STAT40780 Data Programming with C (online)

Lab Sheet 10 (Solutions)

Dr Marie Galligan

Summer 2015

This week's lab involves learning about the gradient descent algorithm, and getting more practice with Rcpp, and Rcpp sugar. Solutions will be provided in the R script file lab10_solutions.R

1 Gradient descent algorithm

This task involves implementing a gradient descent algorithm in C++ with the help of Rcpp. But first, some background...

Working with data often involves minimization or maximization of functions with respect to one or more parameters. Consider the following convex function with a single parameter θ:

$$f(\theta) = 1 + 3(\theta + 3)^2$$

A gradient descent algorithm can be used to find the value of θ that minimizes $f(\theta)$. To implement gradient descent, you need to know the derivative of the function you are trying to minimize, with respect to each parameter. In this case, there is a single parameter θ and hence only a single derivative is required:

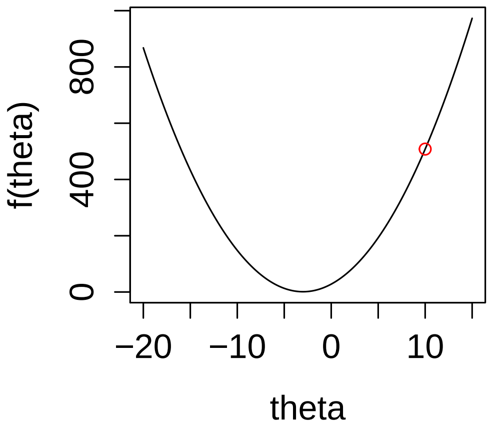
$$\frac{d}{d\theta} f(\theta) = 6(\theta + 3)$$

Remember that the derivative of this function is equal to its slope at a value θ. If the derivative is positive, it means that the function is increasing at theta, while a negative derivative indicates that the function is decreasing at theta.

The gradient descent algorithm for a single parameter works as follows:

1. Choose an initial value for θ (initial guess)
2. Calculate the derivative of the function at the current 'guess' for θ
3. Update the current 'guess' for theta by taking a step in the direction that decreases $f(\theta)$

Figure 1: Plot of $f(\theta)$. Red dot shows a current 'guess' of $\theta = 10$, where the objective is to find θ to minimize $f(\theta)$.



4. Repeat steps 2 and 3 until convergence to the minimum

At step 3, the current guess for θ is updated by taking a step either to the right or left of the current value of theta. Whether the value of θ is moved to the right or to the left depends on the derivative of θ at that point.

- if the derivative is positive at theta, $f(\theta)$ is increasing with θ, hence decreasing the value of θ will decrease $f(\theta)$
- if the derivative is negative at theta, $f(\theta)$ is decreasing with θ, hence increasing the value of θ will decrease $f(\theta)$

Suppose the current guess for $\theta = 10$ (see red dot in Figure 1). At this point, the function $f(\theta)$ is increasing with theta, and this will be reflected in the derivative (derivative will be positive at $\theta = 10$. Hence, to reach the minimum, the update in gradient descent will reduce the value of θ by a little bit.

At iteration j of the gradient descent algorithm, denote the previous guess for θ as θ^{j-1}. The rule for updating θ^{j-1} to a new guess θ^j is:

$$\theta^j = \theta^{j-1} - \alpha \frac{d}{d\theta} f(\theta^{j-1})$$

This update moves θ in the direction that decreases $f(\theta)$. Here, α is called the "learning rate" and determines the size of the step taken in gradient descent. A larger value for α means that the algorithm takes larger steps.

The following R code implements gradient descent to find the minimum of the function $f(\theta)$ defined above. The algorithm plots the current guess for θ at each iteration, and operates in slow motion to allow you to see individual updates being added to the plot.

Convergence of the algorithm is assessed at each iteration j by monitoring the absoluate relative change in $f(\theta)$:

$$rel\_ch = \left| \frac{f(\theta^j) - f(\theta^{j-1})}{f(\theta^{j-1})} \right|$$

If this relative change becomes really small (less than some specified level of tolerance - defined here as *tol*), the algorithm is assumed to have reached convergence.

