STA6106 Statistic Computing Project2

Robert Norberg, Jung-Han Wang

Wednesday, October 29, 2014

Problem1

The function g is given by

$$g(x, y) = 4xy + (x + y^2)^2$$

The goal is to find the minimum of g.

a) Minimize g using Newton's method

The Newton's method update is given by

$$\boldsymbol{x}^{(t+1)} = \boldsymbol{x}^{(t)} - \boldsymbol{g}''(\boldsymbol{x}^{(t)})^{-1}\boldsymbol{g}'(\boldsymbol{x}^{(t)})$$

Where $\boldsymbol{g}'(\boldsymbol{x}^{(t)})$ is the gradient and $\boldsymbol{g}''(\boldsymbol{x}^{(t)})$ is the hessian of $g(x, y)$ evaluated at $(x^{(t)}, y^{(t)})$. We use the R function `deriv3()` to define another function for us that will compute the gradient and Hessian of $\boldsymbol{x}$ for given values of x and y.

```
deriv_g <- deriv3(g~4*x*y+(x+y^2)^2, c('x', 'y'), c('x', 'y'))
```

`deriv_g()` is now a function that takes arguments `x` and `y` and will return an object with attributes `gradient` and `hessian`. We demonstrate the use of this function using $(x, y) = (1, 1)$.

```
my_deriv <- deriv_g(x=1, y=1) # test the function deriv_g()
attr(my_deriv, 'gradient') # checks out
```

```
     x  y
[1,] 8 12
```

```
attr(my_deriv, 'hessian') # checks out
```

```
, , x

     x y
[1,] 2 8

, , y

     x  y
[1,] 8 16
```

On scratch paper we found the gradient and hessian by hand and confirmed that this function returns the appropriate values.

We leverage this function to create another function that will provide the Newton update given a value of x and y.

```
newton_update <- function(x, y){
  my_deriv <- deriv_g(x=x, y=y) # evaluate gradient and hessian
  gradient <- matrix(attr(my_deriv, 'gradient'), nrow=2) # extract gradient
  hessian <- matrix(attr(my_deriv, 'hessian'), nrow=2) # extract hessian
  update <- c(x, y) - c(solve(hessian)%*%gradient) # calculate update
  names(update) <- c('x', 'y')
  return(update)
}
```

We will choose some starting values $(x^{(0)}, y^{(0)})$ and continue to update these values until some criterion is met. We define the stopping criteria to be

$$D(\boldsymbol{x}^{(t+1)}, \boldsymbol{x}^{(t)}) \leq \epsilon$$

where $D(\boldsymbol{x}^{(t+1)}, \boldsymbol{x}^{(t)})$ is the distance between the values of $\boldsymbol{x}$ at iteration t and $t+1$. This distance may be computed several ways. We take advantage of the `dist()` function in R to calculate distances for us and we write our code in such a way that the user may easily choose which measure of distance they prefer. We choose the familiar euclidean distance for this problem.

We set $\epsilon = 1E-9$, choose initial values $\boldsymbol{x}^{(0)} = (x^{(0)}, y^{(0)}) = (1, 1)$ and begin to iterate.

```
dist_type <- 'euclidean' # see ?dist for other options
epsilon <- 1E-9 # convergence criterion
x_current <- c('x'=1, 'y'=1) # inital values
x_prev <- c('x'=Inf, 'y'=Inf) # no previous values yet!
counter <- 0 # for counting iterations
verbose <- TRUE # do you want to get updates?

while(dist(rbind(x_current, x_prev), method=dist_type) > epsilon){
  counter <- counter+1
  x_prev <- x_current
  x_current <- newton_update(x=x_current['x'], y=x_current['y'])
  if(verbose==T){
    msg <- paste0('Iteration ', counter, ': x = (', x_current[1], ', ', x_current[2], ')')
    cat(msg)
    cat('\n')
    }
  }
```

```
Iteration 1: x = (2, -0.25)
Iteration 2: x = (0.878676470588236, -0.544117647058824)
Iteration 3: x = (0.927155651746275, -0.692165022218875)
Iteration 4: x = (0.890249404538808, -0.667820449643462)
Iteration 5: x = (0.888891768038794, -0.666668996633616)
Iteration 6: x = (0.888888888900776, -0.666666666676354)
Iteration 7: x = (0.888888888888889, -0.666666666666667)
```

The algorithm eventually converges and we find that $g(x, y)$ has an optimum at $(0.8889, -0.6667)$. Note that given different initial values the algorithm might converge to a different optimum, which may or may not be the global minimum of the function. In fact, the function $g(x, y)$ has no global minimum. If $x = -1 * y^2$, the function decreases infinitely as y increases.

b) Minimize g using the steepest descent method. Use (1,0) as starting point.

Using Golden-Section method we wish to find $x_l < x_m < x_r$ such that $g(x_l) \geq g(x_m)$ and $g(x_r) \geq g(x_m)$. We define a function to compute this

```
gsection = function(ftn, x.l, x.r, x.m, tol = 1e-9) {
  # applies the golden-section algorithm to minimize ftn
  # we assume that ftn is a function of a single variable
  # and that x.l < x.m < x.r and ftn(x.l), ftn(x.r) >= ftn(x.m)

  # the algorithm iteratively refines x.l, x.r, and x.m and terminates
  # when x.r - x.l <= tol, then returns x.m

  # golden ratio plus one
  gr1 = 1 + (1 + sqrt(5))/2
  # successively refine x.l, x.r, and x.m
  f.l = ftn(x.l)
  f.r = ftn(x.r)
  f.m = ftn(x.m)
  while ((x.r - x.l) > tol) {
    if ((x.r - x.m) > (x.m - x.l)) {
      y = x.m + (x.r - x.m)/gr1
      f.y = ftn(y)
      if (f.y <= f.m) {
        x.l = x.m
        f.l = f.m
        x.m = y
        f.m = f.y
        } else {
          x.r = y
          f.r = f.y
          }
      } else {
        y = x.m - (x.m - x.l)/gr1
        f.y = ftn(y)
        if (f.y <= f.m) {
          x.r = x.m
          f.r = f.m
          x.m = y
          f.m = f.y
          } else {
            x.l = y
            f.l = f.y
            }
        }
    }
  return(x.m)
  }
```

We define in R our function $g(x, y) = 4xy + (x + y^2)^2$

```
f <- function(x) {
  y <- 4*x[1]*x[2]+(x[1]+x[2]^2)^2
  return(y)
  }
```

Then we define a function to compute the gradient of $g(x, y)$.

```
gradf<- function (x)
  {##Calculate First Derivative of Function to x[1]  (f1)
  f1<-4*x[2]+2*(x[1] + x[2]^2)
  ##Calculate First Derivative of Function to x[2]  (f2)
  f2<-4 * x[1] + 2 * (2 * x[2] * (x[1] + x[2]^2))
  return(c(f1, f2))
  }
```

