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>, ...
```

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

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

We also have the gradient function:

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

and

$$\nabla_\beta f_i(\beta) = [\sigma(x_i^\intercal \beta) - y_i] \cdot x_i$$

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

$$\begin{aligned}
\nabla f(x) &= \frac{1}{m}\sum_{i=1}^{m} \nabla f_i(x),\ \nabla_\beta f_i(\beta) = [\sigma(x_i^\intercal \beta) - y_i] \cdot x_i \\
\nabla f(\beta) &= \frac{1}{m}\sum_{i=1}^{m} [\sigma(x_i^\intercal \beta) - y_i] \cdot x_i
\end{aligned}$$

And we can also matricize this:

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

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

```
# 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) {
```

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

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

```
betas
```

```r
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
 ↪  = 0.5) {
  # 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
  # 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)
```

```
    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

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

PRINTING OUTPUT

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

```
betas
```

```
print(log_reg_bls_n$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
```

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

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

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

```
Eta Vals:
```

```r
print(log_reg_bls_n$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_n$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
```

(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 <- max(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 <- max(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) {
#   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

```
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

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

```
betas
```

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

```
                 y
X1   -0.1418463622
X2   -0.0601523924
X3    0.1588541573
X4    0.1328428605
X5   -0.0480703025
X6    0.0992737155
X7    0.1190029511
X8    0.1165767113
```

```
X9    0.0121181465
X10   0.0002687163
X11   0.0440520463
X12  -0.1794462460
X13  -0.0107345508
X14  -0.1230689700
X15   0.0724947456
X16   0.0572062077
X17   0.1299656238
X18   0.1249283104
X19  -0.0018098372
X20   0.1249226315
X21  -0.0107941081
X22  -0.1432408159
X23  -0.1095163677
X24   0.0576437775
X25  -0.1190444250
X26   0.0164815558
X27  -0.0977827701
X28   0.1544841023
X29  -0.0276535650
X30   0.0164057779
X31  -0.0589244496
X32   0.0205423298
X33   0.1352412184
X34  -0.0302001111
X35  -0.0097285921
X36   0.0631465984
X37   0.1972980191
X38   0.0932625799
X39   0.1242670193
X40   0.1466159862
X41   0.1113132350
X42  -0.1226774468
X43  -0.0375124689
X44  -0.0155647662
X45  -0.0103122341
X46  -0.1807760542
X47   0.0122965142
X48   0.0309589790
X49   0.0257828924
X50   0.1231055162
X51  -0.0237032815
```

```
X52  -0.0136671230
X53   0.0802685889
X54   0.1696197757
X55   0.1711787988
X56  -0.0447899442
X57  -0.0407526558
X58  -0.0768861287
X59   0.0786501603
X60  -0.1192489709
X61  -0.0080705701
X62   0.0701660587
X63   0.0295336021
X64  -0.1090461166
X65   0.0634079799
X66  -0.1451078658
X67   0.1404694799
X68   0.0649117255
X69  -0.1596152179
X70   0.1128306371
X71   0.1889251386
X72   0.0920819446
X73  -0.0647859932
X74  -0.0684622678
X75   0.2307039691
X76  -0.1312469777
X77   0.0301753385
X78  -0.0742125391
X79   0.0695862174
X80  -0.0273963433
X81   0.0183796051
X82   0.0555625883
X83  -0.0196148306
X84  -0.0119200074
X85   0.0981226620
X86   0.1724823303
X87   0.0832700595
X88  -0.0070410770
X89   0.0720618508
X90   0.0779171504
X91   0.0026816137
X92  -0.1224128603
X93   0.0073612843
X94  -0.0996681796
```

```
X95   -0.0486043333
X96    0.0338486816
X97    0.1497347964
X98    0.1702659904
X99    0.0197950906
X100   0.0070016149
```

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

```
The function converged after 275  iterations
```

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

