```
[1] 0.003482767
```

```
print(sqrt(sum((log_reg_bls_n$beta - coefs)^2)))
```

```
[1] 0.00348093
```

```
print(sqrt(sum((log_reg_adam$beta - coefs)^2)))
```

```
[1] 0.003482475
```

```
print(sqrt(sum((log_reg_sgd$beta - coefs)^2)))
```

```
[1] 0.915068
```

```
print(sqrt(sum((log_reg_sgd_adam$beta - coefs)^2)))
```

```
[1] 0.7681599
```

For the algorithm BLS, BLS_N, AMS_ADAM, SGD, SGD_AMS_ADAM_W

The estimation errors were: [1] 0.003482767, [1] 0.00348093, [1] 0.003482475, [1] 0.915068, [1] 0.6947875

The iterations took 1909, 1909, 275, 769, and 56 respectively

Formatted Table:

Algorithm	Estimation Error	Iterations
BLS	0.003482767	1909
BLS_N	0.00348093	1909
AMS_ADAM	0.003482475	275
SGD	0.915068	769
SGD_AMS_ADAM_W	0.6947875	56

I see that ADAM performed very well in reducing the iterations for converging, and we can see that all the models perform relatively well in terms of estimation error. I would definitely use ADAM if I was going to perform gradient descent in the future. For stochastic gradient descent, it is likely that the tolerance needs to increase, hence the estimation error also needs to increase a lot. However, the benefit is that the function converges in less iterations, the example being stochastic gradient descent with AMSGRAD-ADAM-W