Alternative Direction Method of Multipliers (ADMM)

Implement the ADMM for fitting the lasso regression model, and compare it to the Proximal Gradient method.

Proximal Gradient Method

```r
# Exercises 6 functions
#_________________________________________________________________
# Estimate the Lasso Objective Function

Lasso_OF <- function(beta, Y, X, lambda){
  # Compute the Negative Log-Likelihood function
  # Input:
  # beta = regression parameters (vector M)
  # X = features (matrix M x P)
  # Y = vector of observations (vector M)
  # lambda = complexity parameter (scalar)
  # Output:
  # Lasso_OF = Lasso Objective Function (scalar)
  t <- X %*% beta
  OF <- (0.5 ) * crossprod(Y - t) + lambda * sum(abs(beta))
  OF <- as.numeric(OF)
}

#_________________________________________________________________
# Estimate the Gradient of the Negative Log-Likelihood

Gradient <- function(beta, Y, X){
  # Estimate the gradient of  the Negative Log-Likelihood of the Logit Model
  # Input:
  # beta = regression parameters (vector M)
  # X = features (matrix M x P)
  # Y = vector of observations (vector M)
  # Output:
  # gradient = gradient (vector M)
  t <- Y - X %*% beta
  grad <- -(1/nrow(X)) * crossprod(X, t)
}

#_________________________________________________________________
# Estimate the Soft-Thresholding Function
S_lambda <- function(Y, lambda){
  # Estimate beta using the soft-thresholding operator
  # Input:
  # Y = vector of observations (vector M)
  # lambda = complexity parameter (constant)
  # Output:
  # beta = estimated beta vector (vector M)
  t <- cbind(rep(0, length(Y)), abs(Y) - lambda)
  prox <- sign(Y) * apply(t, 1, max)
}
```

```r
#___________________________________________________________________
# The Proximal Gradient Method Implementation

proximal_gradient <- function(Y, X, lambda, gamma, iter, tol){
  # Estimate Lasso regression using The Proximal Gradient Method
  # Input:
  # Y = vector of observations (vector M)
  # X = features (matrix M x P)
  # lambda = complexity parameter (constant)
  # gamma = proximal operator parameter (scalar)
  # iter = number of iterations (scalar = N)
  # tol = tolerance (scalar)
  # Output:
  # beta = estimated beta vector (vector P)
  # Lasso_OF = negative log-likelihood per iteration (vector N)

  P <- dim(X)[2]
  betas <- array(NA, dim=c(iter, P))
  betas[1,] <- rep(0, P) # Initial guess is zero
  OF <- array(NA, dim = iter)
  OF[1] <- Lasso_OF(betas[1,], Y, X, lambda)

  for (i in 2:iter){
    gradient <- Gradient(betas[i-1,], Y, X) # Step 1 of my pseudocode
    u <- betas[i-1,] - gamma * gradient # Step 2 of my pseudocode
    betas[i,] <- S_lambda(u, gamma * lambda)  # Step 3 of my pseudocode
    OF[i] <- Lasso_OF(betas[i,], Y, X, lambda)

    # Convergence Check
    e <- abs(OF[i-1] - OF[i]) / (OF[i-1] + tol)
    if ( e < tol){
      cat('Converged at_', i)
      OF <- OF[1:i]
      betas <- betas[1:i, ]
      break
    }
    else if ((i == iter) & (e) >= tol){
      print('Did not Converge')
      break
    }
  }
  return(list("Lasso_OF" = OF, "betas" = betas[i,]))
}

#___________________________________________________________________
# The Accelerated Proximal Gradient Method Implementation

ac_proximal_gradient <- function(Y, X, lambda, gamma, iter, tol){
  # Estimate Lasso regression using The Accelerated Proximal Gradient Method
  # Input:
  # Y = vector of observations (vector M)
  # X = features (matrix M x P)
  # lambda = complexity parameter (constant)
```

```
  # gamma = proximal operator parameter (scalar)
  # iter = number of iterations (scalar = N)
  # tol = tolerance (scalar)
  # Output:
  # beta = estimated beta vector (vector P)
  # Lasso_OF = negative log-likelihood per iteration (vector N)

  P <- dim(X)[2]
  betas <- array(NA, dim=c(iter, P))
  betas[1,] <- rep(0, P) # Initial guess is zero
  OF <- array(NA, dim = iter)
  OF[1] <- Lasso_OF(betas[1,], Y, X, lambda)
  s <- rep(NA, iter)
  s[1] <- 1
  z <- rep(0, P)

  for (i in 2:iter){
    gradient <- Gradient(betas[i-1,], Y, X) # Step 1 of my pseudocode
    u <- z - gamma * gradient # Step 2 of my pseudocode
    betas[i,] <- S_lambda(u, gamma * lambda)  # Step 3 of my pseudocode
    s[i] <- (1 + sqrt(1 + 4 * s[i-1]^2))/2 # Step 4 of my pseudocode
    # Step 5 of my pseudocode
    z <- betas[i, ] + ((s[i-1] - 1)/s[i]) * (betas[i, ] - betas[i-1, ])
    OF[i] <- Lasso_OF(betas[i,], Y, X, lambda)

    # Convergence Check
    e <- abs(OF[i-1] - OF[i]) / (OF[i-1] + tol)
    if ( e < tol){
      cat('Converged at_', i)
      OF <- OF[1:i]
      betas <- betas[1:i, ]
      break
    }
    else if ((i == iter) & (e) >= tol){
      print('Did not Converge')
      break
    }
  }
  return(list("Lasso_OF" = OF, "betas" = betas[i,]))
}

#__________________________________________________________________
```

