Lecture 17: Numerical Optimization

36-350

22 October 2014

Agenda

- Basics of optimization
- Gradient descent
- Newton's method
- Curve-fitting
- R: `optim`, `nls`

Reading: Recipes 13.1 and 13.2 in *The R Cookbook*

Optional reading: 1.1, 2.1 and 2.2 in *Red Plenty*

Examples of Optimization Problems

- Minimize mean-squared error of regression surface (Gauss, c. 1800)
- Maximize likelihood of distribution (Fisher, c. 1918)
- Maximize output of plywood from given supplies and factories (Kantorovich, 1939)
- Maximize output of tanks from given supplies and factories; minimize number of bombing runs to destroy factory (c. 1939–1945)
- Maximize return of portfolio for given volatility (Markowitz, 1950s)
- Minimize cost of airline flight schedule (Kantorovich...)
- Maximize reproductive fitness of an organism (Maynard Smith)

Optimization Problems

Given an **objective function** $f : \mathcal{D} \mapsto R$, find

$$\theta^* = \text{argmin}_\theta f(\theta)$$

Basics: maximizing f is minimizing $-f$:

$$\text{argmax}_\theta f(\theta) = \text{argmin}_\theta -f(\theta)$$

If h is strictly increasing (e.g., log), then

$$\text{argmin}_\theta f(\theta) = \text{argmin}_\theta h(f(\theta))$$

Considerations

- Approximation: How close can we get to θ^*, and/or $f(\theta^*)$?
- Time complexity: How many computer steps does that take?
  Varies with precision of approximation, niceness of f, size of $\mathcal{D}$, size of data, method...
- Most optimization algorithms use **successive approximation**, so distinguish number of iterations from cost of each iteration

You remember calculus, right?

Suppose x is one dimensional and f is smooth. If x^* is an **interior** minimum / maximum / extremum point

$$\left.\frac{df}{dx}\right|_{x=x^*} = 0$$

If x^* a minimum,

$$\left.\frac{d^2f}{dx^2}\right|_{x=x^*} > 0$$

You remember calculus, right?

This all carries over to multiple dimensions:

At an **interior extremum**,

$$\nabla f(\theta^*) = 0$$

At an **interior minimum**,

$$\nabla^2 f(\theta^*) \geq 0$$

meaning for any vector v,

$$v^T \nabla^2 f(\theta^*) v \geq 0$$

$\nabla^2 f =$ the **Hessian**, **H**

θ might just be a **local** minimum

Gradients and Changes to f

$$f'(x_0) = \left.\frac{df}{dx}\right|_{x=x_0} = \lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0}$$

$$f(x) \approx f(x_0) + (x - x_0) f'(x_0)$$

Locally, the function looks linear; to minimize a linear function, move down the slope

Multivariate version:

$$f(\theta) \approx f(\theta_0) + (\theta - \theta_0) \cdot \nabla f(\theta_0)$$

$\nabla f(\theta_0)$ points in the direction of fastest ascent at θ_0

Gradient Descent

1. Start with initial guess for θ, step-size η
2. While ((not too tired) and (making adequate progress))
   - Find gradient $\nabla f(\theta)$
   - Set $\theta \leftarrow \theta - \eta \nabla f(\theta)$
3. Return final θ as approximate θ^*

Variations: adaptively adjust η to make sure of improvement or search along the gradient direction for minimum

Pros and Cons of Gradient Descent

Pro:

- Moves in direction of greatest immediate improvement
- If η is small enough, gets to a local minimum eventually, and then stops

Cons:

- "small enough" η can be really, really small
- Slowness or zig-zagging if components of ∇f are very different sizes

How much work do we need?

Scaling

Big-O notation:

$$h(x) = O(g(x))$$

means

$$\lim_{x \to \infty} \frac{h(x)}{g(x)} = c$$

for some $c \neq 0$

e.g., $x^2 - 5000x + 123456778 = O(x^2)$

e.g., $e^x/(1 + e^x) = O(1)$

Useful to look at over-all scaling, hiding details

Also done when the limit is $x \to 0$

How Much Work is Gradient Descent?

