Numerical Optimization: first- and second-order methods

Code

Ruoqing Zhu

Last Updated: August 20, 2024

Second-order Methods

Newton’s Method

Now, let’s discuss several specific methods. One of the oldest one is **Newton’s method.** This is motivated form a quadratic approximation (essentially second order Taylor expansion) at a current point $\mathbf{x}$,

$$f(\mathbf{x}^*) \approx f(\mathbf{x}) + \nabla f(\mathbf{x})^{\mathrm{T}}(\mathbf{x}^* - \mathbf{x}) + \frac{1}{2}(\mathbf{x}^* - \mathbf{x})^{\mathrm{T}}\mathbf{H}(\mathbf{x})(\mathbf{x}^* - \mathbf{x})$$

Our goal is to find a new stationary point $\mathbf{x}^*$ such that $\nabla f(\mathbf{x}^*) = 0$. By taking derivative of the above equation on both sides, with respect to $\mathbf{x}^*$, we need

$$0 = \nabla f(\mathbf{x}^*) = 0 + \nabla f(\mathbf{x}) + \mathbf{H}(\mathbf{x})(\mathbf{x}^* - \mathbf{x})$$

which leads to

$$\mathbf{x}^* = \mathbf{x} - \mathbf{H}(\mathbf{x})^{-1}\nabla f(\mathbf{x}).$$

Hence, if we are currently at a point $\mathbf{x}^{(k)}$, we need to calculate the gradient $\nabla f(\mathbf{x}^{(k)})$ and Hessian $\mathbf{H}(\mathbf{x})$ at this point, then move to the new point using $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathbf{H}(\mathbf{x}^{(k)})^{-1}\nabla f(\mathbf{x}^{(k)})$. Some properties and things to concern regarding Newton’s method:

- For numerical stability, we may consider moving in smaller step sizes, this meaning using a $\delta \in (0, 1)$ and move with

  $$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \delta\,\mathbf{H}(\mathbf{x}^{(k)})^{-1}\nabla f(\mathbf{x}^{(k)})$$

  This main concern here is that a local quadratic approximation may not be accurate once we reach a far enough point. Hence its better to be caucious.
- There are also alternatives to take care of the step size. For example, **line search** is frequently used, which will try to find the optimal δ that minimizes the function

  $$f(\mathbf{x}^{(k)} + \delta\mathbf{v})$$

once we determined on a certain direction $\mathbf{v}$, e.g., $\mathbf{v} = \mathbf{H}(\mathbf{x}^{(k)})^{-1} \nabla f(\mathbf{x}^{(k)})$. For more details, please see SMLR book (https://teazrq.github.io/SMLR/optimization-basics.html#newtons-method)

- When you do not have the explicit formula of Hessian and even the gradient, you may **numerically approximate the derivative** using the definition. For example, we could use the definition of derivative

$$\frac{f(\mathbf{x}^{(k)} + \epsilon) - f(\mathbf{x}^{(k)})}{\epsilon}$$

  with ϵ small enough, e.g., 10^{-5}. However, this is very costly for the Hessian matrix if the number of variables is large.

Quasi-Newton Methods

Since the idea of Newton's method is to solve a vector $\mathbf{v}$ such that

$$\mathbf{H}(\mathbf{x}^{(k)})\mathbf{v} = -\nabla f(\mathbf{x}^{(k)}),$$

If $\mathbf{H}$ is difficult to compute, we may use some matrix to substitute it. For example, if we simplify use the identity matrix $\mathbf{I}$, then this reduces to a first-order method to be introduced later. However, if we can get a good approximation, we can still solve this linear system and get to a better point. Then the question is how to obtain such matrix in a computationally inexpensive way. The Broyden–Fletcher–Goldfarb–Shanno (BFSG) algorithm is such an approach by iteratively updating its (inverse) estimation. For details, please see the SMLR book (https://teazrq.github.io/SMLR/optimization-basics.html#quasi-newton-methods). We have already used the BFGS method previously in the `optim()` example.

First-order Methods

Gradient Descent

When simply using $\mathbf{H} = \mathbf{I}$, we update

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \delta \nabla f(\mathbf{x}^{(k)}).$$

However, it is then crucial to figure out the step size δ. A step size too large may not even converge at all, however, a step size too small will take too many iterations to converge. Alternatively, line search could be used.