ADMM Method

```
#__________________________________________________________________
# The Alternative Direction Method of Multipliers (ADMM) Method Implementation

ADMM <- function(Y, X, lambda, rho, iter, tol){
  # Estimate Lasso regression using The ADMM Method
  # Input:
  # Y = vector of observations (vector M)
```

```
  # X = features (matrix M x P)
  # lambda = complexity parameter (constant)
  # rho = step size (scalar)
  # iter = number of iterations (scalar = N)
  # tol = tolerance (scalar)
  # Output:
  # beta = estimated beta vector (vector P)
  # Lasso_OF = negative log-likelihood per iteration (vector N)

  P <- dim(X)[2]
  betas <- array(NA, dim=c(iter, P))
  betas[1,] <- rep(0, P) # Initial guess is zero

  OF <- array(NA, dim = iter)
  OF[1] <- Lasso_OF(betas[1,], Y, X, lambda)

  inv <- solve(crossprod(X) + diag(rho, P))
  XtY <- crossprod(X, Y)

  z <- rep(0, P) # gamma
    v <- rep(0, P)

  for (i in 2:iter){
    betas[i,] <- inv %*% (XtY + rho * ( z - v)) # Step 1 of my pseudocode
    z <- S_lambda(betas[i,] + v, lambda / rho) # Step 2 of my pseudocode
    v <- v + betas[i,] - z # Step 3 of my pseudocode

    OF[i] <- Lasso_OF(betas[i,], Y, X, lambda)

    # Convergence Check
    e <- abs(OF[i-1] - OF[i]) / (OF[i-1] + tol)
    if ( e < tol){
      cat('Converged at_', i)
      OF <- OF[1:i]
      betas <- betas[1:i, ]
      break
    }
    else if ((i == iter) & (e) >= tol){
      print('Did not Converge')
      break
    }
  }
  return(list("Lasso_OF" = OF, "betas" = betas[i,]))
}
#__________________________________________________________________
```

Results using Diabetes Data

```
#__________________________________________________________________
# Load Diabetes Data
X = as.matrix(read.csv('diabetesX.csv'))
X = scale(X)
```

```r
Y = as.numeric(read.csv('diabetesY.csv',header=F)[,1])
Y = scale(Y)

#_________________________________________________________________
# Set Values
lambda <- 0.01
gamma <- 0.01
iter <- 10000
tol <- 1E-10
rho <- 5

#_________________________________________________________________
```

```r
#_________________________________________________________________
# Apply Functions
Prox_Grad <- proximal_gradient(Y, X, lambda, gamma, iter, tol)
```

```
## Converged at_ 8614
```

```r
Ac_Prox_Grad <- ac_proximal_gradient(Y, X, lambda, gamma, iter, tol)
```

