STAT 432 Homework-2

Sharvi Tomar (stomar2)

30/08/21

Contents

Question 1 Optimizing a Mean Model

For this question, you need to write functions to iteratively update the parameter to reduce the functional value, meaning that your function must contain the iteration steps. You can still use other build-in R functions to simplify the calculation. You cannot use the `optim()` function unless explicitly asked to.

The goal of this question is to estimate the mean of a set of samples using numerical optimization. In other words, we observe $\{y_i\}_{i=1}^n$ and using the ℓ_2 loss, which is the same as a regression model, we want to estimate θ by minimizing the objective function

$$\ell(\theta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \theta)^2$$

```
# Generating 100 observations and setting seed to UIN.
set.seed(667346304)
y = rnorm(100)
mean(y)
```

```
## [1] 0.04883245
```

a) [10 Points] Write a function `mean_loss(theta, trainy)` that calculates the loss function given an θ values and a vector **y** of observations.

```
# Function to calculate loss function for a given theta and vector y of observations
mean_loss<-function(theta, trainy)
{
  return(mean((trainy-theta)^2))
}
```

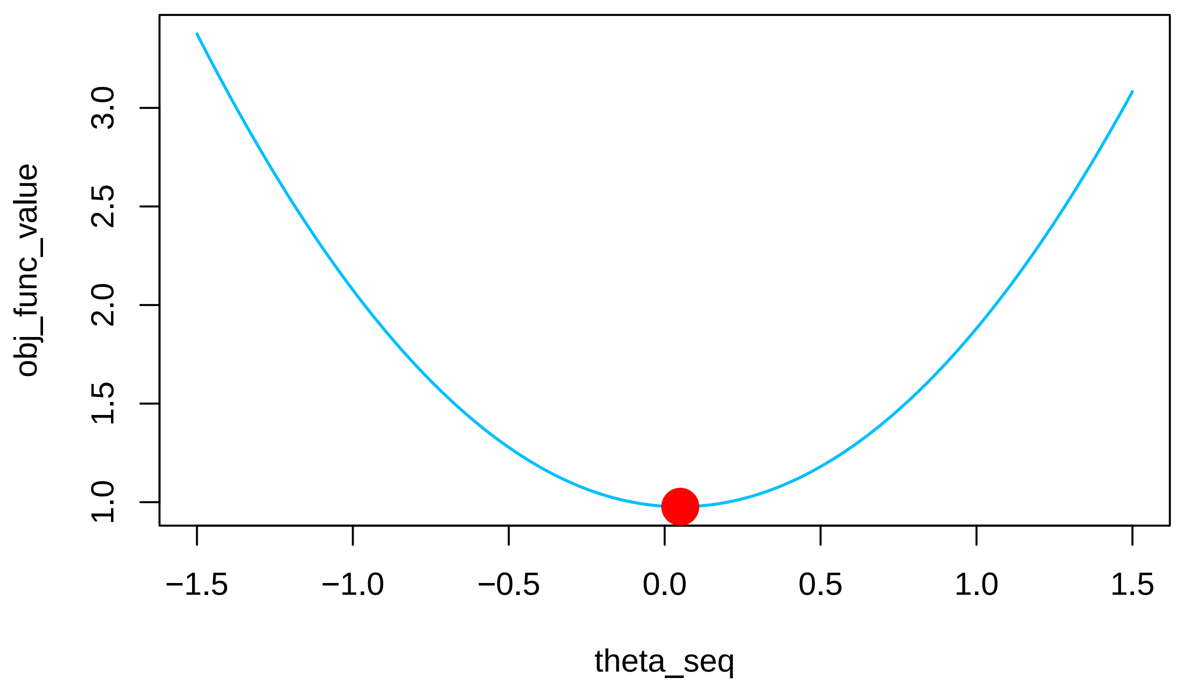
b) [10 Points] Use your function to calculate its value at a grid of θ values using `seq(-1.5, 1.5, 0.01)`. Plot the objective function in a figure, with θ as the horizontal axis and the objective function value as the vertical axis. The figure may look similar to this one. Add the optimal point to the figure.

```r
# A grid of theta values
theta_seq=seq(-1.5, 1.5, 0.01)
# Using mean_loss() function to calculate objective function value for theta values
obj_func_value_lists=lapply(theta_seq,mean_loss,y)
# Printing the first 5 values of objective function as calculated by mean_loss()
obj_func_value_lists[1:5]
```

```
## [[1]]
## [1] 3.375641
##
## [[2]]
## [1] 3.344764
##
## [[3]]
## [1] 3.314088
##
## [[4]]
## [1] 3.283611
##
## [[5]]
## [1] 3.253334
```

```r
# Combining the list of lists into a single list(flatten the list)
obj_func_value=unlist(obj_func_value_lists, recursive = FALSE)
# Creating a data frame with theta value and corresponding objective function value
df=data.frame(theta_seq,obj_func_value)
# Finding the min value of objective function and its corresponding theta value
optimal_point=df[which.min(df$obj_func_value),]


# Plot of objective function(vertical axis) with theta(horizontal axis) & labelled optimal point
plot(theta_seq, obj_func_value, type = "l", col = "deepskyblue", lwd = 1.5)
points(optimal_point[1], optimal_point[2], col = "red", pch = 19, cex = 2.5)
```



