Project 4

Will Graham; Richelle Lee; Robin Lin

2022-11-18

# Will Graham; Richelle Lee; Robin Lin  
# 2022-11-17  
# Github repository:  
# https://github.com/willlewisgraham/Statistical-Programming-2022.git  
#  
# Group Contributions:  
# Instead of splitting tasks between our group, we worked collaboratively on each step of the process.  
# Contributions were 1/3 per team member.

# Many statistical learning approaches rely on numerical optimization. This  
# code file is an implementation of Newton's method for function minimisation.  
#  
# In many cases, an objective function cannot be easily optimised with a closed  
# form solution (like when optimising maximum likelihoods). At a high level,  
# Newton's method works by iteratively approximating these objective functions   
# with quadratic functions, and then updating the function parameters to   
# minimise these quadratic functions (which can easily be done with closed form  
# solutions). These updated parameters constitute improved guesses of the true  
# optimal parameters for the objective function.

hess\_with\_finite\_differencing <- function(current\_theta, grad, eps){  
 # This function uses finite differencing to approximate the hessian matrix  
 # of an objective function with respect to a supplied parameter vector of   
 # thetas.   
 #  
 # Inputs:  
 # current\_theta - a parameter vector that the hessian will be evaluated at  
 # grad - the gradient function. Returns the gradient vector of the objective  
 # w.r.t. the elements of parameter vector  
 #   
 # Returns:  
 # a Hessian matrix approximated with finite differencing  
   
 p <- length(current\_theta) # number of parameters in theta  
 hess <- matrix(0,p,p) # initialize hessian approximation  
   
 for (i in 1:p){ # loop over parameters  
 change\_vector <- matrix(0,p)   
 change\_vector[i] <- eps / 2 # used to change one parameter of theta  
   
 # gradient evalauted at current theta - epsilon / 2  
 lower\_grad <- grad(current\_theta - change\_vector)   
   
 # gradient evaluated at current theta + epsilon / 2  
 upper\_grad <- grad(current\_theta + change\_vector)  
   
 hess[i,] <- (upper\_grad - lower\_grad) / eps # store finite approximation  
 }  
   
 return(hess)  
   
}

newt <- function(theta, func, grad, hess = NULL, ..., tol = 1e-8, fscale = 1,   
 maxit = 100, max.half = 20, eps = 1e-6){  
 # This function applies Newton's ("newt") method for function minimisation.  
 # The newt function was to be set up in a way where it would operate in a   
 # similar way to the functions nlm and optim.  
 #  
 # The newt function arguments are:  
 # newt(theta,func,grad,hess=NULL,...,tol=1e-8,fscale=1,maxit=100,max.half=20,   
 # eps# =1e-6)  
 #  
 # Where 'theta' is the vector of initial values for the optimisation   
 # parameters;  
 # 'func' is the objective function to be minimised;  
 # 'grad' is the gradient function;  
 # 'hess' is the Hessian matrix;  
 # '...' are arguments of 'func', 'grad', and 'hess';  
 # 'tol' is the convergence tolerance;  
 # 'fscale' is an estimate of the value of the objective function near the   
 # optimum;  
 # 'maxit' is the limit of newt iterations to try before stopping.  
 # 'max.half' denotes the restriction on the number of times a step should be   
 # halved before concluding that the step was not able to improve the   
 # objective;  
 # and 'eps' is the finite difference intervals to be used when a Hessian   
 # function is not supplied.  
 #  
 # The constructed newt function returns the optimised/minimised value of the   
 # objective function, the value of 'theta' at the minimum, the number of   
 # iterations it took to reach the minimum, the gradient at the minimum, and   
 # the inverse of the Hessian matrix at the minimum, while at the same time,  
 # issues errors and warnings in the following scenarios:  
 #  
 # 1. Where the objective or derivatives are not finite at the initial theta;  
 # 2. Where the step fails to bring down the objective after reaching   
 # 'max.half' number of step halvings.  
 # 3. Where 'maxit' is reached without convergence\*.  
 # 4. Where the Hessian is not positive definite at convergence\*.  
 #  
 # \*convergence is assessed by checking whether all absolute values of the   
 # gradient vector is less than the convergence tolerance, 'tol', multiplied by   
 # the absolute value of the objective function plus the estimate value of the   
 # objective function near the optimum, 'fscale'.  
   