Next we define a function to perform a line search to find the minimum point between x_l *and* x_r.

```
line.search <- function(f, x, gradf, tol = 1e-9, a.max = 100) {
  # x and gradf are vectors of length d
  # g(a) =f(x +a*gradf) hasa local minumum at a,
  # within a tolerance
  # if no local minimum is found then we use 0 or a.max for a
  # the value returned is x + a*y
  if (sum(abs(gradf)) == 0) return(x) # g(a) constant
  g <- function(a) return(f(x - a*gradf))

  # find a.l < a.m < a.r such that
  # g(a.m) >=g(a.l)  and g(a.m) >= g(a.r)
  # a.l
  a.l <- 0
  g.l <- g(a.l)
  # a.m
  a.m <- 1
  g.m <- g(a.m)
  while ((g.m > g.l) & (a.m > tol)) {
    a.m <- a.m/2
    g.m <- g(a.m)
    }
  # if a suitable a.m was not found then use 0 for a
  if ((a.m <= tol) & (g.m >= g.l)) return(x)
  # a.r
  a.r <- 2*a.m
  g.r <- g(a.r)
  while ((g.m >= g.r) & (a.r < a.max)) {
    a.m <- a.r
    g.m <- g.r
    a.r <- 2*a.m
    g.r <- g(a.r)
    }
  # if a suitable a.r was not found then use a.max for a
  if ((a.r >= a.max) & (g.m > g.r)) return(x - a.max*gradf)
```

```
  # apply golden-section algorithm to g to find a
  a <- gsection(g, a.l, a.r, a.m)
  return(x - a*gradf)
  }
```

Finally we use the functions defined above to perform a steepest descent optimization.

```
descent <- function(f,gradf, x0, tol = 1e-9, n.max = 100) {
  # steepest descent algorithm
  # find a local minimum of f starting at x0
  # function gradf is the gradient of f
  x <- x0
  x.old <- x
  x <- line.search(f, x, gradf(x))
  n <- 1
  while (f(x.old)-(f(x)> tol) & (n < n.max)) {
    x.old <- x
    x <- line.search(f, x, gradf(x))
    n <- n + 1
    }
  return(x)
  }
descent(f,gradf,c(1,0) )
```

```
[1]  0.8889 -0.6667
```

The values that minimize $g(x, y)$ are (0.8888889,-0.6666667), which is identical to the result obtained in part (a).

Problem2

In 1986, the space shuttle Challenger exploded during takeoff, killing the seven astronauts aboard. The explosion was the result of an O-ring failure, a splitting of a ring of rubber that seals the parts of the ship together. The accident was believed to have been caused by the unusually cold weather ($31°F$ or $0°C$) at the time of launch, as there is reason to believe that the O-ring failure probabilities increase as temperature decreases. Data on previous space shuttle launches and O-ring failures is given in the data set challenger provided with the "mcsm" package of R. The first column corresponds to the failure indicators y_i and the second column to the corresponding temperature x_i, $(1 \leq i \leq 24)$.

```
# load challenger data
library(mcsm)
data(challenger)
```

a) The goal is to obtain MLEs for β_0 and β_1 in the following logistic regression model: $log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x$, where p is the probability that at least one O-ring is damaged and x is the temperature. Create computer programs using Newton-Raphson algorithm to find MLEs of $\widehat{\beta}_0, \widehat{\beta}_1$

The log-likelihood of the data, given values of β_0 and β_1 is given by

$$\ell(x|\beta_0, \beta_1) = ln\left(\prod_{i=1}^{n} \frac{e^{\beta_0+\beta_1 x_i}}{1+e^{\beta_0+\beta_1 x_i}}\right)$$

$$= \sum_{i=1}^{n} -ln(1+e^{\beta_0+\beta_1 x_i}) + y_i(\beta_0+\beta_1 x_i)$$

The Newton's method update is given by

$$(\beta_0^{(t+1)}, \beta_1^{(t+1)}) = (\beta_0^{(t)}, \beta_1^{(t)}) - \ell''(\beta_0^{(t)}, \beta_1^{(t)})^{-1}\ell'(\beta_0^{(t)}, \beta_1^{(t)})$$

,

Where $\ell'(\beta_0^{(t)}, \beta_1^{(t)})$ is the gradient of the log-likelihood and $\ell''(\beta_0^{(t)}, \beta_1^{(t)})$ is the hessian of the log-likelihood evaluated at $(\beta_0^{(t)}, \beta_1^{(t)})$.

Calculating the gradient and hessian by hand is rather messy, so we use the R function `deriv3()` to take the algebraic derivatives needed. The call to `deriv3()` defines a function that will compute the gradient and Hessian of the log-likelihood for given values of x_i, y_i, β_0, and β_1.

```
my_deriv_func <- deriv3(l~y*log(1/(1+exp(-b0-b1*x)))+(1-y)*log(1/(1+exp(b0+b1*x))), namevec=c('b0', 'b1
```

Note that we do not include the sum operator in the call to `deriv3()` because the function does not handle sums, but the derivative of a sum is equivalent to the sum of the derivative of each term being summed, so we can just evaluate the gradient and hessian at each (x, y) pair and sum the results to find the overall gradient and hessian for a given β_0, and β_1.

Since the function `my_deriv_func()` created above will evaluate the gradient and hessian at a single (x, y) pair, I define another function to evaluate the gradient and hessian over all (x, y) pairs in the data set. I employ `mapply()` to "apply" the function over the two vectors `mydat$age` and `mydat$kyphosis` simultaneously ("apply" over (x, y) pairs). A call to `apply()` or `sapply()` will only operate over one vector at a time.

```
deriv_all <- function(b0, b1, deriv_func){
  # create gradient and hessian object for each (x, y) pair
  deriv_list <- mapply(deriv_func,
                       x=challenger$temp, y=challenger$oring,
                       MoreArgs=list(b0=b0, b1=b1),
                       SIMPLIFY=FALSE
                       )
  # extract all gradient components
  gradient_list <- lapply(deriv_list, function(g) matrix(attr(g, 'gradient'), nrow=2))
  # and sum them to get overall gradient at (b0, b1)
  gradient <- Reduce('+', gradient_list)
  # extract all hessian components
  hessian_list <- lapply(deriv_list, function(g) matrix(attr(g, 'hessian'), nrow=2))
  # and sum them to get overall hessian at (b0, b1)
  hessian <- Reduce('+', hessian_list)
  return(list('gradient'=gradient, 'hessian'=hessian))
  }
```

