Lecture 13: Optimization

Statistical Computing, 36-350

Monday November 2, 2015

Outline

- Basics of optimization
- Gradient descent
- The importance of convexity
- Newton's method
- Built-in R optimizers: `optim()`, `nls()`
- A peak at advanced topics: sparsity, stochasticity

Optional reading: take a look at the Convex Optimization class notes at http://www.stat.cmu.edu/~ryantibs/convexopt/

Important examples of optimization problems

- Minimize mean squared error of regression surface (Gauss, early 1800s)
- Maximize likelihood of distribution (Fisher, early 1900s)
- 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 (wartime engineers, 1939–1945)
- Maximize return of portfolio for given volatility (Markowitz, 1950s)
- Maximize reproductive fitness of an organism (M. Smith, 1970s and 80s)
- Maximize probability of you clicking on adds that are displayed (Google, 2000s, worth more $$$ than you can imagine)

Optimization problems

Given an **objective function** or **criterion function** $f : \mathcal{D} \to \mathbb{R}$, find

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

Note that maximizing f is the same as minimizing $-f$:

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

And if ϕ is strictly increasing (e.g., log), then

$$\underset{\theta}{\text{argmin}} \; f(\theta) = \underset{\theta}{\text{argmin}} \; \phi(f(\theta))$$

Considerations

In general, not possible to find **exact** minimums (or maximums) of optimization problems, but we will learn algorithms that can achieve **arbitrarily good approximations**, by forming iterative guesses at the minimum (or maximum)

Considerations:

- Accuracy: how close do we need to $f(\theta^*)$, or (better yet) to θ^*?
- Convergence speed: how many iterations does it take to get there? Varies a lot based on the algorithm, and objective f
- Iteration cost: how expensive is it to perform a single iteration? Again, varies a lot based on the algorithm, and on f

You remember calculus, right?

Suppose x is one-dimensional and f is smooth. If x^* is a **local minimum** or **local maximum** of f, then

$$f'(x^*) = 0$$

Is the converse true? (No! Could be a saddle point. Draw yourself an example)

How to tell the difference between local minimums and maximums? Check the second derivative. If x^* is a local minimum, then in addition to the above,

$$f''(x^*) > 0$$

and the opposite sign holds for local maximums

This all carries over to multiple dimensions: if now θ is n-dimensional and f is smooth, then

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

at any local minimum or local maximum, where recall ∇f is the **gradient** of f, which contains the all partial derivatives:

$$\nabla f(\theta) = \begin{pmatrix} \frac{\partial f(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial f(\theta)}{\partial \theta_n} \end{pmatrix}$$

What about the second derivative? First we have to recall the generalization of second derivatives to n-dimensions. This is given by the **Hessian** of f, written $\nabla^2 f$. At a point θ, the Hessian $\nabla^2 f(\theta)$, is a matrix containing all the mixed partial derivatives:

$$[\nabla^2 f(\theta)]_{ij} = \frac{\partial^2 f(\theta)}{\partial \theta_i \partial \theta_j}$$

At a local minimum θ^*, the Hessian is going to be positive definite, meaning

$$v^T \nabla^2 f(\theta^*) v > 0, \quad \text{for all } v$$

This roughly means that the function f is "curved upwards" around θ^*

And the opposite sign will be true for a local maximum (associated property is called negative definite)

Gradients and changes to f

By definition,

$$f'(x_0) = \lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0}$$

so for x close to x_0,

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

That is, locally the function looks linear, so to minimize it, we move down the slope

Similar idea is true in higher dimensions: for x close to x_0,

$$f(\theta) \approx f(\theta_0) + \nabla f(\theta_0)^T(\theta - \theta_0)$$

so f looks linear in a small neighborhood, with $\nabla f(\theta_0)$ pointing in the direction of fastest ascent. So to minimize f, move in direction given by $-\nabla f(\theta_0)$

(Note: here $a^T b = \sum_{i=1}^n a_i b_i$, the inner product or dot product of a and b)

Gradient descent

Simple but fundamental algorithm for minimizing f. Just repeatedly move in the direction of the negative gradient

1. Start with initial guess $\theta^{(0)}$, step size η
2. For $k = 1, 2, 3, \ldots$:
   - Compute the gradient $\nabla f(\theta^{(k-1)})$
   - Check if gradient is close to zero; is so stop, otherwise continue
   - Update $\theta^{(k)} = \theta^{(k-1)} - \eta \nabla f(\theta^{(k-1)})$
3. Return final $\theta^{(k)}$ as approximate solution θ^*