The plot shows the objective function(calculated by the mean_loss() function) as the vertical axis with θ as the horizontal axis . The labelled red point is the optimal point i.e. the point with the smallest value of objective function in the given range of theta values calculated using the mean_loss() function.

c) [10 Points] Test this function using the `optim()` function and obtain the optimizer, which should be the same as the sample mean. Use 1 as the starting value and "BFGS" as the method. Compare the result with the true sample mean.

```
# Creating optimizer using optim function & setting starting value=1, method= "BFGS"
optim_result<-optim(par = 1, mean_loss, trainy=y, method="BFGS")
# Printing the output of optim() optimizer
optim_result
```

```
## $par
## [1] 0.04883245
##
## $value
## [1] 0.9767589
##
## $counts
## function gradient
##        4        3
##
## $convergence
## [1] 0
##
```

```
## $message
## NULL
```

```
# The value of theta parameter as obtained from the optim() function is
optim_result$par
```

```
## [1] 0.04883245
```

```
# Comparing the theta value obtained with the true sample mean
optim_result$par-mean(y)
```

```
## [1] 1.324635e-14
```

The difference in the values of optimal theta as obtained from the optim() and the mean_loss() function is of the order e^-14 which is a very small value.

d)i) [10 Points] Derive the gradient (derivative with respect to θ) of the objective function, and write that down using Latex

Objective function

$$l(\theta) = \frac{\sum_{i=1}^{n}(y_i - \theta)^2}{n}$$

Differentiating objective function w.r.t. to θ to obtain the gradient of objective function

$$\frac{\partial l(\theta)}{\partial \theta} = -2\frac{\sum_{i=1}^{n}(y_i - \theta)}{n}$$

ii) [5 Points] Perform a calculation of this gradient at $\theta = 1$. The result should be positive because the objective function should be moving upwards at $\theta = 1$. However, note that during the parameter update, you should move towards the negative gradient.

```
# Gradient function
my_gd<-function(theta, trainy){
  return(mean(trainy-theta)*(-2))
}
# Calculating the value of gradient at theta=1
my_gd(1,y)
```

```
## [1] 1.902335
```

The result is positive as per expectation.

iii) [25 Points] Write your own function `optim_mean_g(trainy, theta0, delta, epsilon, maxitr)` to solve for the optimizer. If you need an example of this, see the First-order Methods section of this week's lecture. Make sure that you keep the history of the iterations for a plot later. Here, * `theta0` is the initial value * `delta` is the step size, with default value 0.3. * `epsilon` is the stopping rule criteria * `maxitr` is the maximum number of iterations

```r
# gradient descent function, which also record the path
optim_mean_g <- function(y,
                   theta0, # initial value
                   delta = 0.3, # step size
                   epsilon = 1e-6, #stopping rule
                   maxitr = 5000) # maximum iterations
{
  if (!is.vector(y)) stop("y must be a vector")

  # initialize theta values
  allt = matrix(theta0, 1, length(theta0))

  # iterative update
  for (k in 1:maxitr)
  {
    # the new theta value
    theta1 = theta0 + sum(y - theta0) * 2*delta / length(y)

    # record the new theta
    allt = rbind(allt, as.vector(theta1))

    # stopping rule
    if (max(abs(theta0 - theta1)) < epsilon)
      break;

    # reset theta)
    theta0 = theta1
  }
  if (k == maxitr) cat("maximum iteration reached\n")
  return(list("allt" = allt, "theta" = theta1, "iterations" = k))
}
```

iv) [10 Points] Finally, run your algorithm with initial value $\theta_0 = 1$ and report the optimizer.

```r
# Running the optim_mean_g() function with theta0=1 and default step-size=0.3
output1=optim_mean_g(y,theta0=1)
# Reporting the output
output1
```

```
## $allt
##              [,1]
##  [1,] 1.00000000
##  [2,] 0.42929947
##  [3,] 0.20101926
##  [4,] 0.10970717
##  [5,] 0.07318234
##  [6,] 0.05857241
##  [7,] 0.05272843
##  [8,] 0.05039084
##  [9,] 0.04945581
## [10,] 0.04908179
## [11,] 0.04893219
## [12,] 0.04887235
## [13,] 0.04884841
## [14,] 0.04883883
## [15,] 0.04883500
## [16,] 0.04883347
## [17,] 0.04883286
##
## $theta
## [1] 0.04883286
##
## $iterations
## [1] 16
```