Now a simple call to `deriv_all()` will return the gradient and hessian for a given β_0, and β_1. To demonstrate, we find the gradient and hessian of the log-likelihood at $(\beta_0, \beta_1) = (0.5, 0)$.

```
deriv_all(b0=0.5, b1=0, deriv_func=my_deriv_func)
```

```
$gradient
         [,1]
[1,]   -7.317
[2,] -549.935

$hessian
         [,1]   [,2]
[1,]   -5.405   -376
[2,] -376.006 -26414
```

To maximize the log-likelihood of (β_0, β_1) using the Newton-Raphson algorithm, we define a function to compute the Newton update.

```
newton_update <- function(b0, b1){
  # get gradient and hessian
  my_deriv <- deriv_all(b0=b0, b1=b1, deriv_func=my_deriv_func)
  # calculate update
  update <- c(b0, b1) - c(solve(my_deriv$hessian)%*%my_deriv$gradient)
  # retun update
  names(update) <- c('b0', 'b1')
  return(update)
}
```

Finally we define a stopping criteria to terminate the Newton-Raphson iterations when the values of (β_0, β_1) are close enough to an optimum.

We define the stopping criteria to be

$$D((\beta_0^{(t)}, \beta_1^{(t)}), (\beta_0^{(t+1)}, \beta_1^{(t+1)})) \leq \epsilon$$

,

where $D((\beta_0^{(t)}, \beta_1^{(t)}), (\beta_0^{(t+1)}, \beta_1^{(t+1)}))$ is the distance in two dimensional space between the values of (β_0, β_1) at iteration t and $t+1$. We use the euclidean distance, but alternative distance measures may easily be substituted. (see `?dist`)

We set $\epsilon = 1E-9$, choose initial values $(\beta_0, \beta_1) = (0, 0)$ and begin to iterate.

```
dist_type <- 'euclidean' # see ?dist for other options
epsilon <- 1E-9 # convergence criterion
##x_current <- c('b0'=0, 'b1'=0) # inital values
x_current <- c('b0'=0, 'b1'=0) # inital values
x_prev <- c('b0'=Inf, 'b1'=Inf) # no previous values yet!
counter <- 0 # for counting iterations
verbose <- TRUE # do you want to get updates?

while(dist(rbind(x_current, x_prev), method=dist_type) > epsilon){
  counter <- counter+1
  x_prev <- x_current
  x_current <- newton_update(b0=x_current['b0'], b1=x_current['b1'])
  if(verbose==T){
    msg <- paste0('Iteration ', counter, ': b = (',
```

```
                  paste(signif(x_current, 5), collapse=', '),
                  ')')
    cat(msg)
    cat('\n')
    }
  }
```

```
Iteration 1: b = (9.619, -0.14952)
Iteration 2: b = (13.656, -0.21125)
Iteration 3: b = (14.938, -0.2306)
Iteration 4: b = (15.042, -0.23215)
Iteration 5: b = (15.043, -0.23216)
Iteration 6: b = (15.043, -0.23216)
Iteration 7: b = (15.043, -0.23216)
```

I check the maximum likelihood estimates obtained above with R's estimates, computed using the `glm()` (generalized linear model) function.

```
glm_mod <- glm(oring~temp, data=challenger, family=binomial)
glm_mod$coefficients
```

```
(Intercept)        temp
    15.0429     -0.2322
```

b) Solve the same problem using the "Iterative Reweighted Least Squares" algorithm and the Newton-Raphson algorithm to find MLEs of $\widehat{\beta}_0, \widehat{\beta}_1$

The log-likelihood of (β_0, β_1) can be formulated using matrix notation as

$$\ell(\beta) = \boldsymbol{y}^T \boldsymbol{Z}\boldsymbol{\beta} - \boldsymbol{b}^T \mathbf{1}$$

,

where $\mathbf{1}$ is a column vector of ones, $\boldsymbol{y} = (y_1 ... y_n)^T$, $\boldsymbol{b} = (b(\theta_1) ... b(\theta_n))^T$, and $\boldsymbol{Z}$ is the $n\,X\,2$ matrix whose i th row is $[1, x_i]$.

The gradient is

$$\ell'(\boldsymbol{\beta}) = \boldsymbol{Z}^T(\boldsymbol{y} - \boldsymbol{\pi})$$

where $\boldsymbol{\pi}$ is a column vector of the Bernoulli probabilities $\pi_1, ..., \pi_n$. The Hessian is

$$\ell''(\boldsymbol{\beta}) = \frac{d}{d\boldsymbol{\beta}}(\boldsymbol{Z}^T(\boldsymbol{y} - \boldsymbol{\pi})) = -\left(\frac{d\boldsymbol{\pi}}{d\boldsymbol{\beta}}\right)^T \boldsymbol{Z} = -\boldsymbol{Z}^T\boldsymbol{W}\boldsymbol{Z}$$

where $\boldsymbol{W}$ is a diagonal matrix with i th diagonal entry equal to $\pi_i(1 - \pi_i)$.

Newton's update is therefore

$$\boldsymbol{\beta}^{(t+1)} = \boldsymbol{\beta}^{(t)} - \boldsymbol{l}''(\boldsymbol{\beta}^{(t)})^{-1}\boldsymbol{l}'(\boldsymbol{\beta}^{(t)}) = \boldsymbol{\beta}^{(t)} + \left(\boldsymbol{Z}^T\boldsymbol{W}^{(t)}\boldsymbol{Z}\right)^{-1}\left(\boldsymbol{Z}^T(\boldsymbol{y} - \boldsymbol{\pi}^{(t)})\right)$$

where $\boldsymbol{\pi}^{(t)}$ is the value of $\boldsymbol{\pi}$ corresponding to $\boldsymbol{\beta}^{(t)}$, and $\boldsymbol{W}^{(t)}$ is the diagonal weight matrix evaluated at $\boldsymbol{\pi}^{(t)}$.