Gradient descent code

From last lecture, simple code to perform gradient descent. Check that it makes sense to you

```
library(numDeriv) # to use the grad() function

grad.descent = function(f, x0, max.iter=200, step.size=0.05,
  stopping.deriv=0.01, ...) {

  n = length(x0)
  xmat = matrix(0,nrow=n,ncol=max.iter)
  xmat[,1] = x0

  for (k in 2:max.iter) {
    # Calculate the gradient
    grad.cur = grad(f,xmat[,k-1],...)

    # Should we stop?
    if (all(abs(grad.cur) < stopping.deriv)) {
      k = k-1; break
```

```
    }

    # Move in the opposite direction of the grad
    xmat[,k] = xmat[,k-1] - step.size * grad.cur
  }

  xmat = xmat[,1:k] # Trim
  return(list(x=xmat[,k], xmat=xmat, k=k))
}
```

Recall our example from last time

```
simpleFun = function(x) {
  return(x[1]^2 + 1/3*x[2]^2)
}

x0 = c(-1.9,-1.9)
gd = grad.descent(simpleFun,x0,step.size=1/3)
gd$x
```

```
## [1] -5.448641e-10 -1.246994e-02
```

```
gd$k
```

```
## [1] 21
```

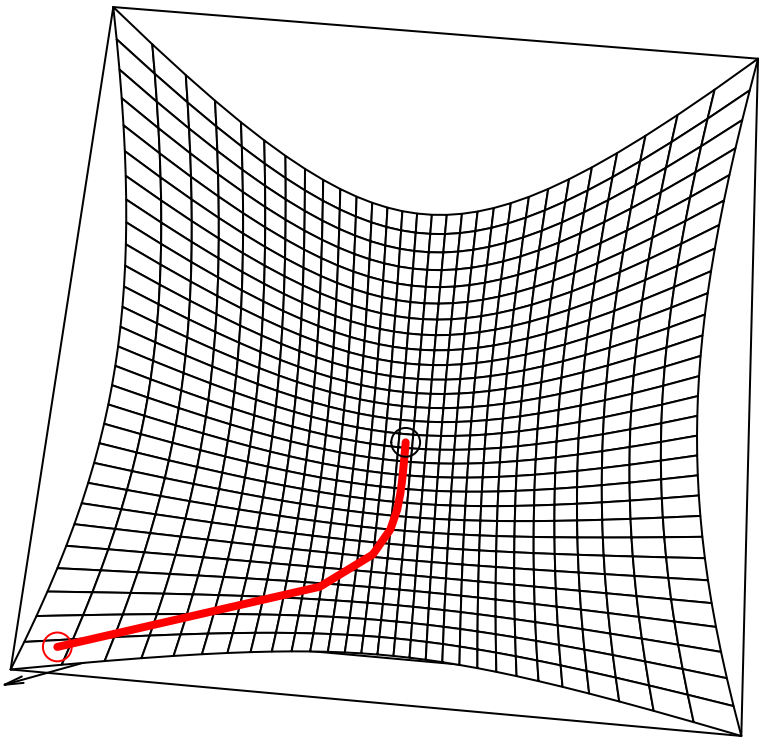
Let's make a nice plot to visualize the results

```
surface = function(f, from.x=0, to.x=1, from.y=0, to.y=1,
                   n.x=30, n.y=30, theta=5, phi=80, ...) {
  # Build the 2d grid
  x.seq = seq(from=from.x,to=to.x,length.out=n.x)
  y.seq = seq(from=from.y,to=to.y,length.out=n.y)
  plot.grid = expand.grid(x.seq,y.seq)
  z.vals = apply(plot.grid,1,f)
  z.mat = matrix(z.vals,nrow=n.x)

  # Plot with the persp function
  orig.mar = par()$mar # Save the original margins
  par(mar=c(1,1,1,1))  # Make the margins small
  r = persp(x.seq,y.seq,z.mat,theta=theta,phi=phi,...)
  par(mar=orig.mar) # Restore the original margins
  invisible(r)
}

# Draw our simple function in 2d, and our gradient descent path on top
r = surface(simpleFun,from.x=-2,to.x=2,from.y=-2,to.y=2,xlab="",ylab="",zlab="")
```

```
points(trans3d(x0[1],x0[2],simpleFun(x0),r),col="red",cex=2)
lines(trans3d(gd$xmat[1,],gd$xmat[2,],apply(gd$xmat,2,simpleFun),r),
      lwd=4,col="red")
points(points(trans3d(gd$x[1],gd$x[2],simpleFun(gd$x),r),
              col="black",cex=2))
```



What's a good step size?