Gradient Descent Example: Linear Regression

We use linear regression as an example. The objective function for linear regression is:

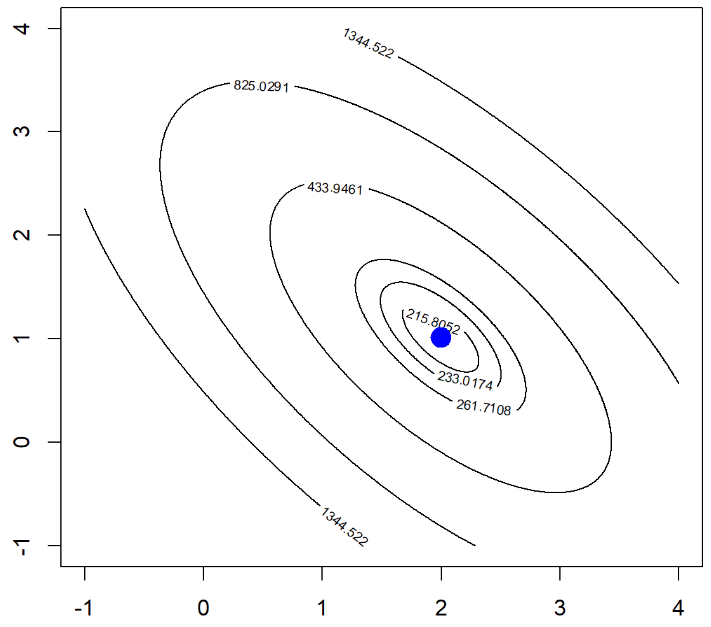
$$\ell(\boldsymbol{\beta}) = \frac{1}{2n} ||\mathbf{y} - \mathbf{X}\boldsymbol{\beta}||^2$$

with solution

$$\widehat{\boldsymbol{\beta}} = \left(\mathbf{X}^\text{T}\mathbf{X}\right)^{-1} \mathbf{X}^\text{T}\mathbf{y}$$

Hide

```r
par(bg="transparent")
library(MASS)
set.seed(3)
n = 200

# create some data with linear model
X = mvrnorm(n, c(0, 0), matrix(c(1,0.7, 0.7, 1), 2,2))
y = rnorm(n, mean = 2*X[,1] + X[,2])

beta1 <- seq(-1, 4, 0.005)
beta2 <- seq(-1, 4, 0.005)
allbeta <- data.matrix(expand.grid(beta1, beta2))
rss <- matrix(apply(allbeta, 1, function(b, X, y) sum((y - X %*% b)^2), X, y),
              length(beta1), length(beta2))

# quantile levels for drawing contour
quanlvl = c(0.01, 0.025, 0.05, 0.2, 0.5, 0.75)

# plot the contour
contour(beta1, beta2, rss, levels = quantile(rss, quanlvl))
box()

# the truth
b = solve(t(X) %*% X) %*% t(X) %*% y
points(b[1], b[2], pch = 19, col = "blue", cex = 2)
```



This is a contour plot, with its lowest point at the linear regression parameter estimate $\widehat{\boldsymbol{\beta}}$. We use an optimization approach to solve this problem. By taking the derivative with respect to $\boldsymbol{\beta}$, we have the gradient

$$\frac{\partial \ell(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -\frac{1}{n}\sum_{i=1}^{n}(y_i - x_i^{\mathrm{T}}\boldsymbol{\beta})x_i\,.$$

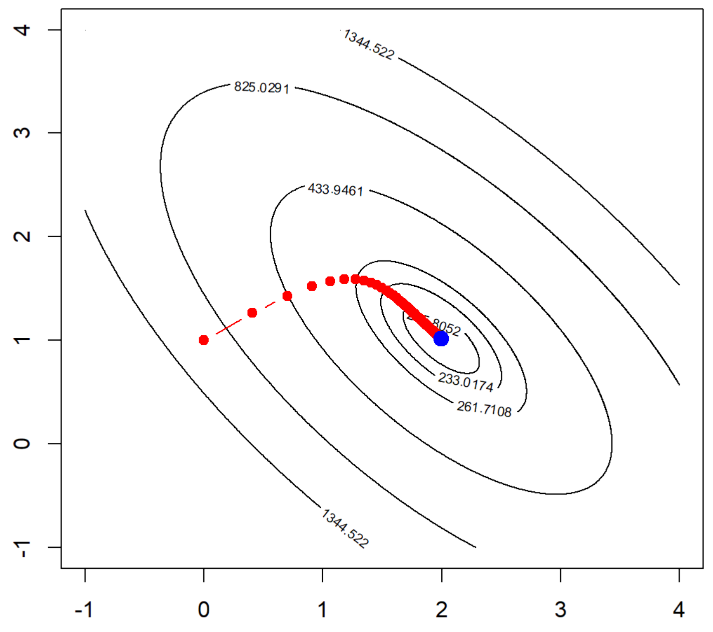
To perform the optimization, we will first set an initial beta value, say $\boldsymbol{\beta} = \mathbf{0}$ for all entries, then proceed with the update

