Exercise6\_FINAL\_TURNEDIN.R

jrdha

Sun Nov 04 13:41:50 2018

## AUTHOR: Jared Hansen  
## LAST MODIFIED: Sunday, 11/04/2018  
  
  
  
  
#============================================================  
#==== CODE FOR STATISTICAL LEARNING II, PROBLEM 6 ===========  
#============================================================  
  
  
# Necessary libraries for the script  
library(dplyr) # For data manipulation  
library(R.matlab) # For needed Matlab functionalities  
library(geometry) # For needed Matlab functionalities  
library(pracma) # For needed Matlab functionalities  
  
  
# Change the working directory so that the image file can be read in.  
# The line below needs to be changed if you want to run the code.  
setwd("C:/Users/jrdha/OneDrive/Desktop/USU\_Fa2018/Moon\_\_SLDM2/hw3")  
  
  
# Read in the MNIST data as a dataframe with response "Y" and predictors "X\_1", "X\_2",...  
df <- R.matlab::readMat("mnist\_49\_3000.mat") %>% lapply(t) %>% lapply(as\_tibble)  
colnames(df[[1]]) <- sprintf("X\_%s",seq(1:ncol(df[[1]])))  
colnames(df[[2]]) <- c("Y")  
df <- bind\_cols(df) %>% select(Y, everything())  
  
  
# This function calculates and returns the value of the objective function J(theta), where  
# J(theta) = -l(theta) + lambda||theta||^2, the regularized logistic regression obj. function  
# Arguments passed in are the feature vectors s (xs), response (y),  
# theta (vector of parameters, constant b and weights), and lambda (scalar constant).  
calcObjFctn <- function(xs, y, theta, lambda){  
 indivTerms <- double(nrow(xs))  
 for(i in 1:nrow(xs)){  
 xi <- as.numeric(xs[i,])  
 yi <- as.numeric(y$Y[i])  
 t1 <- yi\*log(1/(1 + exp(dot(-theta, xi))))  
 t2 <- (1-yi)\*log((exp(dot(-theta, xi)))/(1 + exp(dot(-theta, xi))))  
 indivTerms[i] <- t1 + t2  
 }  
 objFctnMin <- (-1 \* sum(indivTerms)) + (lambda \* dot(theta, theta))  
 return(objFctnMin)  
}  
  
  
# Used the gradient and Hessian calculated in part 4, but a resource from Carnegie Mellon  
# gave a more concise, clean way of doing it. This involves calculating a value 'mu' for each  
# feature vector x\_i. As such, used the Carnegie Mellon formulation.  
# The arguments passed are the vector theta, and the x\_i.  
calcMuVec <- function(theta, x\_i){  
 # calculates mu as defined in   
 # Args:  
 # theta: vector containing intercept (b) and weights (w)  
 # x\_i: vector of predictor variables (one observation ("row"))  
 #   
 # theta and x\_i must be vectors of same length  
 #  
 # Returns:  
 # The optimized value of theta  
 denom <- (1 + exp(dot(-theta, x\_i)))  
 return(1/denom)  
}  
  
  
# As is mentioned in the function above, I'm using the gradient and Hessian of the objective  
# function as defined in a resource from Carnegie Mellon. Here is the link:  
# http://www.cs.cmu.edu/~mgormley/courses/10701-f16/slides/lecture5.pdf  
# The function below calculates both the gradient and the Hessian of the regularized logistic  
# regression objective function. It relies on the function that calculates the mu's from   
# above. The return is the gradient and Hessian in a list.  
# Also, the calculation is tuned by adjusting the value of the constant lambda as seen within this fctn.  
# Arguments: xMatrix contains the predictor variables (with a leading column of 1s so that dims match  
# because the first element of the theta vector is the constant b),  
# yVector contains the values of response (-1 or 1)  
# theta is still a vector containing [b, weight1, weight2, ....]^T  
# Original formulas for this can be seen in the commented out code near the bottom of the file.  
calcCMU\_grad\_Hess <- function(xMatrix, yVector, theta){  
   
 # TUNE LAMBDA HERE  
 lambda <- 1  
 muVec <- double(nrow(xMatrix))  
   
 for(i in 1:nrow(xMatrix)){  
 x\_i <- as.numeric(xMatrix[i,])  
 muVec[i] <- calcMuVec(theta, x\_i)  
 }  
 # This formulation of gradient is the CMU definition using the mu's in the calculation.  
 gradient <- t(xMatrix) %\*% (muVec - as.numeric(yVector$Y)) + 2\*lambda\*theta  
 ds <- muVec \* (1 - muVec)  
 D <- diag(ds)  
 XT\_matrix <- t(sapply(xMatrix, as.numeric))  
 X\_hess\_matrix <- sapply(xMatrix, as.numeric)  
 twoLambIdMatrix <- diag(rep(2 \* lambda, 785))  
 hessian <- XT\_matrix %\*% D %\*% X\_hess\_matrix + twoLambIdMatrix  
   
 results <- list("gradient" = gradient, "hessian" = hessian)  
 return(results)  
}  
  