- If the step size is too small, convergence will be slow
- If the step size is too big, the algorithm can diverge!

```
gd = grad.descent(simpleFun,x0,step.size=0.001)
gd$x
```

```
## [1] -1.275649 -1.663865
```

```
gd$k
```

```
## [1] 200
```

```
gd = grad.descent(simpleFun,x0,step.size=2)
gd$x
```

```
## [1] 1.682222e+95 4.575293e-03
```

```
gd$k
```

```
## [1] 200
```

There is some theory about how to choose step sizes, depending on characteristics of f, but more useful is an empirically adaptive step size choice

Backtracking line search

A way to choose step sizes adaptively at each step of gradient descent: **backtracking line search**

Before taking an update as in $\theta - t\nabla f(\theta)$, we check whether this update is going to give us decent progress in minimizing the function, i.e., we check

$$f(\theta - t\nabla f(\theta)) \leq f(\theta) - 0.1t\nabla f(\theta)^T \nabla f(\theta)$$

If this passes, then we perform the update with the step size t; otherwise we shrink t by a factor of 0.9, and try again

Exercise: implement this, and modify the `grad.descent()` so that it has the option to use this, instead of a fixed choice of step size

Pros and cons of gradient descent

Pros:

- Simple and intuitive
- Easy to implement
- Iterations are usually cheap (just compute the gradient)

Cons:

- Can be slow or can zig-zag if components of $\nabla f(\theta)$ are of very different sizes
- In general, can take us a long time to get us close to the minimum
- For most functions, to get to a point $x^{(k)}$ such that $f(x^{(k)}) - f(x^*) \leq \epsilon$, we will need $k \approx 1/\epsilon$ iterations. (For nice functions this will be $k \approx \log(1/\epsilon)$)

Interlude: the role of convexity

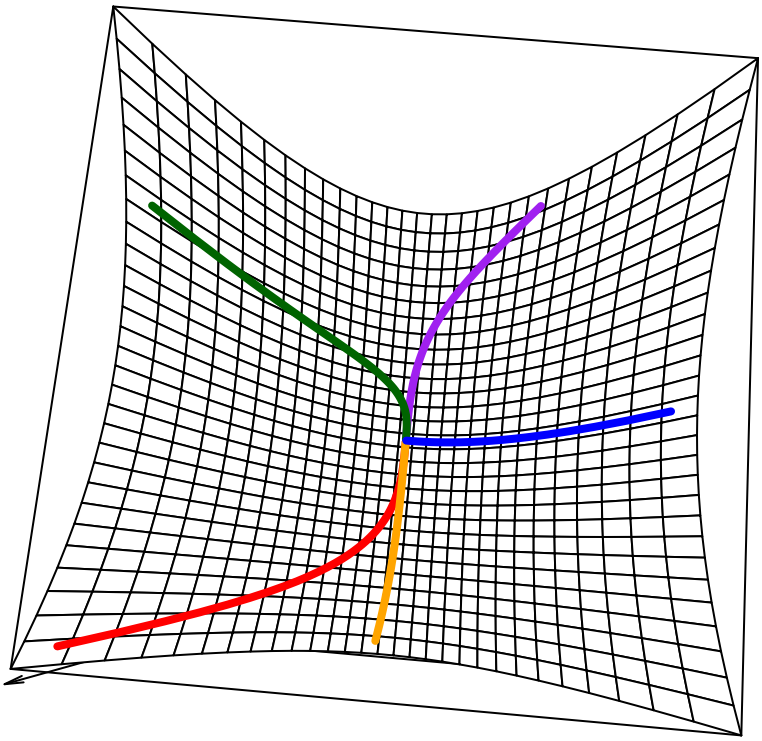
When is it a good idea to go downhill? And is where we end up going to depend on where we started?