   
 # If no hessian matrix function is supplied, default to finite differencing  
 hess.supplied <- TRUE # Default to assume hessian is supplied  
 if(is.null(hess)){ # If hessian is not supplied  
 hess.supplied <- FALSE # Set hessian to not supplied  
 }  
   
 obj\_at\_theta <- func(theta) # Evaluate objective function at initial theta  
 grad\_at\_theta <- grad(theta) # Evaluate gradient at initial theta  
   
 if(hess.supplied){ # Checks whether hessian is supplied, if so,  
 hess\_at\_theta <- hess(theta) # evaluate hessian at inital theta  
 } else { #otherwise,  
   
 # approximate hessian matrix by finite differencing  
 hess\_at\_theta <- hess\_with\_finite\_differencing(theta, grad, eps)  
 }  
   
 # Collate obj, gradient, and hessian values at intial theta into 1 vector,  
 obj\_and\_derivatives <- c(obj\_at\_theta, grad\_at\_theta, hess\_at\_theta)  
   
 # and check that all values are finite.  
 # If not finite, stop the function and throw an error message  
 if (!all(is.finite(obj\_and\_derivatives)) ){  
 stop("Objective Funciton or Derivatives Not Finite at Initial Theta")  
 }  
   
 i <- 0 # Start iterations at 0  
 while(i < maxit){ # Track the number of iterations up to the max allowance.  
 original\_theta <- theta # store value of theta before the current loop runs  
   
 # Evaluates the corresponding function value at theta  
 obj\_at\_theta <- func(theta)   
   
 # Evaluates the corresponding gradient value at theta  
 grad\_at\_theta <- grad(theta)  
   
 # Reckon the Hessian.  
 if(hess.supplied){ # If a user specifies the Hessian matrix.  
 hess\_at\_theta <- hess(theta) # Evaluates the value at theta.  
 } else { # If the user doesn't specify the Hessian matrix  
   
 # Applys the finite differencing function, and approximates the hessian  
 # at theta  
 hess\_at\_theta <- hess\_with\_finite\_differencing(theta, grad, eps)  
 }  
   
 # Make a trial of constructing Cholesky decomposition to the Hessian  
 chol\_of\_hess <- try(chol(hess\_at\_theta), silent = TRUE)   
   
 # Check if the cholesky decomposition is possible  
 if(all(class(chol\_of\_hess) != 'try-error')){   
 hess\_is\_pd <- TRUE # Determines that the Hessian is positive definite  
 hess\_is\_inv <- TRUE # Determines that the Hessian is invertible  
   
 # Reckons the inverse of Hessian from Cholesky decomposition  
 inv\_hess <- chol2inv(chol\_of\_hess)   
   
 } else { # Cholesky decomposition not possible --> hessian is not PD  
 hess\_is\_pd <- FALSE # Hessian cannot be positive definite  
   
 # Computes the Eigen-decomposition of the Hessian matrix  
 eig\_hess <- eigen(hess\_at\_theta)   
   
 lambdas <- eig\_hess$values # Identifies the eigenvalue matrix of the  
 if (length(lambdas) == 1){ # check for special case when hessian is 1x1  
 lambdas = matrix(lambdas) # coerce lambda from a scalar to a matrix  
 }  
   
 U <- eig\_hess$vectors # Identifies matrix of eigenvectors of the Hessian  
   
 # If there is an eigenvalue of 0, hessian is not invertible  
 if(0 %in% lambdas){   
 hess\_is\_inv <- FALSE # Hessian is not invertible.  
 } else { # If all eigenvalues are non-zero  
 hess\_is\_inv <- TRUE # Hessian is invertible.  
   
 # Calculates the inverse of Hessian by Eigen-decomposition.  
 inv\_hess\_before\_perturb <- U %\*% (diag(1 / lambdas)) %\*% t(U)   
 }  
   
 # Perturb the Hessian by the absolute value of the smallest eigenvalue,  
 # plus one  
 pert\_hess\_at\_theta <- hess\_at\_theta + (-(min(lambdas)) + 1) \*   
 diag(dim(as.matrix(hess\_at\_theta))[1])   
   
 # Calculates the inverse of the perturbed Hessian  
 inv\_hess <- chol2inv(chol(pert\_hess\_at\_theta))   
 }   
   
 # Determine whether the convergence condition is met.  
 if (all(abs(grad\_at\_theta) < tol \* (abs(obj\_at\_theta) + fscale))){   
 # If all elements in the gradient less than tolerance multiples of  
 # absolute value of objective function plus a rough estimate of the   
 # magnitude of the objective function near the optimum, the convergent  
 # condition is met.  
   