$$\begin{aligned}\boldsymbol{\beta}^{\text{new}} =& \boldsymbol{\beta}^{\text{old}} - \frac{\partial \ell(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} \times \delta.\\ =& \boldsymbol{\beta}^{\text{old}} + \delta \frac{1}{n}\sum_{i=1}^{n}(y_i - x_i^{\mathrm{T}}\boldsymbol{\beta})x_i\,.\end{aligned}$$

Let's set $\delta = 0.2$ for now. The following function performs gradient descent.

Hide

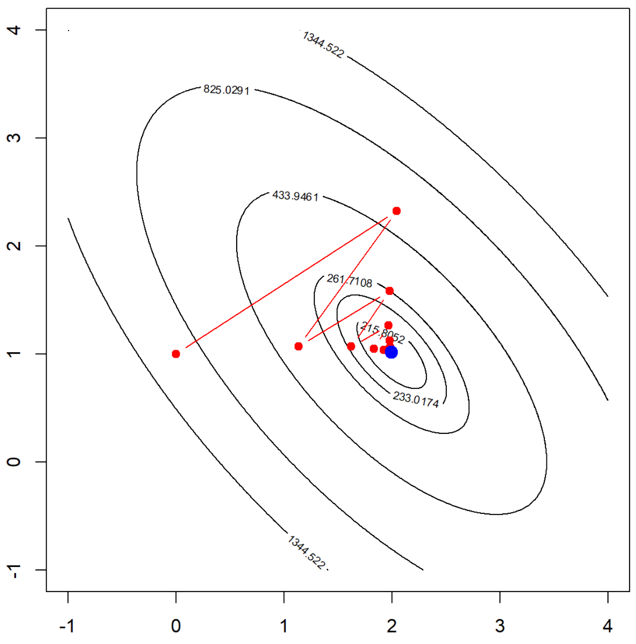
```
# gradient descent function, which also record the path
mylm_g <- function(x, y,
                   b0 = rep(0, ncol(x)), # initial value
                   delta = 0.2, # step size
                   epsilon = 1e-6, #stopping rule
                   maxitr = 5000) # maximum iterations
{
  if (!is.matrix(x)) stop("x must be a matrix")
  if (!is.vector(y)) stop("y must be a vector")
  if (nrow(x) != length(y)) stop("number of observations different")

  # initialize beta values
  allb = matrix(b0, 1, length(b0))

  # iterative update
  for (k in 1:maxitr)
  {
    # the new beta value
    b1 = b0 + t(x) %*% (y - x %*% b0) * delta / length(y)

    # record the new beta
    allb = rbind(allb, as.vector(b1))

    # stopping rule
    if (max(abs(b0 - b1)) < epsilon)
      break;

    # reset beta0
    b0 = b1
  }

  if (k == maxitr) cat("maximum iteration reached\n")
  return(list("allb" = allb, "beta" = b1))
}

# fit the model
mybeta = mylm_g(X, y, b0 = c(0, 1))

par(bg="transparent")
contour(beta1, beta2, rss, levels = quantile(rss, quanlvl))
points(mybeta$allb[,1], mybeta$allb[,2], type = "b", col = "red", pch = 19)
points(b[1], b[2], pch = 19, col = "blue", cex = 1.5)
box()
```

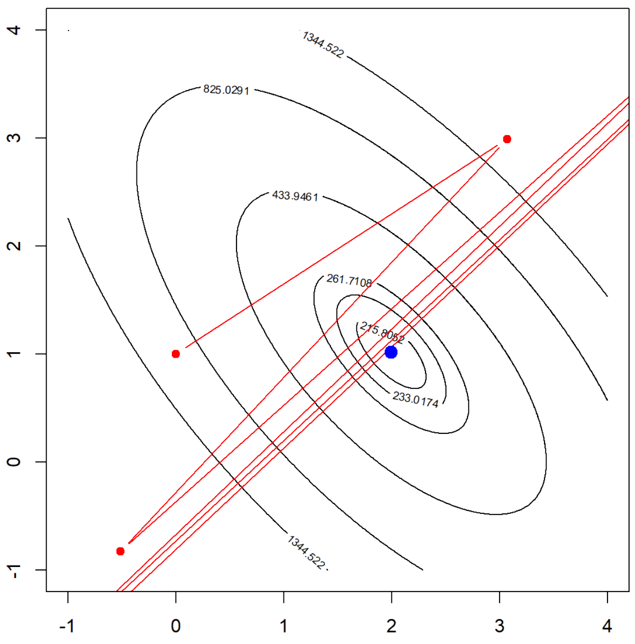


The descent path is very smooth because we choose a very small step size. However, if the step size is too large, we may observe unstable results or even unable to converge. For example, if We set $\delta = 1$ or $\delta = 1.5$.