```
Eta Vals:
```

```
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
```

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

```
Objective Function vals
```

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

```
 [1]        Inf        Inf        Inf        Inf 19.8520058 12.9304874
 [7]  5.8396817  4.3830145  4.7467728  1.7187755  3.3341688  4.2608418
[13]  4.4904099  4.1356096  3.3187136  2.1972655  1.2498733  3.3696220
[19]  2.2058144  1.3250353  2.0043160  2.4767619  2.5952903  2.3798886
[25]  1.8988031  1.3061757  1.2508394  2.1151974  1.1939994  1.1456952
[31]  1.4557426  1.5921738  1.4919325  1.2018727  0.9106533  1.2013993
[37]  1.1157532  0.8317901  0.9954680  1.0783952  0.9797300  0.7667368
[43]  0.7678432  0.8976035  0.6489309  0.7665641  0.7968736  0.6749666
[49]  0.5739384  0.7735166
```

(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

```
# 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, 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

  # 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))))
  }
```

```r
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) {
  if (iter %% 1000 == 0) cat("iter", iter, "eta:", eta_k, "obj:", obj_sub,
   ↪  "\n")
  if(iter > 1) {
    eta_k = eta / iter
  } else {
    eta_k = eta
  }

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

  # 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
  }

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

TESTING: SGD(No ADAM)

```
log_reg_sgd <- log_sgd(X, y, tol = 1e-6, max_iter = 10000, s = 128, eta = 1,
 ↪  b_1 = 0.9, b_2 = 0.999)
```

```
iter 1000 eta: 0.001001001 obj: 0.4833996
iter 2000 eta: 0.0005002501 obj: 0.4805931
iter 3000 eta: 0.0003334445 obj: 0.5491107
iter 4000 eta: 0.0002500625 obj: 0.5893639
iter 5000 eta: 0.00020004 obj: 0.5901737
iter 6000 eta: 0.0001666944 obj: 0.5079305
iter 7000 eta: 0.0001428776 obj: 0.5557577
iter 8000 eta: 0.0001250156 obj: 0.5302283
iter 9000 eta: 0.0001111235 obj: 0.5144824
iter 10000 eta: 0.00010001 obj: 0.4482717
```

PRINTING OUTPUT

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

betas

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

```
                y
X1   -0.0374177304
X2   -0.0552814780
X3    0.0871480285
X4    0.0580243502
X5   -0.0189962609
X6    0.0600327036
X7    0.0583065964
X8    0.0826544426
X9   -0.0130573669
X10  -0.0288722073
X11   0.0442981902
X12  -0.0596011069
X13  -0.0231924668
X14  -0.0177032117
X15   0.0775251237
X16   0.0598423649
X17   0.0928739761
X18   0.0452066563
X19  -0.0135841524
X20   0.0788491230
X21  -0.0033564909
X22  -0.0622182148
X23  -0.0492124789
X24   0.0433869223
X25  -0.0585099124
X26   0.0077968745
X27  -0.0573407763
X28   0.1064990038
X29  -0.0294319184
X30   0.0637885418
X31   0.0465411244
X32   0.0036779891
X33   0.1000613104
X34   0.0057702215
X35   0.0118514858
X36   0.0300477880
X37   0.1262146128
```

```
X38   0.0627882793
X39   0.0782829264
X40   0.0849589547
X41   0.1180254984
X42  -0.0870878801
X43   0.0020454805
X44   0.0005051946
X45  -0.0378314671
X46  -0.0785590551
X47   0.0119187505
X48  -0.0345115973
X49  -0.0047797042
X50   0.0651658439
X51  -0.0198412887
X52   0.0103724821
X53   0.0669682034
X54   0.1025625684
X55   0.0715028738
X56  -0.0212135909
X57  -0.0274997280
X58  -0.0144037167
X59   0.0468246109
X60  -0.0802790289
X61   0.0240443319
X62   0.0435862402
X63   0.0232969485
X64  -0.0350831451
X65   0.0581775678
X66  -0.1065839898
X67   0.1121147286
X68   0.0532103946
X69  -0.0896226804
X70   0.0611916463
X71   0.1201832638
X72   0.0652131401
X73  -0.0147496220
X74  -0.0022954433
X75   0.1160352733
X76  -0.0676031996
X77   0.0333043484
X78  -0.0609328499
X79   0.0708992186
X80   0.0272890246
```

```
X81    0.0080875462
X82    0.0563924362
X83   -0.0142391766
X84    0.0264097326
X85    0.0654634209
X86    0.0970999207
X87    0.0604138632
X88   -0.0266525494
X89    0.0652664682
X90    0.0393979076
X91    0.0184868293
X92   -0.0408868594
X93   -0.0154172422
X94   -0.0778373353
X95   -0.0178457479
X96    0.0297788833
X97    0.0882470995
X98    0.0896540902
X99    0.0287010143
X100   0.0234507700
```