Let's return to our simple example, and run a bunch more gradient descents starting from different places, to see where they end up

```
x0 = c(-1.9,-1.9)
gd0 = grad.descent(simpleFun,x0)
x1 = c(-0.1,-1.9)
gd1 = grad.descent(simpleFun,x1)
x2 = c(1,1.9)
```

```
gd2 = grad.descent(simpleFun,x2)
x3 = c(-1.9,1.1)
gd3 = grad.descent(simpleFun,x3)
x4 = c(1.9,0)
gd4 = grad.descent(simpleFun,x4)

# Draw our simple function in 2d, and all gradient descent paths on top
r = surface(simpleFun,from.x=-2,to.x=2,from.y=-2,to.y=2,
            xlab="",ylab="",zlab="")
lines(trans3d(gd0$xmat[1,],gd0$xmat[2,],apply(gd0$xmat,2,simpleFun),r),
      lwd=4,col="red")
lines(trans3d(gd1$xmat[1,],gd1$xmat[2,],apply(gd1$xmat,2,simpleFun),r),
      lwd=4,col="orange")
lines(trans3d(gd2$xmat[1,],gd2$xmat[2,],apply(gd2$xmat,2,simpleFun),r),
      lwd=4,col="purple")
lines(trans3d(gd3$xmat[1,],gd3$xmat[2,],apply(gd3$xmat,2,simpleFun),r),
      lwd=4,col="darkgreen")
lines(trans3d(gd4$xmat[1,],gd4$xmat[2,],apply(gd4$xmat,2,simpleFun),r),
      lwd=4,col="blue")
```



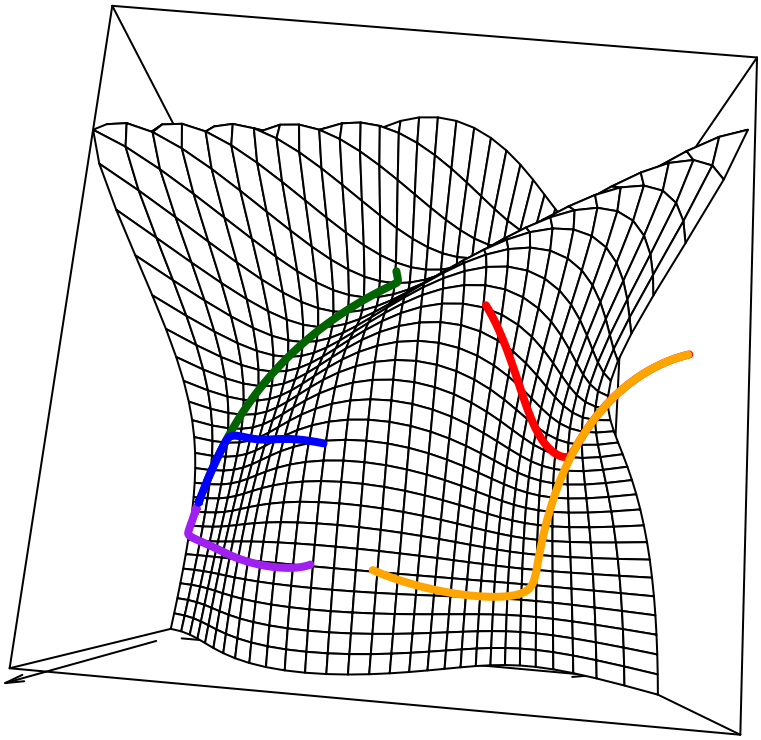
They all end up at the same place! That's a relief

```
complicatedFun = function(x) {
  return((1/2*x[1]^2-1/4*x[2]^2+3)*cos(2*x[1]+1-exp(x[2])))
}

x0 = c(0.5,0.5)
gd0 = grad.descent(complicatedFun,x0,step.size=0.01)
x1 = c(-0.1,-1.3)
gd1 = grad.descent(complicatedFun,x1,step.size=0.01,max.iter=400)
```

```
x2 = c(-0.5,-1.3)
gd2 = grad.descent(complicatedFun,x2,step.size=0.01,max.iter=400)
x3 = c(-0.2,1.4)
gd3 = grad.descent(complicatedFun,x3,step.size=0.01,max.iter=400)
x4 = c(-0.5,-0.5)
gd4 = grad.descent(complicatedFun,x4,step.size=0.01,max.iter=400)

# Draw our simple function in 2d, and all gradient descent paths on top
r = surface(complicatedFun,from.x=-2,to.x=2,from.y=-2,to.y=2,
            xlab="",ylab="",zlab="")
lines(trans3d(gd0$xmat[1,],gd0$xmat[2,],apply(gd0$xmat,2,complicatedFun),r),
      lwd=4,col="red")
lines(trans3d(gd1$xmat[1,],gd1$xmat[2,],apply(gd1$xmat,2,complicatedFun),r),
      lwd=4,col="orange")
lines(trans3d(gd2$xmat[1,],gd2$xmat[2,],apply(gd2$xmat,2,complicatedFun),r),
      lwd=4,col="purple")
lines(trans3d(gd3$xmat[1,],gd3$xmat[2,],apply(gd3$xmat,2,complicatedFun),r),
      lwd=4,col="darkgreen")
lines(trans3d(gd4$xmat[1,],gd4$xmat[2,],apply(gd4$xmat,2,complicatedFun),r),
      lwd=4,col="blue")
```