Pro:

- For nice f, $f(\theta) \leq f(\theta^*) + \epsilon$ in $O(\epsilon^{-2})$ iterations
  - For *very* nice f, only $O(\log \epsilon^{-1})$ iterations
- To get $\nabla f(\theta)$, take p derivatives, $\therefore$ each iteration costs $O(p)$

Con:

- Taking derivatives can slow down as data grows — each iteration might really be $O(np)$

Taylor Series

What if we do a quadratic approximation to f?

$$f(x) \approx f(x_0) + (x - x_0)f'(x_0) + \frac{1}{2}(x - x_0)^2 f''(x_0)$$

Special cases of general idea of Taylor approximation
Simplifies if x_0 is a minimum since then $f'(x_0) = 0$:

$$f(x) \approx f(x_0) + \frac{1}{2}(x - x_0)^2 f''(x_0)$$

Near a minimum, smooth functions look like parabolas
Carries over to the multivariate case:

$$f(\theta) \approx f(\theta_0) + (\theta - \theta_0) \cdot \nabla f(\theta_0) + \frac{1}{2}(\theta - \theta_0)^T \mathbf{H}(\theta_0)(\theta - \theta_0)$$

Minimizing a Quadratic

If we know

$$f(x) = ax^2 + bx + c$$

we minimize exactly:

$$\begin{aligned} 2ax^* + b &= 0 \\ x^* &= \frac{-b}{2a} \end{aligned}$$

If

$$f(x) = \frac{1}{2}a(x - x_0)^2 + b(x - x_0) + c$$

then

$$x^* = x_0 - a^{-1}b$$

Newton's Method

Taylor-expand for the value *at the minimum* θ^*

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

Take gradient, set to zero, solve for θ^*:

$$\begin{aligned} 0 &= \nabla f(\theta) + \mathbf{H}(\theta)(\theta^* - \theta) \\ \theta^* &= \theta - (\mathbf{H}(\theta))^{-1}\nabla f(\theta) \end{aligned}$$

Works *exactly* if f is quadratic
and $\mathbf{H}^{-1}$ exists, etc.

If f isn't quadratic, keep pretending it is until we get close to θ^*, when it will be nearly true

Newton's Method: The Algorithm

1. Start with guess for θ
2. While ((not too tired) and (making adequate progress))
   - Find gradient $\nabla f(\theta)$ and Hessian $\mathbf{H}(\theta)$
   - Set $\theta \leftarrow \theta - \mathbf{H}(\theta)^{-1}\nabla f(\theta)$
3. Return final θ as approximation to θ^*

Like gradient descent, but with inverse Hessian giving the step-size

"This is about how far you can go with that gradient"

Advantages and Disadvantages of Newton's Method

Pros:

- Step-sizes chosen adaptively through 2nd derivatives, much harder to get zig-zagging, over-shooting, etc.
- Also guaranteed to need $O(\epsilon^{-2})$ steps to get within ϵ of optimum
- Only $O(\log\log\epsilon^{-1})$ for very nice functions
- Typically many fewer iterations than gradient descent

Advantages and Disadvantages of Newton's Method

Cons:

- Hopeless if $\mathbf{H}$ doesn't exist or isn't invertible
- Need to take $O(p^2)$ second derivatives *plus* p first derivatives
- Need to solve $\mathbf{H}\theta_{\text{new}} = \mathbf{H}\theta_{\text{old}} - \nabla f(\theta_{\text{old}})$ for θ_{new}
  - inverting $\mathbf{H}$ is $O(p^3)$, but cleverness gives $O(p^2)$ for solving for θ_{new}

Getting Around the Hessian

Want to use the Hessian to improve convergence

Don't want to have to keep computing the Hessian at each step

Approaches:

- Use knowledge of the system to get some approximation to the Hessian, use that instead of taking derivatives ("Fisher scoring")
- Use only diagonal entries (p unmixed 2nd derivatives)
- Use $\mathbf{H}(\theta)$ at initial guess, hope $\mathbf{H}$ changes *very* slowly with θ
- Re-compute $\mathbf{H}(\theta)$ every k steps, $k > 1$
- Fast, approximate updates to the Hessian at each step (BFGS)

Other Methods

- Lots!
- See bonus slides at end for for "Nedler-Mead", a.k.a. "the simplex method", which doesn't need any derivatives
- See bonus slides for the meta-method "coordinate descent"

Curve-Fitting by Optimizing

We have data $(x_1, y_1), (x_2, y_2), \ldots (x_n, y_n)$

We also have possible curves, $r(x; \theta)$

e.g., $r(x) = x \cdot \theta$

e.g., $r(x) = \theta_1 x^{\theta_2}$

e.g., $r(x) = \sum_{j=1}^{q} \theta_j b_j(x)$ for fixed "basis" functions b_j

Curve-Fitting by Optimizing

Least-squares curve fitting:

$$\hat{\theta} = \text{argmin}_\theta \frac{1}{n} \sum_{i=1}^{n} (y_i - r(x_i; \theta))^2$$