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

```
The function converged after 10000  iterations
```

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

```
Eta Vals:
```

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

```
 [1] 1.00000000 0.50000000 0.33333333 0.25000000 0.20000000 0.16666667
 [7] 0.14285714 0.12500000 0.11111111 0.10000000 0.09090909 0.08333333
[13] 0.07692308 0.07142857 0.06666667 0.06250000 0.05882353 0.05555556
[19] 0.05263158 0.05000000 0.04761905 0.04545455 0.04347826 0.04166667
[25] 0.04000000 0.03846154 0.03703704 0.03571429 0.03448276 0.03333333
[31] 0.03225806 0.03125000 0.03030303 0.02941176 0.02857143 0.02777778
[37] 0.02702703 0.02631579 0.02564103 0.02500000 0.02439024 0.02380952
[43] 0.02325581 0.02272727 0.02222222 0.02173913 0.02127660 0.02083333
[49] 0.02040816 0.02000000
```

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

```
Objective Function vals
```

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

```
 [1] 0.6931472 3.3224752 1.1757203 0.8951751 0.7049514 1.1566889 0.7740529
 [8] 0.8481838 0.5961463 0.4365674 0.4961555 0.5629517 0.6110768 0.6097521
[15] 0.5525478 0.5041885 0.4843160 0.4767519 0.4881898 0.4940512 0.5001145
[22] 0.5452203 0.5080727 0.5241516 0.4911822 0.5035805 0.5921574 0.4454085
[29] 0.6098483 0.5116747 0.4614430 0.5215265 0.6431649 0.5169591 0.5203553
[36] 0.4789210 0.4938642 0.4818634 0.5021117 0.5493111 0.4736630 0.6490636
[43] 0.5836519 0.5319409 0.5918284 0.4718493 0.4914902 0.5303020 0.5701241
[50] 0.4897690
```

(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)
  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
```

```r
# 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) {
  if (iter %% 1000 == 0) cat("iter", "obj:", obj_sub, "\n")

  # 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)

  # ADAM step
  if (iter == 1) {
    m_k <- (1 - b_1) * grad_sub
    z_k <- (1 - b_2) * grad_sub^2
```

```
      m_hat_k <- m_k / (1 - b_1^iter)
      z_hat_k <- max(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 <- max(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
  }

  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-6, max_iter = 10000, lambda =
 ↪  1e-4, s = 32, eta = 1, epsilon = 1e-8, b_1 = 0.9, b_2 = 0.999)
```

```
iter obj: 12.06407
iter obj: 2.531643
iter obj: 5.737204
iter obj: 3.60273
iter obj: 1.100128
iter obj: 2.217027
iter obj: 5.417208
iter obj: 4.39512
iter obj: 2.842576
iter obj: 2.220743
```

PRINTING OUTPUT: SGD ADAM

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