To find MLEs of $\widehat{\beta}_0$ and $\widehat{\beta}_1$ we designate the vector $\boldsymbol{y}$ to be the response vector in the data and $\boldsymbol{Z}$ to be the $n\,X\,2$ matrix of ones and ages.

```
y <- challenger$oring
x  <- challenger$temp
Z <- cbind(rep(1, nrow(challenger)), challenger$temp)
```

$\boldsymbol{Z}$ will remain unchanged, but for each iteration we will need to recompute the predicted probabilities $\pi_1, ...\pi_n$ using the current values of $\widehat{\beta}_0$ and $\widehat{\beta}_1$. We will also need to compute at each iteration $\boldsymbol{W}$, the diagonal matrix with diagonal entry i equal to $\pi_i(1-\pi_i)$. Below, we define a function to do each of these computations.

```
find_pis <- function(beta, x){
  # beta a vector of parameter estimates, x a vector of independent variable observations
  pis <- exp(beta%*%t(Z))/(1+exp(beta%*%t(Z)))
  return(pis)
}

find_W <- function(pis){
  W <- diag(c(pis*(1-pis)))
  return(W)
}
```

Next we define a function to calculate Newton's update using $\boldsymbol{Z}$, $\boldsymbol{y}$, $\boldsymbol{\beta}$, and the πs and $\boldsymbol{W}$ calculated using the functions above.

```
find_update <- function(beta, Z, W, y, pis){
  numerator <- t(Z)%*%(y-t(pis))
  denominator <- t(Z)%*%W%*%Z
  update <- solve(denominator)%*%numerator
  return(beta+c(update))
}
```

Finally, we must define a stopping criterion so that the algorithm ceases to update at some point and returns values for $\widehat{\beta}_0$ and $\widehat{\beta}_1$. We will stop updating the existing estimates when an iteration takes place that does not change them. We are not concerned with accuracy at more than seven decimal places, so we round the estimates at each iteration to seven decimal places and check at each iteration to see if the updated estimates are different from the previous iteration. This is equivalent to the convergence criteria $D(\boldsymbol{\beta}^{(t+1)}, \boldsymbol{\beta}^{(t)}) < 5E-7$.

We choose starting values for $\widehat{\beta}_0$ and $\widehat{\beta}_1$ to be $(0, 0)$. Then we begin a while loop to update these estimates until convergence occurs.

```
beta <- c(0, 0)
old_beta <- c(NA, NA)
counter <- 0
decimal_places_of_precision <- 7
verbose <- TRUE

while(!identical(beta, old_beta)){
  pis <- find_pis(beta, x)
  W <- find_W(pis)
```

```
  # set old beta equal to current beta before updating
  old_beta <- beta
  # replace beta with updated estimates
  beta <- find_update(beta, Z, W, y, pis)
  # round to the number of decimal places desired
  beta <- round(beta, decimal_places_of_precision)
  counter <- counter+1
  # show us the current value of beta if "verbose" is TRUE
  if(verbose==T){
    msg <- paste0('Iteration ', counter, ': Beta = (', beta[1], ', ', beta[2], ')')
  cat(msg)
  cat('\n')
  }
}
```

```
Iteration 1: Beta = (9.6190476, -0.1495238)
Iteration 2: Beta = (13.6557372, -0.211247)
Iteration 3: Beta = (14.9382896, -0.2306001)
Iteration 4: Beta = (15.0422911, -0.2321537)
Iteration 5: Beta = (15.0429016, -0.2321627)
Iteration 6: Beta = (15.0429016, -0.2321627)
```

Now that the while loop has terminated, the current values of `beta` should be the MLEs of $\widehat{\beta}_0$ and $\widehat{\beta}_1$. I check these against the estimates given using the `glm()` function in R.

```
# see our estimates
print(beta)
```

```
[1] 15.0429 -0.2322
```

```
# see the glm estimates
glm(oring~temp, data=challenger, family=binomial)$coefficients
```

```
(Intercept)        temp
    15.0429     -0.2322
```

Our estimates match those produced by the `glm()` function, so we are confident that we have correctly found the MLEs of $\widehat{\beta}_0$ and $\widehat{\beta}_1$.

c) We are also interested in predicting O-ring failure. Challenger was launched at $31°F$. What is the predicted probability of O-ring damage at $31°F$? How many O-ring failures should be expected at $31°F$? What can you conclude?

At $31°F$ we predict the probability of at least one O-ring failure to be

$$p = \frac{e^{\beta_0+(\beta_1)(31)}}{1+e^{\beta_0+(\beta_1)(31)}} = \frac{e^{15.0429+(-0.2322)(31)}}{1+e^{15.0429+(-0.2322)(31)}} = 0.9996$$

This prediction means that we are almost certain that at least one O-ring will fail at $31°F$.

Problem 3

The elastic net (Zou and Hastie, 2006) is considered to be a compromise between the ridge and lasso penalties. The elastic net can be formulated using the Lagrangian as follows:

$$\widehat{\beta}^{enet} = \underset{\beta}{\operatorname{argmin}} \sum_{i=1}^{n}(y_i - x_1'\beta)^2 + \lambda_1 \sum_{j=1}^{p}|\beta_j| + \lambda_2 \sum_{j=1}^{p}\beta_j^2$$

where $\lambda_1 \geq 0$ and $\lambda_2 \geq 0$. The "credit" data set is discussed in the textbook of James et al., p83. We will fit the elastic net model to the "credit" data set using only the quantitative predictors. Our challenge is to select the appropriate λ_1 and λ_2 before fitting the final model.

a) Write a function in R using the cross-validation approach to find the optimum values of λ_1 and λ_2

We opt to use ten fold cross-validation, but we will write the code so that it may be generalized for k-fold cross validation.

The first task for our ten fold cross-validation is to divide the data into ten folds. We borrow some code from last assignment to accomplish this.

```
k <- 10 # for 10-fold cross-validation
n <- nrow(credit) # sample size
seg_size <- floor(n/k) # approx size of each segment
leftovers <- n%%k # n mod k
seg_assignments <- rep(1:k, each=seg_size) # create segment assignments
if(leftovers > 0){
  seg_assignments <- c(seg_assignments, 1:leftovers) # add "leftover" observations if necessary

}
set.seed(8675309) # set random seed (is it bad that I always use 867-5309?)
seg_assignments <- sample(seg_assignments) # randomly permute segment assignments
```

For each value of k, λ_1, and λ_2 we want to asses the accuracy of the model somehow. We will use as our criterion the cross-validation error

$$CV_k(\lambda_1, \lambda_2) = \frac{1}{n_k}\sum_{i \in F_k}(y_i - \widehat{f}^{-k}_{(\lambda_1,\lambda_2)}(x_i))^2$$

where n_k is the fold size, F_k is the set of observation in fold k (the test set), and $\widehat{f}^{-k}_{(\lambda_1,\lambda_2)}(x)$ is the prediction function from the model fit to all observations *not* in F_k using parameters (λ_1, λ_2).

Next, we must conduct a grid search across many possible values of λ_1 and λ_2. The `glmnet()` function in the `glmnet` package takes as arguments `alpha` and `lambda`, where what we are calling λ_1 is given by $\alpha\lambda$ and what we are calling λ_2 is given by $(1-\alpha)\lambda$, so we actually want to grid search over possible values of `lambda` and `alpha` being passed to the `glmnet()` function. It is important to note that using this parameterization λ is constrained to be larger than 0 and α is constrained to be between 0 and 1.