"Robust" curve fitting:

$$\hat{\theta} = \text{argmin}_\theta \frac{1}{n} \sum_{i=1}^{n} \psi(y_i - r(x_i; \theta))$$

Optimization in R: optim()

`optim(par, fn, gr, method, control, hessian)`

- `fn`: function to be minimized; mandatory
- `par`: initial parameter guess; mandatory
- `gr`: gradient function; only needed for some methods
- `method`: defaults to a gradient-free method ("Nedler-Mead"), could be BFGS (Newton-ish)
- `control`: optional list of control settings
  - (maximum iterations, scaling, tolerance for convergence, etc.)
- `hessian`: should the final Hessian be returned? default FALSE

Return contains the location (`$par`) and the value (`$val`) of the optimum, diagnostics, possibly `$hessian`

Optimization in R: optim()

```
gmp <- read.table("gmp.dat")
gmp$pop <- gmp$gmp/gmp$pcgmp
library(numDeriv)
mse <- function(theta) { mean((gmp$pcgmp - theta[1]*gmp$pop^theta[2])^2) }
grad.mse <- function(theta) { grad(func=mse,x=theta) }
theta0=c(5000,0.15)
fit1 <- optim(theta0,mse,grad.mse,method="BFGS",hessian=TRUE)
```

fit1: Newton-ish BFGS method

```
fit1[1:3]
```

```
## $par
## [1] 6493.2564    0.1277
##
## $value
## [1] 61853983
##
## $counts
## function gradient
##       63       11
```

fit1: Newton-ish BFGS method

```
fit1[4:6]
```

```
## $convergence
## [1] 0
##
## $message
## NULL
##
## $hessian
##           [,1]      [,2]
## [1,]      52.5 4.422e+06
## [2,] 4422070.4 3.757e+11
```

nls

`optim` is a general-purpose optimizer

So is `nlm` — try them both if one doesn't work

`nls` is for nonlinear least squares

nls

```
nls(formula, data, start, control, [[many other options]])
```

- `formula`: Mathematical expression with response variable, predictor variable(s), and unknown parameter(s)
- `data`: Data frame with variable names matching `formula`
- `start`: Guess at parameters (optional)
- `control`: Like with `optim` (optional)

Returns an `nls` object, with fitted values, prediction methods, etc.

The default optimization is a version of Newton's method

fit2: Fitting the Same Model with nls()

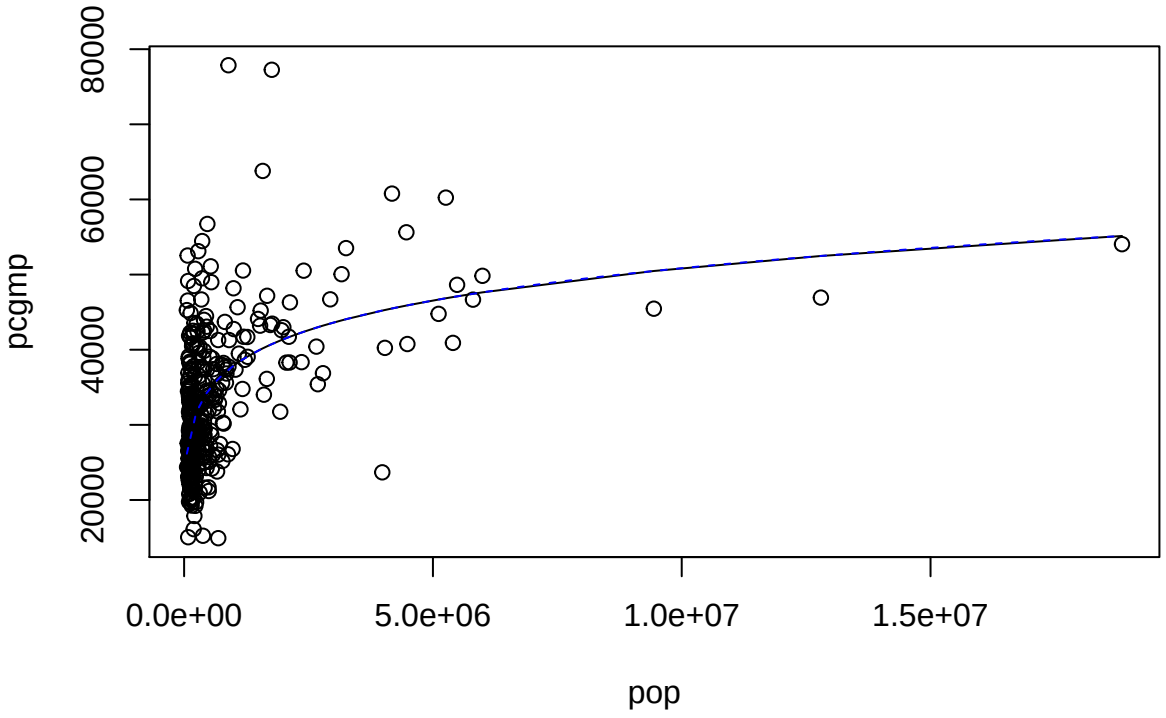
```
fit2 <- nls(pcgmp~y0*pop^a,data=gmp,start=list(y0=5000,a=0.1))
summary(fit2)
```

```
##
## Formula: pcgmp ~ y0 * pop^a
##
## Parameters:
##    Estimate Std. Error t value Pr(>|t|)
## y0 6.49e+03   8.57e+02    7.58  2.9e-13 ***
## a  1.28e-01   1.01e-02   12.61  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```