Copy the code into an R script file and run. Be careful, as copy and paste moves the code around a little - you will probably need to move some things around! Experiment with different 'initial guess' values for theta, and different values for α.

gradient descent algorithm

```


#FIRST, DEFINE THE FUNCTION TO BE MINIMIZED f(theta)
objfun <- function(theta){
  return(1 + 3*(theta + 3)^2 )
}

#DEFINE A FUNCTION THAT CALCULATES THE DERIVATIVE f'(theta)
# AT INPUT VALUE theta
deriv <- function(theta)
{
  return( 6*(theta + 3 ) )
}

#PLOT THE FUNCTION WE ARE SEEKING TO MINIMIZE OVER THE RANGE [-15, 15]
theta_seq <- seq(-15,15, len = 1000)
plot(theta_seq, objfun(theta_seq), type = "l",
     ylab = "f(theta)", xlab = "theta")

 tol <- 0.0000001 #CONVERGENCE THRESHOLD
 alpha <- 0.01  #LEARNING RATE
 theta0 <- 10  #SELECT INITIAL GUESS FOR THETA
 newval <- objfun( theta0 ) #inital value of f(theta)
 points(theta0, newval, col = "red") #add current theta to plot
 rel_ch <- 1 #initialize relative change marker to any value larger than tol
 j <- 1 #iteration counter
 theta <- c() #vector to store theta at each iteration
 theta[j] <- theta0 #theta[j] stores current value of theta

 #update theta while relative change in f(theta) is greater than tol
 while( rel_ch > tol )
 {
  j <- j+1; #increment j
  #update theta
  #set theta_new =  theta_previous - ( learning rate )* f'(theta_previous)
  theta[j] <- theta[j-1] - alpha * deriv(theta[ j-1 ])

  #test relative absolute change in target function
  oldval <- newval #store f(theta_previous)
  newval <- objfun(theta[j]) #calculate f(theta_new)
  points(theta[j], newval, col = "red") #add new theta to plot
  Sys.sleep(0.1) #pause algorithm to give you time to see dot appear on plot
  #calculate relative change in f(theta)
  rel_ch <- abs(  ( newval - oldval ) / oldval ) #use to test convergence
 }
```

Be sure that you understand the gradient descent algorithm. The following resources give a more detailed explanation:
`http://bayen.eecs.berkeley.edu/bayen/?q=webfm_send/262` `https://www.youtube.com/watch?v=Fn8qXpIcdnI`
and there are many others!

TASK: Implement this gradient descent algorithm in C++ with help from Rcpp.

A possible solution

R script for: gradient descent in C++

```
#Define C++ functions to calculate f(theta) and derivative
incl <- '
#include <iostream>
using std::endl;

double obj( double theta )
{
  return( 0.1 + 0.1*theta + pow(theta,2) /( 0.1 + pow(theta, 2) ) );
}

double deriv( double theta )
{
  return( 0.1 + ( 0.2*theta + 4*pow(theta, 3) ) / pow(0.1 + theta, 2) );
}
'

body_gdC <- '
 double val0 = as<double>(theta0);
 double tol = as<double>(tolerance);
 double learnrate = as<double>(alpha);

 double theta = val0; //current value of theta
 double theta_prev; //stores previous value of theta
 double rel_ch = 1.0;
 double ftheta = obj( theta );  //stores value of target fn at current theta
 Rcout << "Objective function = " << ftheta << std::endl ;
 double ftheta_prev; //to store value of target function at previous theta
 int j = 0;

 while( rel_ch > tol )
 {
  j++;
  theta_prev = theta;
  ftheta_prev = ftheta;
  theta = theta_prev - learnrate * deriv( theta_prev );
  ftheta = obj( theta );
  Rcout << "Objective function = " << ftheta << std::endl;
  rel_ch =  fabs( ( ftheta - ftheta_prev ) / ftheta_prev ) ;
 }

 return wrap( List::create(
              _["theta"] = theta,
              _["ftheta"] = obj( val0 )
              ));
'

#compile, link, load
gdC <- cxxfunction( signature( theta0 = "numeric" ,
         tolerance = "numeric", alpha = "numeric"),
                    body = body_gdC,
                    includes = incl,
                    plugin = "Rcpp")

gdC( -0.5 , 0.00000000001, 0.01) #call C++ from R
```

2 Choosing the learning rate α

In the gradient descent algorithm implemented in the previous task, a fixed learning rate was chosen e.g. $\alpha = 0.01$. Choosing α can have a large impact on the algorithm. If α is too large, the function may not converge, instead bouncing around, missing the minimum. If α is too small, the algorithm can be slow to converge. See the following video for details:

`http://www.dailymotion.com/video/x2clpb9_4-4-machine-learning-gradient-descent-in-practice-ii-learning-rate_school`

From the video, you can see that if the learning rate is too large, the function you are trying to minimize could actually increase. In addition, ideally, the algorithm should take large steps when far away from the minimum value to move faster towards it, and take smaller steps as it approaches the minimum (to be sure it doesn't miss it).