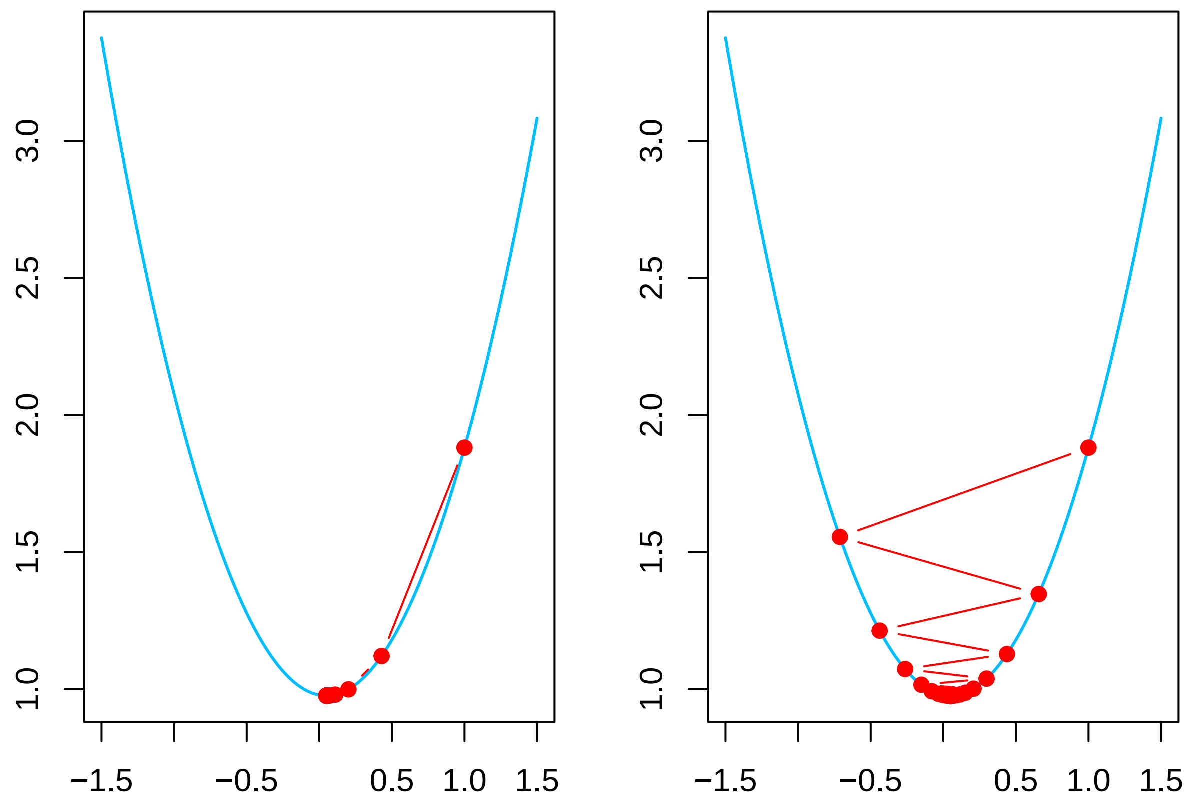
v) [10 Points] Plot the path of your gradient descent in the figure you constructed previously. Choose step size = 0.9. What do you observe? Comment on the difference between these two situations and the impact of the step size.

```r
# Running the optim_mean_g() function with theta0=1 and step-size=0.9
  output2=optim_mean_g(y,theta0=1,delta = 0.9)

  par(mfrow=c(1,2))
  par(mar=c(2,2,2,2))

  # Left plot with initial value theta_0 = 1 and step-size=0.3
  plot(theta_seq, obj_func_value, type = "l", col = "deepskyblue", lwd = 1.5)
  points(output1$allt,lapply(output1$allt,mean_loss,y),type = "b", col = "red", pch = 19)

  # Right plot with initial value theta_0 = 1 and step-size=0.9
  plot(theta_seq, obj_func_value, type = "l", col = "deepskyblue", lwd = 1.5)
  points(output2$allt,lapply(output2$allt,mean_loss,y),type = "b", col = "red", pch = 19)
```



The plot shows the path of the gradient descent with step-size=0.3 and step-size=0.9 respectively. We can see that for both the values of step-size, the objective function seems to converge to the optimal point.

Parameters' initialization such as the step-size and starting value play a critical role in speeding up convergence. In general, smaller step-size may take longer to converge but a very large step size may cause you to overstep local minima. Thus, a right balance should be maintained while setting the step-size.

We can see the plot on the right to have taken more iterations for convergence.

```
# Comparing the number of iterations taken by optimization function to converge in both cases
output1$iterations
```

```
## [1] 16
```

```
output2$iterations
```

```
## [1] 66
```

For the same starting point(theta_0 = 1), the optimization algorithm converges faster for the smaller step-size in contrast to the expectation. This is because large step-size led to overstepping the minima point and thus taking longer to converge.