Oh no! What's going on here? It seems like we end up at drastically different spots depending on where we started

---

The fundamental difference between these two examples is that of **convexity**. A convex function is one that is "bowl shaped", or always "curved upwards"; more precisely, its second derivative matrix is always positive definite

- For convex functions, there are no local minimums; only a global minimum
- Loosely speaking, gradient descent will *always work* for a smooth convex function (provided we choose the step sizes properly)
- But gradient descent can fail emphatically for nonconvex functions; for these functions, moving downhill locally is not always the best idea
- Nonconvex optimization is in general *much, much harder* for this reason

Back to Taylor series approximation

We saw that for x close to x_0, we could write

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

This is called a **first-order Taylor approximation**, using the first derivative

Meanwhile, a **second-order Taylor approximation** uses the second dervative:

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

This is indeed a better (more accurate) approximation for x close to x_0 if we do a quadratic approximation to f

This carries over to the multivariate case:

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

The idea we'll follow now is try to make the quadratic on the right-hand side as small as possible, then Taylor expand again, and then repeat

Minimizing a quadratic

If we know

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

then we can minimize f exactly:

$$x^* = -b/(2a)$$

If instead

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

then

$$x^* = x_0 - b/a$$

Similarly for multidimensional case

Newton's method

More sophisticated than gradient descent, but also more accurate. Repeatedly form and minimize quadratic approximation to f

1. Start with initial guess $\theta^{(0)}$, step size η

2. For $k = 1, 2, 3, \ldots$:
   - Compute the gradient $g = \nabla f(\theta^{(k-1)})$, Hessian $H = \nabla^2 f(x^{(k-1)})$
   - Check if gradient is close to zero; is so stop, otherwise continue
   - Update $\theta^{(k)} = \theta^{(k-1)} - \eta H^{-1} g$
3. Return final $\theta^{(k)}$ as approximate solution θ^*

Pros and cons of Newton's method

Pros:

- Converges a lot faster than gradient descent, because the Hessian and gradient together point in a more fruitful direction than the gradient does alone
- For nice functions, to get $f(x^{(k)}) - f(x^*) \leq \epsilon$, we will need $k \approx \log\log(1/\epsilon)$ iterations
- So, typically many fewer iterations than gradient descent
- For quadratic functions, converges in just one step

Cons:

- In general, a single Newton iteration is *much* more expensive than a single gradient descent update. Requires inverting the Hessian
- If the Hessian isn't invertible (or is close to singular), then we are in trouble

Other approaches

What else can we do? There are an enormous amount of options

- First-order extensions (on top of gradient descent): stochastic gradients, proximal mappings, projections, acceleration
- Second-order extensions (on top of Newton): quasi-Newton (approximate the Hessian), proximal mappings, interior-point methods (handle constraints)
- Dual methods: operator splitting, alternative direction method of multipliers (ADMM)
- Coordinate descent methods, zeroth-order methods, fast stochastic methods, etc., etc.