A brief look at the documentation for the `glmnet()` function (see `?glmnet`) reveals that the argument `alpha` takes a scalar value, but the argument `lambda` may take a vector of many λ values to try. The documentation states

```
Do not supply a single value for lambda ... . Supply instead a decreasing sequence of
lambda values. glmnet relies on its warms starts for speed, and its often faster to fit
a whole path than compute a single fit.
```

So for the sake of speed we will create a vector of (decreasing) `lambda` values to pass to the `glmnet()` function with each call and repeat this for several values of `alpha`, supplied in separate calls. We will do this for each of our k (10) folds, computing the validation errors, $CV_k(\alpha, \lambda)$, as we go.

Since we will pass many `lambda` values to `glmnet()` at a time, but only one `alpha` value at a time to the function, it will simplify things to define a function to extract the validation error, $CV_k(\alpha, \lambda)$, for all of the `lambda`s supplied with one call to the `glmnet()` function.

```
get_val_errs <- function(alpha, lambdas, train_data, train_response, test_data, test_response){
  # fit the elastic net for all lambda values
  fit <- glmnet(train_data, train_response, alpha=alpha, lambda=lambdas)
  # use the models fit to predict test data
  preds <- predict(fit, newx=test_data)
  # calculate the mean squared error in the test data for each value of lambda
  MSEs <- apply(preds, 2, function(x) mean((test_response-x)^2))
  # return a data.frame with alpha, lambdas, and corresponding MSEs
  return(data.frame(alpha=rep(alpha, length(MSEs)), lambda=lambdas, MSE=MSEs))
}
```

Next we define the grid of `alpha` and `lambda` values over which we will search. We have refined this search grid by some trial and error.

```
lambda_vec <- seq(40, 0, by=-1) # lambdas to try for each alpha
alphas <- seq(0, 1, by=0.01) # alphas to try
```

We will search the grid defined above once for each of our k (10) cross-validation folds, so we preallocate an empty list to hold the results for each of our k folds.

```
# preallocate an empty list with a slot for each of the k validation folds
results <- vector(mode='list', length=k)
```

Finally we search over the defined grid of `alpha` and `lambda` values k times, once for each cross-validation fold, and compute the mean squared error of cross-validation at each pair of values.

```
for(i in 1:k){ # k validation folds
  # define training data matrix
  x <- model.matrix(Balance~., subset(credit, seg_assignments!=i))[,-1]
  # and the training response vector
  y <- credit$Balance[seg_assignments!=i]

  # define the test data matrix
  x_test <- model.matrix(Balance~., subset(credit, seg_assignments==i))[,-1]
  # and the test response vector
  y_test <- credit$Balance[seg_assignments==i]

  # succinctly loop over alphas, finding MSE for each lambda as we go
  result_list <- sapply(alphas, get_val_errs, lambdas=lambda_vec, train_data=x, train_response=y, test_
  # convert the list into a data.frame using rbind()
  result_df <- Reduce(rbind, result_list)
```

```
  # assign a field for k
  result_df$k <- i
  # deposit results into preallocated list
  results[[i]] <- result_df
}
```

The object `results` is now a list with k elements and each element is a `data.frame` with fields `k`, `alpha`, `lambda`, and `MSE`. We combine these homogeneous list elements into one `data.frame` and find the average MSE for each set of `alpha` and `lambda` values (averaging the k MSE values calculated for each pair).

```
# average the MSEs from all k folds
results_all <- Reduce(rbind, results)
results_agg <- aggregate(MSE~alpha+lambda, data=results_all, mean)
```

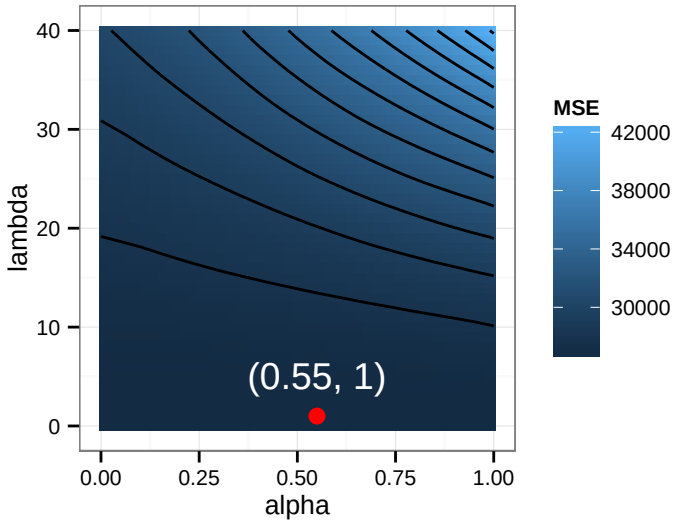
The lowest MSEs appear to coincide with higher values of α and values of λ in the 15-30 range.

```
bestMSE <- results_agg[which.min(results_agg$MSE),]
bestMSE
```

```
    alpha lambda   MSE
157  0.55      1 26666
```

The smallest MSE occurs at $\alpha = 0.55$, $\lambda = 1$.

Since we are searching over two parameters we can visualize the cross-validation errors for each `alpha`, `lambda` pair with a heat map.



b) Repeat the same question as in (a) but using now the *one-standard-error* (1 SE) rule cross validation.

We define the standard error of cross validation as follows:

$$SE(\alpha, \lambda) = \frac{\sqrt{var(CV_1(\alpha, \lambda), ..., CV_k(\alpha, \lambda))}}{\sqrt{k}}$$

For a particular (α, λ) pair, this is the standard deviation of the k computed MSEs $(CV_1(\alpha, \lambda), ..., CV_k(\alpha, \lambda))$ divided by $\sqrt{(k)}$. Recall that each of the MSEs $(CV_1(\alpha, \lambda), ..., CV_k(\alpha, \lambda))$ are stored in the list object `results_all`. From these we can compute the standard error

$$SE(\alpha, \lambda)$$

for each (α, λ) pair.

```
# calculate std dev of MSEs
results_SEs <- aggregate(MSE~alpha+lambda, data=results_all, sd)
# divide by sqrt(k)
results_SEs$MSE <- results_SEs$MSE/sqrt(k)
# rename column for less confusion
names(results_SEs) <- gsub('MSE', 'StdErr', names(results_SEs))
```

The standard error of the best (α, λ) pair is

```
SE_best_model <- results_SEs[results_SEs$alpha==bestMSE$alpha & results_SEs$lambda==bestMSE$lambda, ]

SE_best_model
```

```
    alpha lambda StdErr
157  0.55      1   3156
```