The learning rate can be adapted as the algorithm progresses to ensure that this happens, using the Bold Driver method. At each iteration, check whether the target function $f(\alpha)$ has decreased since the previous iteration i.e. check whether

$$f(\theta^j) - f(\theta^{j-1}) < 0$$

If the the target function HAS decreased (i.e. got closer to minimum), then increase the learning rate a little on the next iteration by 5% so that larger steps are taken. However, if the target function has actually increased (BAD!), the algorithm has missed the minimum, so the learning rate should be decreased drastically by 50% on the next iteration (to ensure smaller steps are taken).

Task: implement the Bold Driver method in your C++ gradient descent function to adapt the learning rate at each iteration.

A possible solution

R script for: gradient descent in C++ with Bold Driver method

```
#same as question 1 solution but with Bold Driver method implemented
#adapts learning rate alpha at each iteration
incl <- '
#include <iostream>
using std::endl;

double obj( double theta )
{
return( 0.1 + 0.1*theta + pow(theta,2) /( 0.1 + pow(theta, 2) ) );
}

double deriv( double theta )
{
return( 0.1 + ( 0.2*theta + 4*pow(theta, 3) ) / pow(0.1 + theta, 2) );
}
'

body_gdC_2 <- '
 double val0 = as<double>(theta0);
 double tol = as<double>(tolerance);
 double learnrate = as<double>(alpha);

 double theta = val0; //current value of theta
 double theta_prev; //stores previous value of theta
 double rel_ch = 1.0;
 double ftheta = obj( theta );  //stores value of target fn at current theta
 Rcout << "Objective function = " << ftheta << std::endl ;
 double ftheta_prev; //to store value of target function at previous theta
 int j = 0;

 while( rel_ch > tol )
 {
  j++;
  theta_prev = theta;
  ftheta_prev = ftheta;
  theta = theta_prev - learnrate * deriv( theta_prev );
  ftheta = obj( theta );
  if( ftheta - ftheta_prev <= 0 ) //if target decreased or remained the same
  {
    learnrate *= 1.05; //increase learning rate by 5%
  }else{ //if target function increased
    learnrate *= 0.5; //decrease learning rate by 50%
  }
  Rcout << "f(theta) = " << ftheta <<
    " and learning rate = " << learnrate  << std::endl;
  rel_ch =  fabs( ( ftheta - ftheta_prev ) / ftheta_prev ) ;
 }

return wrap( List::create(
_["theta"] = theta,
_["ftheta"] = obj( val0 )
));
'

gdC_2 <- cxxfunction( signature( theta0 = "numeric" , tolerance = "numeric",
                    alpha = "numeric"),
                    body = body_gdC_2,
                    includes = incl,
                    plugin = "Rcpp")

gdC_2( -0.5 , 0.00000000001, 0.01)
```

3 Rcpp sugar

1. Write a C++ function that calculates the min, max, mean and standard deviation (using Rcpp sugar functions) of a numeric vector passed from R, and returns the results to R in a named vector (elements named “min”, “max”, “mean” and “sd”). Pass this function a vector with some missing elements and observe its behaviour.

2. Rewrite the function to handle missing values.

A possible solution to part (a)

This function does not provide an option to remove missing values before calculating summary measures.

summaryC (missings not handled)

```
body_summaryC <- '
 NumericVector xx(x);
 return wrap( NumericVector::create(
   _["min"] = min(xx),
   _["max"] = max(xx),
   _["mean"] = mean(xx),
   _["sd"] = sd(xx)
  ) );
'

summaryC <- cxxfunction(signature(x = "numeric"),
                        body = body_summaryC,
                        plugin = "Rcpp")

#generate some data to test the function on
x <- rnorm(20)
summaryC(x) #call C++

#now set first 3 elements to missing
x[1:3] <- NA

#how does summaryC() handle missings?
summaryC(x) #call C++
```

A possible solution to part (b)

This function gives an option to remove missing values before calculating summary measures.

summaryC_2 (option to remove missings)

```r
body_summaryC_2 <- '
 NumericVector xx(x);
 int remNA = as<int>(na_rm);
 if(remNA == 1){ //if true
  xx = xx[!is_na(xx)];
 }
 return wrap( NumericVector::create(
   _["min"] = min(xx),
   _["max"] = max(xx),
   _["mean"] = mean(xx),
   _["sd"] = sd(xx)
  ) );
'
summaryC_2 <- cxxfunction(signature(x = "numeric", na_rm = "logical"),
                          body = body_summaryC_2,
                          plugin = "Rcpp")

#generate some data to test the function on
x <- rnorm(20)
summaryC_2(x, na_rm = TRUE) #call C++

#now set first 3 elements to missing
x[1:3] <- NA

#how does summaryC() handle missings?
summaryC_2(x, na_rm = TRUE) #remove missings
summaryC_2(x, na_rm = FALSE) #don't remove missings
```