DSA3102 Homework 2

Douglas Wei Jing Allwood (A0183939L)

2 April 2020

Contents


```
library(rmatio)

email_full <- read.mat("HW1data.mat")

X <- email_full$Xtrain
y <- email_full$ytrain

X_test <- email_full$Xtest
y_test <- email_full$ytest
```

Helper Functions

```
# Function to compute the Euclidean Norm of a given vector x
vnorm <- function(x) {
  return(sqrt(c(x %*% x)))
}

# Objective Function: Negative Log Likelihood
negative_log_likelihood <- function(w, X, y) {
  return(sum(log(1 + exp(-y * (X %*% w)))))
}

# Gradient Function
log_likelihood_gradient <- function(w, X, y) {
  coefs <- -y / (1 + exp(y * as.vector(X %*% w)))
  mat <- coefs * X
  return(apply(mat, 2, sum))
}
```

Question 4

Expressions

Expression for $\ell(w)$ and the gradient vector $\nabla\ell(w)$:

$$\ell(w) = \sum_{i=1}^{n} \log(1 + e^{-y_i w^T X_i})$$

$$\nabla\ell(w) = \sum_{i=1}^{n} \frac{-y_i X_i}{1 + e^{y_i w^T X_i}}$$

Descent Algorithms

```
armijo <- function(w, grad, fn, miu, beta, X, y) {
  # Armijo search: Find step size alpha
  alpha = 1
  repeat {
    f_next <- fn(w + alpha * grad, X, y)
    f_curr <- fn(w, X, y)

    # d = -gradientVector
    # <gradientVector,d> == -1 * ||gradientVector||
    inner_prod <- -1 * alpha * miu * (grad %*% grad)
    if (f_next - f_curr <= inner_prod) {
      break
    }
    alpha <- alpha * beta
  }
  return(alpha)
}

gradient_descent <- function(fn, gradient_fn, X, y, initial_soln = NA,
                             tolerance = 0.01, max_iter = 500, miu = 0.25,
                             beta = 0.5, batch_size = 10, steps = 50) {
  if (any(is.na(initial_soln))) {
    # If no initial stating point is provided, use the column mean from X
    initial_soln <- apply(X, 2, mean)
  }
  w <- initial_soln

  step_count <- 1
  repeat {
    grad <- -1 * gradient_fn(w, X, y)

    if (vnorm(grad) < tolerance) {
      break
    }

    # Update the current solution vector according to best step size
    alpha <- armijo(w, grad, fn, miu, beta, X, y)
    w <- w + alpha * grad

    # Terminate if max_iter reached
```

```
    if (step_count >= max_iter) {
      break
    }
    step_count <- step_count + 1

    ##### Print intermediate progress
    if (step_count %% steps == 0) {
      cat("Step: ", step_count, "\t", "Gradient Norm: ", vnorm(grad), "\t",
          "Function Value: ", fn(w, X, y), "\n")
    }
  }
  cat("Armijo Gradient Descent algorithm terminated after step: ", step_count, "\t",
  "The norm of the gradient at the solution vector is: ",
  vnorm(grad),"\t", "The function Value at this point is: ",
  fn(w, X, y), "\n")
  return(w)
}

stochastic_gradient_descent <- function(fn, gradient_fn, X, y,
                                        initial_soln = NA, tolerance = 0.01,
                                        max_iter = 5e6, batch_size = 10,
                                        steps = 1e4) {
  if (any(is.na(initial_soln))) {
    # If no initial stating point is provided, use the column mean from X
    initial_soln <- apply(X, 2, mean)
  }
  w <- initial_soln

  step_count <- 1
  repeat {
    # Perform Batch Stochastic Gradient Descent
    rand <- sample.int(nrow(X), batch_size)
    X_rand <- X[rand,]
    y_rand <- y[rand]
    grad <- -1 * gradient_fn(w, X_rand, y_rand)

    ##### Step Strategy code: 3 strategies
    # 1. Armijo over sampled data
    # alpha <- armijo(w, grad, fn, miu, beta, X_rand, y_rand)

    # 2. Decreasing step size
    alpha <- 1 / (step_count)

    # 3. Fixed step size
    # alpha <- 0.01

    # Update current solution vector according to best step size
    w <- w + alpha * grad

    ##### Termination Code
    # Terminate if norm(z) = norm(alpha * grad) < tolerance
    # Explanation: (w_prev + alpha * grad) - w_prev = alpha * grad
    if (vnorm(alpha * grad) < tolerance) {
```

```
      break
    }

    # Terminate if max_iter reached
    if (step_count >= max_iter) {
      break
    } else if (step_count %% steps == 0) {
      # Check the full gradient once for termination as well
      full_grad <- gradient_fn(w, X, y)
      if (vnorm(full_grad) < tolerance) {
        break
      }
    }
    step_count <- step_count + 1

    ##### Print intermediate progress
    if (step_count %% steps == 0) {
      cat("Step: ", step_count, "\t", "Gradient Norm: ", vnorm(full_grad), "\t",
          "Function Value: ", fn(w, X, y), "\n")
    }
  }
  cat("Stochastic Descent algorithm terminated after step: ", step_count, "\t",
  "The norm of the gradient at the solution vector is: ",
  vnorm(gradient_fn(w, X, y)),"\t", "The function Value at this point is: ",
  fn(w, X, y), "\n")
  return(w)
}
```

```
#initial_solution <- gradient_descent(negative_log_likelihood,
#                                     log_likelihood_gradient, X, y,
#                                     max_iter = 1e10, tolerance = 1,
#                                     steps = 100)
#
#w1 <- gradient_descent(negative_log_likelihood, log_likelihood_gradient, X, y,
#                       max_iter = 1e10, tolerance = 0.1, steps = 100,
#                       initial_soln = initial_solution)
#
#w2 <- gradient_descent(negative_log_likelihood, log_likelihood_gradient, X, y,
#                       max_iter = 1e10, tolerance = 0.01, steps = 100,
#                       initial_soln = w1)
#
#w3 <- gradient_descent(negative_log_likelihood, log_likelihood_gradient, X, y,
#                       max_iter = 1e10, tolerance = 0.001, steps = 100,
#                       initial_soln = w2)

# w1 Steps required: 6610
# w2 Steps required: 19107
# w3 Steps required: 30968
```

```
#w1_sgd <- stochastic_gradient_descent(negative_log_likelihood,
#                                      log_likelihood_gradient, X, y,
#                                      max_iter = 1e20, steps = 5e2,
#                                      tolerance = 0.1, batch_size = 20,
#                                      initial_soln = initial_solution)
```

```
#
#w2_sgd <- stochastic_gradient_descent(negative_log_likelihood,
#                                      log_likelihood_gradient, X, y,
#                                      max_iter = 1e20, steps = 5e2,
#                                      tolerance = 0.01, batch_size = 20,
#                                      initial_soln = initial_solution)
#
#w3_sgd <- stochastic_gradient_descent(negative_log_likelihood,
#                                      log_likelihood_gradient, X, y,
#                                      max_iter = 1e20, steps = 5e2,
#                                      tolerance = 0.001, batch_size = 20,
#                                      initial_soln = initial_solution)

# w1_sgd Steps required: 37
# w2_sgd Steps required: 32
# w3_sgd Steps required: 16
```