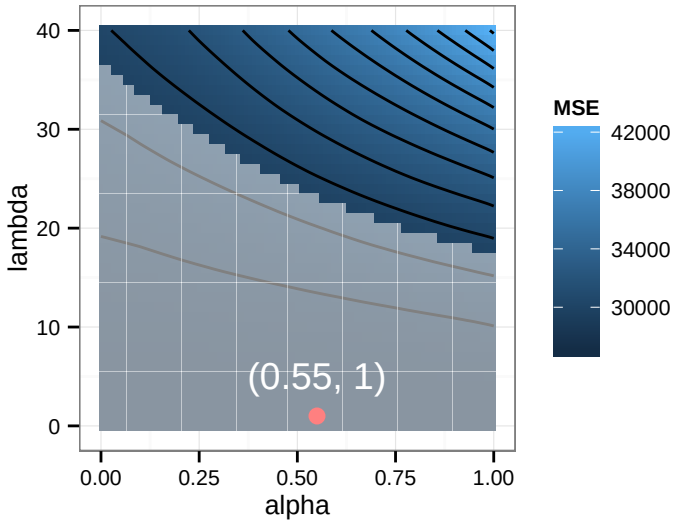
From this we compute the largest tolerable $CV(\alpha, \lambda)$.

```
MSE_max_tolerable <- bestMSE$MSE+SE_best_model$StdErr
```

Next, we retrieve all (α, λ) pairs with a cross-validation error $CV(\alpha, \lambda)$ less than this amount.

```
candidate_lambdas <- results_agg[results_agg$MSE<MSE_max_tolerable, c('alpha', 'lambda')]
```

These candidate (α, λ) pairs are highlighted (in white) on the figure from earlier.



It turns out that there are many candidate models with a cross-validation error $CV(\alpha, \lambda)$ below the threshold $CV(\widehat{\alpha}, \widehat{\lambda}) + SE(\widehat{\alpha}, \widehat{\lambda})$. In fact, we could choose literally any value of α and any λ less than 20. We want to choose the simplest or most regularized model in this candidate model space. Since λ is the penalty parameter for model terms, larger values of λ will result in a simpler model. When $\alpha = 1$, we have the LASSO model, which results in simpler models by setting coefficient values (β's) to 0. From the figure we can see that favoring $\alpha = 1$ limits the largest candidate λ we can choose.

One way to define the "simplest model" is that which has the fewest nonzero parameters. Since some parameter values may be very small, but not quite zero, we will refine this definition slightly so that the "simplest model" is the model in the candidate model space with the fewest parameters larger in absolute value than 0.001. Given this definition, we fit models in the candidate model space on *all* of the data and select the model with the fewest parameter estimates $|\widehat{\beta}_p| \geq 0.001$.

```
# define data matrix
x <- model.matrix(Balance~., credit)[,-1]
# and the response vector
y <- credit$Balance

# define a function
find_nonzero_parameters <- function(alpha, lambda, data, response){
  fit <- glmnet(data, response, alpha=alpha, lambda=c(lambda))
  betas <- fit$beta@x # @ accesses a slot in an s4 object
  num_nonzero <- sum(abs(betas)>0.001)
  return(num_nonzero)
}
# use mapply() to iterate over alpha and lambda together
candidate_lambdas$num_params <- mapply(find_nonzero_parameters,
                                       alpha=candidate_lambdas$alpha,
                                       lambda=candidate_lambdas$lambda,
                                       MoreArgs=list(data=x, response=y))
# how many params does the smallest model have?
min(candidate_lambdas$num_params)
```

```
[1] 4
```

The model(s) with the fewest number of non-zero (approximately) parameters include four parameters. Several models in the candidate model space have four non-zero parameters however.

```
final_candidate_lambdas <- candidate_lambdas[candidate_lambdas$num_params==4,]
final_candidate_lambdas
```

```
     alpha lambda num_params
1616  1.00     15          4
1716  0.99     16          4
1717  1.00     16          4
1815  0.97     17          4
1816  0.98     17          4
1817  0.99     17          4
1818  1.00     17          4
```

Unsurprisingly, these all have large values of α, which corresponds to the LASSO model, which we know tends to favor fewer parameters. Of these models, we will choose the one with the smallest cross-validation error $CV(\alpha, \lambda)$.

```
# merge with MSEs by alpha and lambda
final_candidate_lambdas <- merge(final_candidate_lambdas, results_agg, by=c('alpha', 'lambda'), all.x=T
# best, simplest model using 1-SE rule
final_candidate_lambdas[which.min(final_candidate_lambdas$MSE),]
```

```
  alpha lambda num_params   MSE
5     1     15          4 29017
```