 # If the Hessian is positive definite, returns a list of   
 # the following variables: optimal value, optimal solution, number of   
 # iterations, gradient at optimal solution, and inverse of Hessian   
 # matrix at optimal solution.  
 if(hess\_is\_pd){   
 return(list('f' = obj\_at\_theta, 'theta' = theta, 'iter' = i,   
 'g' = grad\_at\_theta, 'Hi' = inv\_hess))  
 }   
   
 # However, if Hessian is not positive definite at convergence, give a   
 # warning first, and return values anyways  
 warning('Hessian is not positive definite at convergence.\n')  
   
 # If Hessian is invertible, return the similar list, but   
 # return the inverse of the Hessian as the one before perturbation  
 if(hess\_is\_inv){   
 return(list('f' = obj\_at\_theta, 'theta' = theta, 'iter' = i,   
 'g' = grad\_at\_theta, 'Hi' = inv\_hess\_before\_perturb))  
 }   
   
 # If Hessian is not invertible, return a message indicating that it is   
 # not invertible  
 return(list('f' = obj\_at\_theta, 'theta' = theta, 'iter' = i,  
 'g' = grad\_at\_theta, 'Hi' = "Hessian is not invertible"))  
 }  
   
 descent\_direction <- as.vector(-inv\_hess %\*% grad\_at\_theta)  
 stepsize <- 1 # Tentatively use the full descent direction  
 halves <- 0 # Initialise half step variable  
   
 # Update theta by moving it to a step size multiple of descent direction  
 theta\_hat <- theta + stepsize \* descent\_direction   
   
 suppressWarnings( # suppress default NaN warnings if they occur  
   
 # Keep attempting to halve the step size as long as the maximum number of  
 # halves have not been tried, and any of the following conditions are met:  
 # 1. Obj function increases at proposed theta compared to current theta  
 # 2. Obj function not finite at proposed theta  
 while((obj\_at\_theta < func(theta\_hat) && halves <= max.half) |   
 (!(is.finite(func(theta\_hat))) && halves <= max.half)){  
 stepsize <- stepsize / 2   
 theta\_hat <- theta + stepsize \* descent\_direction # halve step size  
 halves <- halves + 1 # increment halve count  
 }  
 )   
   
 if(halves == (max.half + 1)){ # check if maximum half steps were exceeded  
 warning("Maximum number of step halvings reached \n")  
 if(hess\_is\_pd){ # check if hessian is positive definite  
   
 # If the hessian is positive definite, return all values without   
 # additional warnings  
 return(list('f' = obj\_at\_theta, 'theta' = original\_theta, 'iter' = i,  
 'g' = grad\_at\_theta, 'Hi' = inv\_hess))  
 }   
   
 # Warn user that Hessian is not positive definite  
 warning('Hessian is not positive definite at max halve steps.\n')  
 if(hess\_is\_inv){ # check if hessian is invertible  
   
 # return all values if hessian is invertible  
 return(list('f' = obj\_at\_theta, 'theta' = original\_theta, 'iter' = i,  
 'g' = grad\_at\_theta, 'Hi' = inv\_hess\_before\_perturb))  
 }  
   
 # not possible to return inverse hessian if hessian is not invertible  
 return(list('f' = obj\_at\_theta, 'theta' = original\_theta, 'iter' = i,  
 'g' = grad\_at\_theta, 'Hi' = "Hessian is not invertible"))  
 }  
   
 theta <- theta\_hat # update theta  
 i <- i + 1 # increment iteration count  
 }  
   
 # If this portion of the code is reached, that means that the maximum number  
 # of iterations has run and convergence was not reached  
 warning("Maximum iterations reached without convergence \n")  
 if(hess\_is\_pd){ # check if hessian is positive definite  
   
 # If the hessian is positive definite, return all values without additional  
 # warnings  
 return(list('f' = obj\_at\_theta, 'theta' = original\_theta,  
 'iter' = maxit, 'g' = grad\_at\_theta, 'Hi' = inv\_hess))  
 }  
   
 # Warn user that Hessian is not positive definite  
 warning('Hessian is not positive definite at max iterations.\n')  
 if(hess\_is\_inv){ # check if hessian is invertible  
   
 # return all values if hessian is invertible  
 return(list('f' = obj\_at\_theta, 'theta' = original\_theta, 'iter' = maxit,  
 'g' = grad\_at\_theta, 'Hi' = inv\_hess\_before\_perturb))  
 }  
   
 # not possible to return inverse hessian if hessian is not invertible  
 return(list('f' = obj\_at\_theta, 'theta' = original\_theta, 'iter' = maxit,  
 'g' = grad\_at\_theta, 'Hi' = "Hessian is not invertible"))  
}