We will briefly touch on cover stochastic gradients and proximal (thresholding) maps

If you're interested beyond this, then you can browse through the class slides from http://www.stat.cmu.edu/~ryantibs/convexopt/

Optimization in R: `optim()`

Often times it's most convenient just to use built-in R optimization routines. Most functions in R will allow you to access several methods, like `optim()`

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

- `par`: initial parameter guess; mandatory

- `fn`: function to be minimized; mandatory
- `gr`: gradient function; if `NULL`, then `optim()` will approximate it when it needs it
- `method`: defaults to "Nedler-Mead" (a gradient-free method), could also be "CG" (variant of gradient descent), or "BFGS" (variant of Newton)
- `control`: optional list of optimization parameters (maximum iterations, scaling, tolerance for convergence, etc.)
- `hessian`: should the final Hessian be returned? Default is `FALSE`

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

```
gmp = read.table("http://www.stat.cmu.edu/~ryantibs/statcomp/lectures/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
```

```
## $par
## [1] 6493.2563739    0.1276921
##
## $value
## [1] 61853983
##
## $counts
## function gradient
##       63       11
##
## $convergence
## [1] 0
##
## $message
## NULL
##
## $hessian
##               [,1]         [,2]
## [1,]      52.5021      4422071
## [2,] 4422071.3594 375729087979
```

```
eigen(fit1$hessian)$values
```

```
## [1] 3.757291e+11 4.573753e-01
```

Optimization in R: `nls()`

`optim()` is a general-purpose optimizer. `nlm()` is for nonlinear least squares in particular, like the previous GMP example

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

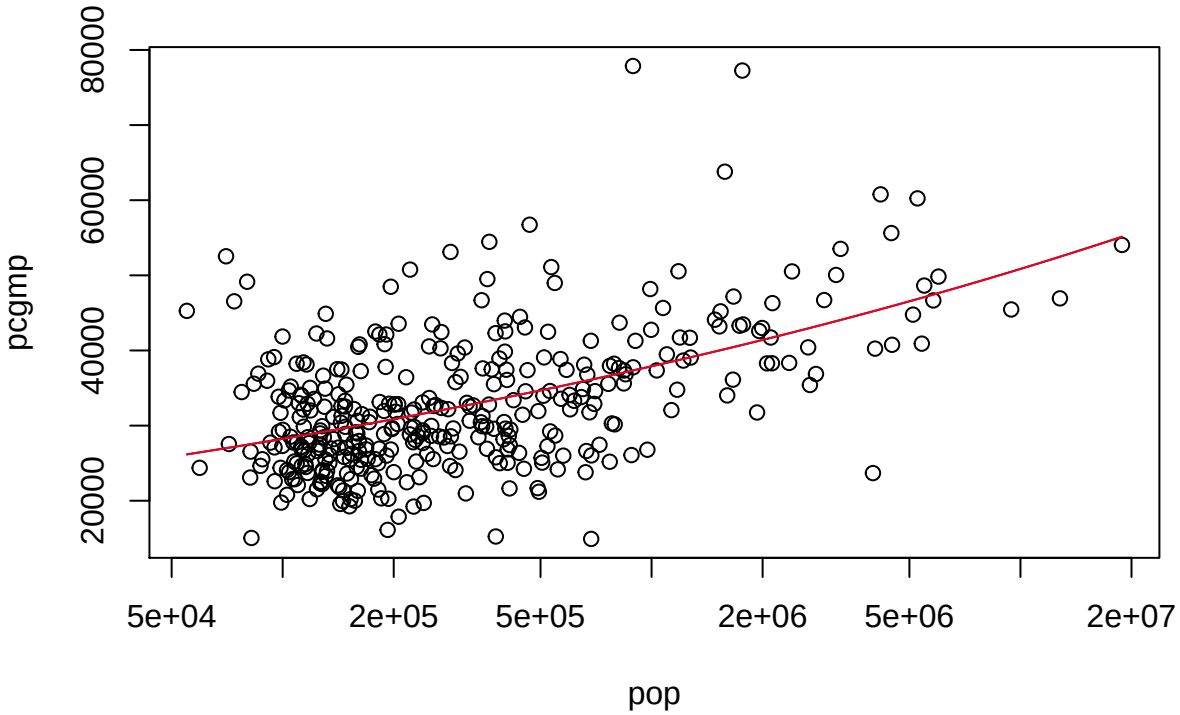
- `formula`: expression with response variable, predictor variable(s), and unknown parameter(s)
- `data`: data frame with variable names matching `formula`
- `start`: initial guess at parameters (optional)
- `control`: optimization parameters like with `optim` (optional)
- `algorithm`: default is a Newton-ish method, many other options available