Appendix with R code

```
# Clear working environment
rm(list=ls())
library(glmnet) # for fitting elastic net models
library(RCurl) # for getting data from the web
library(ggplot2) # for plots

# Options for document compilation
knitr::opts_chunk$set(warning=FALSE, message=FALSE, comment=NA, fig.width=4, fig.height=3)

# #### Begin Problem 1 ####
deriv_g <- deriv3(g~4*x*y+(x+y^2)^2, c('x', 'y'), c('x', 'y'))
my_deriv <- deriv_g(x=1, y=1) # test the function deriv_g()
attr(my_deriv, 'gradient') # checks out
attr(my_deriv, 'hessian') # checks out
newton_update <- function(x, y){
  my_deriv <- deriv_g(x=x, y=y) # evaluate gradient and hessian
  gradient <- matrix(attr(my_deriv, 'gradient'), nrow=2) # extract gradient
  hessian <- matrix(attr(my_deriv, 'hessian'), nrow=2) # extract hessian
  update <- c(x, y) - c(solve(hessian)%*%gradient) # calculate update
  names(update) <- c('x', 'y')
  return(update)
}
dist_type <- 'euclidean' # see ?dist for other options
epsilon <- 1E-9 # convergence criterion
x_current <- c('x'=1, 'y'=1) # inital values
x_prev <- c('x'=Inf, 'y'=Inf) # no previous values yet!
counter <- 0 # for counting iterations
verbose <- TRUE # do you want to get updates?

while(dist(rbind(x_current, x_prev), method=dist_type) > epsilon){
  counter <- counter+1
  x_prev <- x_current
  x_current <- newton_update(x=x_current['x'], y=x_current['y'])
  if(verbose==T){
    msg <- paste0('Iteration ', counter, ': x = (', x_current[1], ', ', x_current[2], ')')
    cat(msg)
    cat('\n')
    }
  }
gsection = function(ftn, x.l, x.r, x.m, tol = 1e-9) {
  # applies the golden-section algorithm to minimize ftn
```

```
  # we assume that ftn is a function of a single variable
  # and that x.l < x.m < x.r and ftn(x.l), ftn(x.r) >= ftn(x.m)

  # the algorithm iteratively refines x.l, x.r, and x.m and terminates
  # when x.r - x.l <= tol, then returns x.m

  # golden ratio plus one
  gr1 = 1 + (1 + sqrt(5))/2
  # successively refine x.l, x.r, and x.m
  f.l = ftn(x.l)
  f.r = ftn(x.r)
  f.m = ftn(x.m)
  while ((x.r - x.l) > tol) {
    if ((x.r - x.m) > (x.m - x.l)) {
      y = x.m + (x.r - x.m)/gr1
      f.y = ftn(y)
      if (f.y <= f.m) {
        x.l = x.m
        f.l = f.m
        x.m = y
        f.m = f.y
        } else {
          x.r = y
          f.r = f.y
          }
      } else {
        y = x.m - (x.m - x.l)/gr1
        f.y = ftn(y)
        if (f.y <= f.m) {
          x.r = x.m
          f.r = f.m
          x.m = y
          f.m = f.y
          } else {
            x.l = y
            f.l = f.y
            }
        }
    }
  return(x.m)
  }
f <- function(x) {
  y <- 4*x[1]*x[2]+(x[1]+x[2]^2)^2
  return(y)
  }
gradf<- function (x)
  {##Calculate First Derivative of Function to x[1]  (f1)
  f1<-4*x[2]+2*(x[1] + x[2]^2)
  ##Calculate First Derivative of Function to x[2]  (f2)
  f2<-4 * x[1] + 2 * (2 * x[2] * (x[1] + x[2]^2))
  return(c(f1, f2))
  }
line.search <- function(f, x, gradf, tol = 1e-9, a.max = 100) {
```

```
  # x and gradf are vectors of length d
  # g(a) =f(x +a*gradf) hasa local minumum at a,
  # within a tolerance
  # if no local minimum is found then we use 0 or a.max for a
  # the value returned is x + a*y
  if (sum(abs(gradf)) == 0) return(x) # g(a) constant
  g <- function(a) return(f(x - a*gradf))

  # find a.l < a.m < a.r such that
  # g(a.m) >=g(a.l)  and g(a.m) >= g(a.r)
  # a.l
  a.l <- 0
  g.l <- g(a.l)
  # a.m
  a.m <- 1
  g.m <- g(a.m)
  while ((g.m > g.l) & (a.m > tol)) {
    a.m <- a.m/2
    g.m <- g(a.m)
    }
  # if a suitable a.m was not found then use 0 for a
  if ((a.m <= tol) & (g.m >= g.l)) return(x)
  # a.r
  a.r <- 2*a.m
  g.r <- g(a.r)
  while ((g.m >= g.r) & (a.r < a.max)) {
    a.m <- a.r
    g.m <- g.r
    a.r <- 2*a.m
    g.r <- g(a.r)
    }
  # if a suitable a.r was not found then use a.max for a
  if ((a.r >= a.max) & (g.m > g.r)) return(x - a.max*gradf)
  # apply golden-section algorithm to g to find a
  a <- gsection(g, a.l, a.r, a.m)
  return(x - a*gradf)
  }
descent <- function(f,gradf, x0, tol = 1e-9, n.max = 100) {
  # steepest descent algorithm
  # find a local minimum of f starting at x0
  # function gradf is the gradient of f
  x <- x0
  x.old <- x
  x <- line.search(f, x, gradf(x))
  n <- 1
  while (f(x.old)-(f(x)> tol) & (n < n.max)) {
    x.old <- x
    x <- line.search(f, x, gradf(x))
    n <- n + 1
    }
  return(x)
  }
descent(f,gradf,c(1,0) )
```

```r
# #### Begin Problem 2 ####
# load challenger data
library(mcsm)
data(challenger)
my_deriv_func <- deriv3(l~y*log(1/(1+exp(-b0-b1*x)))+(1-y)*log(1/(1+exp(b0+b1*x))), namevec=c('b0', 'b1
deriv_all <- function(b0, b1, deriv_func){
  # create gradient and hessian object for each (x, y) pair
  deriv_list <- mapply(deriv_func,
                       x=challenger$temp, y=challenger$oring,
                       MoreArgs=list(b0=b0, b1=b1),
                       SIMPLIFY=FALSE
                       )
  # extract all gradient components
  gradient_list <- lapply(deriv_list, function(g) matrix(attr(g, 'gradient'), nrow=2))
  # and sum them to get overall gradient at (b0, b1)
  gradient <- Reduce('+', gradient_list)
  # extract all hessian components
  hessian_list <- lapply(deriv_list, function(g) matrix(attr(g, 'hessian'), nrow=2))
  # and sum them to get overall hessian at (b0, b1)
  hessian <- Reduce('+', hessian_list)
  return(list('gradient'=gradient, 'hessian'=hessian))
  }
deriv_all(b0=0.5, b1=0, deriv_func=my_deriv_func)
newton_update <- function(b0, b1){
  # get gradient and hessian
  my_deriv <- deriv_all(b0=b0, b1=b1, deriv_func=my_deriv_func)
  # calculate update
  update <- c(b0, b1) - c(solve(my_deriv$hessian)%*%my_deriv$gradient)
  # retun update
  names(update) <- c('b0', 'b1')
  return(update)
}
dist_type <- 'euclidean' # see ?dist for other options
epsilon <- 1E-9 # convergence criterion
##x_current <- c('b0'=0, 'b1'=0) # inital values
x_current <- c('b0'=0, 'b1'=0) # inital values
x_prev <- c('b0'=Inf, 'b1'=Inf) # no previous values yet!
counter <- 0 # for counting iterations
verbose <- TRUE # do you want to get updates?

while(dist(rbind(x_current, x_prev), method=dist_type) > epsilon){
  counter <- counter+1
  x_prev <- x_current
  x_current <- newton_update(b0=x_current['b0'], b1=x_current['b1'])
  if(verbose==T){
    msg <- paste0('Iteration ', counter, ': b = (',
                  paste(signif(x_current, 5), collapse=', '),
                  ')')
    cat(msg)
    cat('\n')
    }
  }
glm_mod <- glm(oring~temp, data=challenger, family=binomial)
```

```
glm_mod$coefficients