```
betas
```

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

```
              y
X1   -3.26835781
X2   -0.44347203
X3    3.26305399
X4    0.94087725
X5   -1.18747337
X6    1.48535302
X7    1.84180957
X8    1.77463298
X9    0.41673067
X10  -1.19771275
X11   1.21756565
X12  -1.41940829
X13   0.52268082
X14  -1.97801057
X15  -0.95729656
X16   2.02827330
X17   0.14700776
X18   0.97803255
X19  -0.84171785
X20   0.67821527
```

```
X21   0.09830971
X22  -4.00163941
X23  -1.17336298
X24  -0.76923237
X25  -0.62323199
X26  -0.18242502
X27  -3.92574861
X28   0.79151178
X29   0.76415956
X30   0.29982800
X31   1.04701580
X32  -1.11262368
X33   2.41667412
X34   1.63027579
X35   0.32333089
X36  -0.19400756
X37   1.36746172
X38   3.79827704
X39   0.67324011
X40   2.06559579
X41   0.92588667
X42  -0.95686149
X43  -0.83831018
X44   0.85262437
X45   0.79173351
X46  -2.37482144
X47   0.24220381
X48   2.55728469
X49  -2.54975580
X50   0.26767449
X51  -0.39598156
X52  -1.71712350
X53  -0.75483463
X54   0.54904300
X55   0.43459976
X56  -1.44002390
X57   0.62120236
X58   0.31331999
X59   0.75910558
X60  -2.79707509
X61   0.05223149
X62  -0.39382081
X63  -1.00469254
```

```
X64   0.18428308
X65   1.03402005
X66  -0.39573095
X67  -0.59921503
X68   0.03854199
X69  -0.13969101
X70   2.57773168
X71   1.05802705
X72  -0.36663329
X73  -0.52748695
X74  -0.47003475
X75   1.22959112
X76   0.37289865
X77   1.06049289
X78  -1.53892119
X79   1.11644354
X80   0.07809200
X81   0.87913100
X82  -0.43623013
X83   0.24923502
X84   0.12628232
X85  -1.68112051
X86   1.22133905
X87   0.45633608
X88  -2.59868351
X89  -0.26967113
X90  -0.83240432
X91  -1.55681440
X92  -2.65887137
X93   0.43522903
X94   0.10414063
X95   1.04254112
X96  -1.35318743
X97   0.31668937
X98   0.60360257
X99  -1.69507364
X100  1.63596965
```

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

```
The function converged after 10000  iterations
```

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

```
Eta Vals:
```

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

```
 [1]      Inf      Inf      Inf      Inf      Inf 28.34960      Inf      Inf
 [9] 31.88056      Inf      Inf      Inf      Inf      Inf      Inf      Inf
[17]      Inf      Inf      Inf 29.73365 55.06724 49.79323 30.45774 28.42583
[25]      Inf      Inf      Inf      Inf      Inf      Inf      Inf      Inf
[33]      Inf 28.07532 24.99386 25.21001 30.87866 30.00103 24.65384 22.25406
[41] 23.06119 23.73060 22.43201 21.15671 20.21419 19.86670 18.91779 18.97949
[49] 22.15657 21.18867
```

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

For BLS with Nesterov, I kept the hyperparameters the same as BLS because it was standard from before

For gradient descent with backtracking, I selected epsilon = 0.5 because that is fairly standard in the Armijo condition

For SGD, The decreasing step size implemented was eta / number of iterations, ensuring that eta decreases with every iteration, as it is also a common algorithm used in literature to decrease eta

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 that is used in

For AMSGRAD-ADAM-W, I selected the same coefficients as AMSGRAD, it's just that I selected lambda to be a very small value ~1e-4

Part (b) Metrics

```
g <- glm(y ~ ., data = data, family = binomial())
```

For the algorithm BLS, BLS_N, ADAM, SGD, SGD_ADAM_W

The iterations took 1909, 1909, 275, 10000, and 10000 respectively