  
# This function gives the initial guess for theta vector: [b=1, w1=0, w2=0,....,w784=0]^T.  
# I used this because it sounds like that's what built-in functions in R do, and trial and error  
# suggested that it was working.  
# Takes as arguments maxIterations (self-explanatory), predictors (matrix of x\_i), response (vec of y\_i)  
# Calls the newtonsMethod function (recursive function).  
initialTheta <- function(maxIterations, predictors, response){  
  
 b\_0 = 1 # initial guess of b  
 w\_0\_Vec = rep(0, 784) # initial guess of w's  
 theta\_0 <- c(b\_0, w\_0\_Vec)  
  
 theta <- newtonsMethod(theta = theta\_0, current\_Iter = 0, maxIterations = maxIterations, pred\_X = predictors, y = response)  
 return(theta)  
}  
  
  
# This recursive function either returns the value of theta (after maxIterations) or calls itself again  
# if maxIterations hasn't yet been reached. Calls the calcCMU\_grad\_Hess function to calculate the  
# gradient and the hessian  
# Takes are arguments: theta (vector of b and weights, gets called in the initialTheta fctn),  
# maxIterations (self-explanatory), current\_Iter (current iteration in the fctn),  
# pred\_X (x predictors dataframe), y (y resp values dataframe)   
newtonsMethod <- function(theta, current\_Iter, maxIterations, pred\_X, y){  
  
 if(current\_Iter >= maxIterations){  
 return(theta)  
 } else {  
 print(paste("Iteration Number: ", current\_Iter))  
   
 gh <- calcCMU\_grad\_Hess(xMatrix = pred\_X, yVector = y, theta = theta)  
 gradient <- gh$gradient  
 hessian <- gh$hessian  
   
 # gradient <- gradientJ(xMatrix = pred\_X, yVector = y, theta = theta)  
 # hessian <- hessianJ(pred\_X = pred\_X, theta = theta)  
  
 current\_theta <- theta - (inv(hessian) %\*% gradient)  
 current\_Iter <- current\_Iter + 1  
 return(newtonsMethod(theta = current\_theta, current\_Iter = current\_Iter, maxIterations = maxIterations, pred\_X = pred\_X, y = y))  
 }  
}  
  
  
# This function implements regularized logistic regression to predict the class (-1, 1) of each obs.  
# It returns a dataframe that contains info on classification (correct, incorrect) as correctIndVar,  
# the probability calculated for each x\_i. Also calculates the correct classification rate (PCC).  
# Returns all of these things together as a list.  
# Takes as arguments: x\_df\_no1s (dataframe of predictors without leading column of 1s),   
# y\_df (dataframe of response values, -1 or 1),  
# theta (self-explanatory)  
logRegr\_Pred <- function(x\_df\_no1s, y\_df, theta){  
  
 w <- theta[2:length(theta)]  
 b <- theta[1]  
 probabilities <- double(nrow(x\_df\_no1s))  
 predictedClass <- double(nrow(x\_df\_no1s))  
 correctIndVar <- double(nrow(x\_df\_no1s))  
   
 for(i in 1:nrow(x\_df\_no1s)){  
 xi <- as.numeric(x\_df\_no1s[i,])  
 prob\_i <- 1 / (1 + exp(-dot(w, xi) + b))  
 probabilities[i] <- prob\_i  
 if(prob\_i > 0.5){  
 predictedClass[i] <- 1  
 } else{  
 predictedClass[i] <- -1  
 }  
   
 if(as.numeric(predictedClass[i]) == as.numeric(y\_df$Y[i])){  
 correctIndVar[i] = 1  
 } else {  
 correctIndVar[i] = 0  
 }  
 }  
 result <- data.frame(probabilities, predictedClass, Y = y\_df$Y, correctIndVar)  
 PCC <- sum(correctIndVar)/nrow(x\_df\_no1s)  
 list\_Results <- list("results" = result, "PCC" = PCC)  
   
 return(list\_Results)  
}  
  
  
# As instructed in the prompt, split the data into a training set (first 2000 obs).  
# Add a column of leading 1s for correct dimensions/calculations when dealing with b in theta.  
trainData <- dplyr::slice(df, 1:2000)  
trainPredictors <- select(trainData, -Y)  
X\_0 <- rep(1, 2000)  
trainPredictors <- cbind(X\_0, trainPredictors)  
trainResponse <- select(trainData, Y)  
  
# As instructed in the prompt, split the data into a test set too (last 1000 obs).  
# Add a column of leading 1s for correct dimensions/calculations when dealing with b in theta.  
testData <- dplyr::slice(df, 2001:3000)  
testPredictors <- select(testData, -Y)  
X\_0 <- rep(1000)  
testPredictors <- cbind(X\_0, testPredictors)  
testResponse <- select(testData, Y)  
  