y <- challenger$oring
x  <- challenger$temp
Z <- cbind(rep(1, nrow(challenger)), challenger$temp)

find_pis <- function(beta, x){
  # beta a vector of parameter estimates, x a vector of independent variable observations
  pis <- exp(beta%*%t(Z))/(1+exp(beta%*%t(Z)))
  return(pis)
}

find_W <- function(pis){
  W <- diag(c(pis*(1-pis)))
  return(W)
}
find_update <- function(beta, Z, W, y, pis){
  numerator <- t(Z)%*%(y-t(pis))
  denominator <- t(Z)%*%W%*%Z
  update <- solve(denominator)%*%numerator
  return(beta+c(update))
}
beta <- c(0, 0)
old_beta <- c(NA, NA)
counter <- 0
decimal_places_of_precision <- 7
verbose <- TRUE

while(!identical(beta, old_beta)){
  pis <- find_pis(beta, x)
  W <- find_W(pis)
  # set old beta equal to current beta before updating
  old_beta <- beta
  # replace beta with updated estimates
  beta <- find_update(beta, Z, W, y, pis)
  # round to the number of decimal places desired
  beta <- round(beta, decimal_places_of_precision)
  counter <- counter+1
  # show us the current value of beta if "verbose" is TRUE
  if(verbose==T){
    msg <- paste0('Iteration ', counter, ': Beta = (', beta[1], ', ', beta[2], ')')
  cat(msg)
  cat('\n')
  }
}
# see our estimates
print(beta)
# see the glm estimates
glm(oring~temp, data=challenger, family=binomial)$coefficients

# #### Begin Problem 3 ####
myfile <- getURL("http://www-bcf.usc.edu/~gareth/ISL/Credit.csv") # grab the content of this page
credit <- read.csv(textConnection(myfile), header=T) # read the page content as a .csv
credit$X <- NULL # get rid of row ID column
```

```r
# find only the column with a numeric class (or integer)
num_cols <- names(credit)[sapply(credit, class)%in%c('numeric', 'integer')]
# now subset the data set to only take the numeric columns
credit <- credit[, num_cols]
k <- 10 # for 10-fold cross-validation
n <- nrow(credit) # sample size
seg_size <- floor(n/k) # approx size of each segment
leftovers <- n%%k # n mod k
seg_assignments <- rep(1:k, each=seg_size) # create segment assignments
if(leftovers > 0){
  seg_assignments <- c(seg_assignments, 1:leftovers) # add "leftover" observations if necessary

}
set.seed(8675309) # set random seed (is it bad that I always use 867-5309?)
seg_assignments <- sample(seg_assignments) # randomly permute segment assignments
get_val_errs <- function(alpha, lambdas, train_data, train_response, test_data, test_response){
  # fit the elastic net for all lambda values
  fit <- glmnet(train_data, train_response, alpha=alpha, lambda=lambdas)
  # use the models fit to predict test data
  preds <- predict(fit, newx=test_data)
  # calculate the mean squared error in the test data for each value of lambda
  MSEs <- apply(preds, 2, function(x) mean((test_response-x)^2))
  # return a data.frame with alpha, lambdas, and corresponding MSEs
  return(data.frame(alpha=rep(alpha, length(MSEs)), lambda=lambdas, MSE=MSEs))
}
lambda_vec <- seq(40, 0, by=-1) # lambdas to try for each alpha
alphas <- seq(0, 1, by=0.01) # alphas to try
# preallocate an empty list with a slot for each of the k validation folds
results <- vector(mode='list', length=k)
for(i in 1:k){ # k validation folds
  # define training data matrix
  x <- model.matrix(Balance~., subset(credit, seg_assignments!=i))[,-1]
  # and the training response vector
  y <- credit$Balance[seg_assignments!=i]

  # define the test data matrix
  x_test <- model.matrix(Balance~., subset(credit, seg_assignments==i))[,-1]
  # and the test response vector
  y_test <- credit$Balance[seg_assignments==i]

  # succinctly loop over alphas, finding MSE for each lambda as we go
  result_list <- sapply(alphas, get_val_errs, lambdas=lambda_vec, train_data=x, train_response=y, test_
  # convert the list into a data.frame using rbind()
  result_df <- Reduce(rbind, result_list)
  # assign a field for k
  result_df$k <- i
  # deposit results into preallocated list
  results[[i]] <- result_df
}
# average the MSEs from all k folds
results_all <- Reduce(rbind, results)
results_agg <- aggregate(MSE~alpha+lambda, data=results_all, mean)
bestMSE <- results_agg[which.min(results_agg$MSE),]
```

```
bestMSE
# plot a heat map of the MSEs by alpha and lambda
ggplot(results_agg, aes(x=alpha, y=lambda, z=MSE))+
  geom_tile(aes(fill=MSE))+
  stat_contour(bins=12)+
  geom_point(data=bestMSE, color='red', size=3)+
  geom_text(data=bestMSE, aes(label=paste0('(', alpha, ', ', lambda, ')')), color='white', vjust=-1)+
  theme_bw(base_size=10)
# calculate std dev of MSEs
results_SEs <- aggregate(MSE~alpha+lambda, data=results_all, sd)
# divide by sqrt(k)
results_SEs$MSE <- results_SEs$MSE/sqrt(k)
# rename column for less confusion
names(results_SEs) <- gsub('MSE', 'StdErr', names(results_SEs))
SE_best_model <- results_SEs[results_SEs$alpha==bestMSE$alpha & results_SEs$lambda==bestMSE$lambda, ]

SE_best_model
MSE_max_tolerable <- bestMSE$MSE+SE_best_model$StdErr
candidate_lambdas <- results_agg[results_agg$MSE<MSE_max_tolerable, c('alpha', 'lambda')]
# plot a heat map of the MSEs by alpha and lambda
ggplot(results_agg, aes(x=alpha, y=lambda, z=MSE))+
  geom_tile(aes(fill=MSE))+
  stat_contour(bins=12)+
  geom_point(data=bestMSE, color='red', size=3)+
  geom_text(data=bestMSE, aes(label=paste0('(', alpha, ', ', lambda, ')')), color='white', vjust=-1)+
  geom_tile(data=candidate_lambdas, aes(fill=NULL, z=NULL), fill='white', alpha=0.5)+
  theme_bw(base_size=10)
# define data matrix
x <- model.matrix(Balance~., credit)[,-1]
# and the response vector
y <- credit$Balance

# define a function
find_nonzero_parameters <- function(alpha, lambda, data, response){
  fit <- glmnet(data, response, alpha=alpha, lambda=c(lambda))
  betas <- fit$beta@x # @ accesses a slot in an s4 object
  num_nonzero <- sum(abs(betas)>0.001)
  return(num_nonzero)
}
# use mapply() to iterate over alpha and lambda together
candidate_lambdas$num_params <- mapply(find_nonzero_parameters,
                                       alpha=candidate_lambdas$alpha,
                                       lambda=candidate_lambdas$lambda,
                                       MoreArgs=list(data=x, response=y))
# how many params does the smallest model have?
min(candidate_lambdas$num_params)
final_candidate_lambdas <- candidate_lambdas[candidate_lambdas$num_params==4,]
final_candidate_lambdas
# merge with MSEs by alpha and lambda
final_candidate_lambdas <- merge(final_candidate_lambdas, results_agg, by=c('alpha', 'lambda'), all.x=T
# best, simplest model using 1-SE rule
final_candidate_lambdas[which.min(final_candidate_lambdas$MSE),]
```