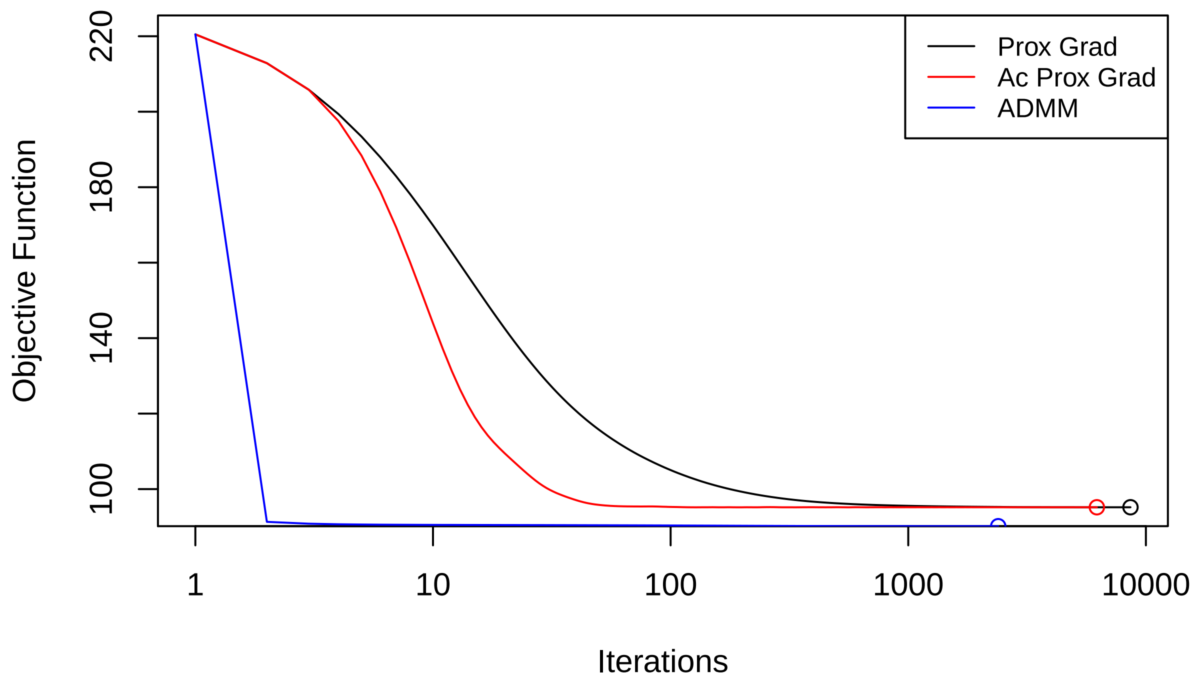
```
## Converged at_ 6218
```

```r
ADMM <- ADMM(Y, X, lambda, rho, iter, tol)
```

```
## Converged at_ 2391
```

```r
#_________________________________________________________________
# Plot Results

#png(filename=paste('Convergency','.png', sep = ""), width = 15,
#    height = 12, units = "cm", res = 200)
plot(Prox_Grad$Lasso_OF, type = 'l', log = 'x', xlab = 'Iterations',
     ylab = 'Objective Function')
lines(Ac_Prox_Grad$Lasso_OF, col="red")
lines(ADMM$Lasso_OF, col="blue")
points(8614, Prox_Grad$Lasso_OF[4807])
points(6218, Ac_Prox_Grad$Lasso_OF[1375], col="red")
points(2391, ADMM$Lasso_OF[1375], col="blue")
legend('topright',legend = c('Prox Grad','Ac Prox Grad','ADMM'),
       col=c("black", "red","blue"), lty=1:1, cex=0.8)
```


Prox Grad
Ac Prox Grad
ADMM
Objective Function
100
140
180
220
1
10
100
1000
10000
Iterations

```r
#dev.off()

#_________________________________________________________________
```