Returns an `nls` object, which you can pretty much use like an `lm` object (print, plot, get fitted values, make predictions, etc.)

```
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 6.494e+03  8.565e+02   7.582 2.87e-13 ***
## a  1.277e-01  1.012e-02  12.612  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 7886 on 364 degrees of freedom
##
## Number of iterations to convergence: 5
## Achieved convergence tolerance: 1.819e-07
```

```
plot(pcgmp~pop,data=gmp,log="x")
curve(fit1$par[1]*x^fit1$par[2],add=TRUE,col="blue")
curve(coef(fit2)[1]*x^coef(fit2)[2],add=TRUE,col="red")
```



Can't tell any difference between the two! That's a good thing

Two advanced topics: stochastic methods, sparse methods

Suppose we are running a linear regression on p variables, and our criterion function is

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

the standard least squares loss, given responses y_i and predictors x_i, $i = 1, \ldots n$

If n, p are reasonable in size, then we can just solve this using linear algebra, as in

$$\theta^* = (X^T X)^{-1} X^T y$$

where X is the $n \times p$ predictor matrix (with ith row x_i), and y is the $n \times 1$ response vector (with ith component y_i)

But now suppose n something like 100 million. It would be prohibitive to even form $X^T X$, so the above solution is out of the question, and we might try gradient descent. However, gradient descent is also expensive, because the gradient

$$\nabla f(\theta) = \frac{1}{n} \sum_{i=1}^{n} x_i (x_i^T \theta - y_i)$$

requires a sum over n things, which again costs a lot. What to do?

This is where **stochastic methods** come into play. They are widely used for large-scale statistical optimization problems in industry. The idea is simple: each time you want to use it, replace the gradient with a random estimate

That is, instead of updating according to

$$\theta - t\nabla f(\theta) = \theta - \frac{t}{n}\sum_{i=1}^{n} x_i(x_i^T\theta - y_i)$$

updating according to

$$\theta - tg = \theta - \frac{t}{m}\sum_{i\in I} x_i(x_i^T\theta - y_i)$$

Here I is a random subset of $\{1, \ldots n\}$ of size m (where m is much, much smaller than n). In other words, we've replaced the full gradient by

$$g = \frac{1}{m}\sum_{i\in I} x_i(x_i^T\theta - y_i)$$

and unbiased estimate of it, computed over just m data points. This is much cheaper when m is reasonably small

Stochastic methods can save a lot of computation; they generally are rapid to make progress in minimizing the criterion at the start, but then take longer to get to high accuracy solutions

---

Now let's consider another variant: suppose p, the number of predictors, is enormous—say, in the millions itself. For both statistical reasons and optimization reasons, we don't want to optimize over p regression parameter

Statistically: we will get bad estimates (high variance), optimization-wise: it will be very expensive and slow to converge

(You'll learn a lot more about the statistical side in 36-401, and further in 36-402)

Idea: if we have 1 million regression parameters, then it's unlikely that they're all super important. So let's ignore the small ones, basically, by just setting them equal to zero. This is called a **sparse method**, and we can work this directly into gradient descent:

1. Start with initial guess $\theta^{(0)}$, step size η
2. For $k = 1, 2, 3, \ldots$:
    - Compute the gradient $\nabla f(\theta^{(k-1)})$
    - Check if gradient is close to zero; is so stop, otherwise continue
    - Update $\theta^{(k)} = \theta^{(k-1)} - \eta\nabla f(\theta^{(k-1)})$
    - *Correct $\theta^{(k)}$ by thresholding the small components to zero*
3. Return final $\theta^{(k)}$ as approximate solution θ^*

The addendum can be viewed as a particular special case of what is called **proximal gradient descent**. If make all components of θ smaller (at the same time that we threshold the small ones to zero), then this is actually guaranteed to converge, and will produce for us what is called a **lasso** solution!

Summary

1. In optimization, we think about tradeoffs: complexity of iteration vs number of iterations vs precision of approximation
    - Gradient descent: less complex iterations, more iterations in total, more robust
    - Newton: more complex iterations, but few of them for good functions, less robust

2. Start with pre-built code like `optim()` or `nls()`, which are incredibly useful. Implement your own as needed
3. Stochastic methods are critical when you have a lot of data; sparse methods are useful when you have a lot of parameters