  
# Call the initialTheta fctn to get the optimized theta vector from the 2000-obs training data.  
optmzThetaTrainData <- initialTheta(maxIterations = 2, predictors = trainPredictors, response = trainResponse)  
  
# Using the optimized theta vector from trained model, predict onto the test dataset.  
pred <- logRegr\_Pred(x\_df\_no1s = select(testPredictors, -X\_0), y\_df = testResponse, theta = optmzThetaTrainData)  
results <- pred$results  
PCC <- pred$PCC  
PCC # view PCC  
  
# Calculate minimum value of Objective function  
minObjFctnVal <- calcObjFctn(xs = trainPredictors, y = trainResponse, theta = optmzThetaTrainData, lambda = 1)  
minObjFctnVal # See what the minimum value of the obj fctn is for optimized theta.  
  
  
# Criterion I used for selecting the "worst" misclassifed instances (logRegr was confident, but wrong):  
# The 20 misclassified images will be those whose output probability is furthest from 0.5  
# (and obviously for which the classification is incorrect).  
# Based on examination of the data, fours are given y values of -1, and nines are given y values of 1.  
# To see which are the "worst" misclassified, we're looking for the output probabilities that are most  
# extreme (nearest 0 or 1) which also have a misclassification.  
# Simply take the absValue(predProb - 0.5) to get confidence, then rank-order these.  
# Only keep those with misclassification (correctIndVar == 0).  
# Then arrange in order of descending 'confLogRegridence' (values closest to 0.5 will be at the top).  
# Keep the top 20 closest values to 0.5.  
results$confLogRegr <- abs(results$probabilities - 0.5000000)  
results$index <- as.numeric(row.names(results))  
# As of the line below, can still see the index in the dataset  
allWrongObs <- results[results$correctIndVar == 0,]  
allWrongObs <- arrange(allWrongObs, desc(confLogRegr))  
# allWrongObs <- results[order(-results$confLogRegr),]  
# Above line should do the same thing as the dplyr code, but doesn't, weird.  
worst20obs <- allWrongObs[1:20,]  
worst20obs.index <- as.numeric(row.names(worst20obs))  
  
  
# Read in the MNIST data in order to display misclassified images  
mnist <- readMat("mnist\_49\_3000.mat")  
imgList = list()  
for(i in seq(1, length(mnist$x), by = 784)) {  
 img <- matrix(mnist$x[i:(i + 783)], nrow = 28, byrow = TRUE)  
 img <- t(apply(img, 2, rev))  
 imgList[[length(imgList)+1]] = img  
}  
  
# This will create the desired 4x5 panel with the 20 'worst' misclassified  
# images (and their correct classification). Loops through the worst 20, giving  
# the image and the correct label above it.  
par(mfrow = c(4,5))  
counter <- 1  
for(i in worst20obs.index){  
 print(i)  
 correctLabel = "intial"  
 if(worst20obs$Y[counter] == -1){  
 correctLabel = "Correct Label: 4"  
 } else {  
 correctLabel = "Correct Label: 9"  
 }  
 image(1:28, 1:28, imgList[[i]], col=gray((255:0)/255), main = correctLabel,  
 xlab = "", ylab = "")  
 counter <- counter + 1  
}  
  
  
# Original Gradient & Hessian Functions (not the CMU versions: math should give same results tho)  
{  
 # gradJ <- function(xmat, yvec, theta){  
 # lambda <- 10  
 # vecs <- data.frame(matrix(ncol = 3000, nrow = 785))  
 # for(i in 1:nrow(xmat)){  
 # xi <- as.numeric(xmat[i,])  
 # yi <- as.numeric(yvec$Y[i])  
 # val <- (exp(dot(-theta,xi)) \* xi)/(1 + exp(dot(-theta,xi))) + ((yi\*xi) - (xi))  
 # vecs[,i] <- val  
 # }  
 # gradient <- (-1 \* rowSums(vecs)) + 2\*lambda\*theta  
 # return(gradient)  
 # }  
   
 # hessJ <- function(xs,theta){  
 # lambda <- 10  
 # twoLambIdMatrix <- diag(rep(2 \* lambda, 785))  
 #   
 # ds <- double(nrow(xs))  
 #   
 # for(i in 1:nrow(xs)){  
 # xi <- as.numeric(xs[i,])  
 # entry <- exp(dot(-theta,xi))/((1 + exp(dot(-theta,xi)))^2)  
 # ds[i] <- entry  
 # }  
 # terms <- double(nrow(xs))  
 # for(i in 1:nrow(xs)){  
 # xi <- as.numeric(xs[i,])  
 # }  
 # D <- diag(ds)  
 # XT\_matrix <- t(sapply(xs, as.numeric))  
 # X\_hess\_matrix <- sapply(xs, as.numeric)  
 # hess <- XT\_matrix %\*% D %\*% X\_hess\_matrix  
 # hess <- hess + twoLambIdMatrix  
 # return(hess)  
 # }  
}