Comparison

This Rmd used two algorithms Gradient Descent with Armijo (GD) and Stochastic gradient descent.

Numbers of iterations required

Gradient Descent (with Armijo)

1. 6610 to achieve a tolerance of 0.1
2. 19107 to achieve a tolerance of 0.01
3. 30968 to achieve a tolerance of 0.001

Stochastic Gradient Descent (with decreasing step sizes)

1. 37 to achieve a tolerance of 0.1
2. 32 to achieve a tolerance of 0.01
3. 16 to achieve a tolerance of 0.001

A possible explanation for the extremely small number of step sizes required by SGD is that the terminating criteria that norm(alpha * grad) < tolerance is too easy to fulfil. This could be because many of the datapoints in the dataset are close to their optimal point and hence the random gradient vector computed on a sample of points is quite small. This means that vnorm(alpha * grad) has a high probability of being less than the required tolerance, hence terminating within just a few iterations of SGD.

A way to combat this could be to normalise the data points by their variance, so that they are more reasonably spaced. Another alternative is to set a minimum number of iterations required or simply sample larger batch sizes.

Speed

Each iteration of GD ran more slowly than SGD. However, GD is able to additionally improve a previously found optimal solution. For example, if the GD algorithm was first run with a tolerance of 0.1 to find the optimal solution. Then if we want to find an optimal solution with a tolerance of 0.01 we could simply supply the previously found optimal solution as the new initial solution and count the additional number of iterations required. On the otherhand, this might not work for SGD when only using a small number of steps as SGD is random and could diverge from the optimal point when the number of iterations it performs are small.

Accuracy

The norm of the resulting gradient vectors at the optimal solutions that were found by each of GD and SGD tended to be smaller (better) via GD. This is because GD uses this exact metric as its terminating condition, requiring that norm(grad_vector) < tolerance. On the other hand, SGD only requires that norm(x_{k+1} - x{k}) < tolerance. This means that while SGD might terminate, its true Gradient Norm at the solution it found might be larger than the tolerance provided.
To get a similar accuracy with SGD as compared to GD, we should use a much smaller tolerance value (e.g. tolerance_gd / number of data points)