```
## Residual standard error: 7890 on 364 degrees of freedom
##
## Number of iterations to convergence: 5
## Achieved convergence tolerance: 1.75e-07
```

fit2: Fitting the Same Model with nls()

```
plot(pcgmp~pop,data=gmp)
pop.order <- order(gmp$pop)
lines(gmp$pop[pop.order],fitted(fit2)[pop.order])
curve(fit1$par[1]*x^fit1$par[2],add=TRUE,lty="dashed",col="blue")
```



Summary

1. Trade-offs: complexity of iteration vs. number of iterations vs. precision of approximation
   - Gradient descent: less complex iterations, more guarantees, less adaptive
   - Newton: more complex iterations, but few of them for good functions, more adaptive, less robust
2. Start with pre-built code like `optim` or `nls`, implement your own as needed

Nelder-Mead, a.k.a. the Simplex Method

Try to cage θ^* with a **simplex** of $p+1$ points

Order the trial points, $f(\theta_1) \leq f(\theta_2) \ldots \leq f(\theta_{p+1})$

θ_{p+1} is the worst guess — try to improve it

Center of the not-worst $= \theta_0 = \frac{1}{n}\sum_{i=1}^{n}\theta_i$

Nelder-Mead, a.k.a. the Simplex Method

Try to improve the worst guess θ_{p+1}

1. **Reflection**: Try $\theta_0 - (\theta_{p+1} - \theta_0)$, across the center from θ_{p+1}
   - if it's better than θ_p but not than θ_1, replace the old θ_{p+1} with it
   - **Expansion**: if the reflected point is the new best, try $\theta_0 - 2(\theta_{p+1} - \theta_0)$; replace the old θ_{p+1} with the better of the reflected and the expanded point
2. **Contraction**: If the reflected point is worse that θ_p, try $\theta_0 + \frac{\theta_{p+1}-\theta_0}{2}$; if the contracted value is better, replace θ_{p+1} with it
3. **Reduction**: If all else fails, $\theta_i \leftarrow \frac{\theta_1+\theta_i}{2}$
4. Go back to (1) until we stop improving or run out of time

Making Sense of Nedler-Mead

The Moves:

- Reflection: try the opposite of the worst point
- Expansion: if that really helps, try it some more
- Contraction: see if we overshot when trying the opposite
- Reduction: if all else fails, try making each point more like the best point

Making Sense of Nedler-Mead

Pros:

- Each iteration ≤ 4 values of f, plus sorting
  (and sorting is at most $O(p \log p)$, usually much better)
- No derivatives used, can even work for dis-continuous f

Con: - Can need *many* more iterations than gradient methods

Coordinate Descent

Gradient descent, Newton's method, simplex, etc., adjust all coordinates of θ at once — gets harder as the number of dimensions p grows

Coordinate descent: never do more than 1D optimization

- Start with initial guess θ
- While ((not too tired) and (making adequate progress))

    - For $i \in (1:p)$
        - do 1D optimization over i^{th} coordinate of θ, holding the others fixed
        - Update i^{th} coordinate to this optimal value
- Return final value of θ

Coordinate Descent

Cons:

- Needs a good 1D optimizer
- Can bog down for very tricky functions, especially with lots of interactions among variables

Pros:

- Can be extremely fast and simple