Hide

```
  par(mfrow=c(1,2))
  par(mar=c(2,2,2,2))
  par(bg="transparent")

  # fit the model with a larger step size
  mybeta = mylm_g(X, y, b0 = c(0, 1), delta = 1)

  contour(beta1, beta2, rss, levels = quantile(rss, quanlvl))
  points(mybeta$allb[,1], mybeta$allb[,2], type = "b", col = "red", pch = 19)
  points(b[1], b[2], pch = 19, col = "blue", cex = 1.5)
  box()

  # and even larger
  mybeta = mylm_g(X, y, b0 = c(0, 1), delta = 1.5, maxitr = 6)
## maximum iteration reached

  contour(beta1, beta2, rss, levels = quantile(rss, quanlvl))
  points(mybeta$allb[,1], mybeta$allb[,2], type = "b", col = "red", pch = 19)
  points(b[1], b[2], pch = 19, col = "blue", cex = 1.5)
  box()
```





The `optim()` function, the third time

When we only supply the `optim()` function the objective function definition `fn =`, it will internally numerically approximate the gradient. Sometimes this is not preferred due to computational reasons. And when you do have the theoretical gradient, it is likely to be more accurate than numerically approximated values. Hence, as long as the gradient itself does not cost too much compute, we may take advantage of it. We could supply the `optim()` function using the gradient `gr =`. This is the example we used previously:

Hide

```
# generate data from a simple linear model
set.seed(20)
n = 150
X <- cbind(1, rnorm(n))
y <- X %*% c(0.5, 1) + rnorm(n)
```

Please note that the calculation of the gradient is in a matrix form, note that $\mathbf{X}$ is the $n \times p$ design matrix. This should look familiar to you since it is the normal equation used in solving the linear regression analytic solution.

$$-\frac{1}{n}\sum_{i=1}^{n}(y_i - x_i^{\mathrm{T}}\boldsymbol{\beta})x_i = -\frac{1}{n}\mathbf{X}^{\mathrm{T}}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})$$

Hide

```
  # calculate the residual sum of squares for a grid of beta values
  rss <- function(b, trainx, trainy) mean((trainy - trainx %*% b)^2)

  # the gradient formula is already provided
  rss_g <- function(b, trainx, trainy) -2*t(trainx) %*% (trainy - trainx %*% b) / n
row(trainx)

  # The solution can be solved by any optimization algorithm
  lm.optim.g <- optim(par = c(2, 2), fn = rss, gr = rss_g, method = "BFGS", trainx
= X, trainy = y)
  lm.optim.g
## $par
## [1] 0.566921 1.016770
##
## $value
## [1] 1.076857
##
## $counts
## function gradient
##        8        5
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Practice question

When fitting linear regressions, outliers could significantly affect the fitting results. However, manually checking and removing outliers can be tricky and time consuming. Some regression methods address this problem by using a more robust loss function. For example, the Huber loss. The idea is to minimize

$$\frac{1}{n}\sum_{i=1}^{n} \ell_\delta(y_i - x_i^\mathrm{T}\boldsymbol{\beta})$$

where the Huber loss function ℓ_δ is defined as

$$\ell_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{if } |a| \leq \delta \\ \delta(|a| - \frac{1}{2}\delta) & \text{o.w.} \end{cases}$$

1. Write a function to calculate the Huber loss, with the argument a, and $\delta = 1$. Compare your results with the ℓ^2 loss of a linear regression. Your plot should be the same as this one (https://en.wikipedia.org/wiki/Huber_loss) from Wikipedia.

Show

2. Use the Huber loss to fit the regression problem with x and y defined previously.

Show

3. The result is not very different from a linear regression since there is no very bad outliers. Let’s use the following data and compare the result with a linear regression.

Show

Constrained Problems

Constrained problems are even more commonly seen in this course, such as Lasso, Ridge and SVM. However, with suitable conditions, they can be transform to a easier, non-constrained problem using Lagrangian. An example is provided at here (https://teazrq.github.io/SMLR/optimization-basics.html#lagrangian-multiplier-for-constrained-problems).