Lecture 10: Optimization

STAT GR5206 *Statistical Computing & Introduction to Data Science*

Gabriel Young
Columbia University

June 21, 2017

Course Notes

Exam:

- Take home final exam.
- Final exam posted on 06/28/2017 (Wednesday Morning).
- Final exam due on 06/29/2017 (Thursday 11:59pm).

Homeworks and Labs

- Lab 7 due this Friday by 11:59pm.
- Lab 8 (maybe?)
- Homework 6 (optimization) due next Tuesday by 11:59pm.
- Recommended homeworks 7 (MOM and MLE) and 8 (Split/Apply/Combine). HW 7 and HW 8 are not due. Solutions will be posted.

Last Time

- **Data as Models**: The Method of Moments and the MLE. Testing model fit.

MLE Example

Let $X_1, \ldots X_n \overset{iid}{\sim} \exp(\lambda)$

$$f(x \mid \lambda) = \lambda e^{-\lambda x}, \; x > 0$$

① Compute MLE of λ

$$L(\lambda \mid x_1, \ldots, x_n) = \prod_{i=1}^{n} f(x_i \mid \lambda)$$

$$= f(x_1 \mid \lambda) \cdot f(x_2 \mid \lambda) \ldots f(x_n \mid \lambda)$$

$$= (\lambda e^{-\lambda x_1})(\lambda e^{-\lambda x_2}) \ldots (\lambda e^{-\lambda x_n})$$

$$= \lambda^n e^{-\lambda \Sigma x_i}$$

$$\ell(\lambda \mid x_i, \dots x_n) = \log(L(\lambda \mid x_1, \dots, x_n))$$
$$= \log(\lambda^n e^{-\lambda \Sigma x_i})$$
$$= n \log \lambda - \lambda \Sigma x_i$$

$$\frac{d\ell}{d\lambda} = \frac{n}{\lambda} - \Sigma x_i \overset{set}{=} 0$$
$$\rightarrow \frac{n}{\lambda} = \Sigma x_i \rightarrow \hat{\lambda}_{MLE} = \frac{n}{\Sigma x_i}$$

Functions As Objects

② Compute MOM estimator of λ

$$Ex = \int_0^\infty x \lambda e^{-\lambda x} dx = \frac{1}{\lambda}$$

Equate $Ex = \bar{x}$, $\frac{1}{\lambda} = \bar{x}$, $\hat{\lambda}_{MM} = \frac{1}{\bar{x}}$

* in this case the estimators are the same for MOM & MLE of λ

Functions as Objects

- In R, functions are objects, just like everything else!
- This means that they can be passed to functions as arguments and returned by functions as outputs as well.

Functions as Objects

- In R, functions are objects, just like everything else!
- This means that they can be passed to functions as arguments and returned by functions as outputs as well.
- We've seen examples of this: `curve()`, `nlm()`, `apply()`, etc.

Functions of Functions, Computationally

- We often want to do very similar things to many different functions.
- The procedure is the same, only the function we're working with changes.
- Thus, write one function to do the job, and pass the function as an argument.
- Because `R` treats a function like any other object, we can do this simply: invoke the function by its argument name in the body.

Peeking at a Function's Definition

Typing a function's name, without parentheses, gives youits source code:

```
> sample
```

```
function (x, size, replace = FALSE, prob = NULL)
{
    if (length(x) == 1L && is.numeric(x) && x >= 1) {
        if (missing(size))
            size <- x
        sample.int(x, size, replace, prob)
    }
    else {
        if (missing(size))
            size <- length(x)
        x[sample.int(length(x), size, replace, prob)]
    }
}
<bytecode: 0x10098d6a8>
<environment: namespace:base>
```

Peeking at a Function's Definition

This isn't always that explicit, because some functions are defined in a lower level language (like C or Fortran).

```
> log
```

```
function (x, base = exp(1))  .Primitive("log")
```

Peeking at a Function's Definition

```
> rowSums
```

```
function (x, na.rm = FALSE, dims = 1L)
{
    if (is.data.frame(x))
        x <- as.matrix(x)
    if (!is.array(x) || length(dn <- dim(x)) < 2L)
        stop("'x' must be an array of at least two dimensions")
    if (dims < 1L || dims > length(dn) - 1L)
        stop("invalid 'dims'")
    p <- prod(dn[-(1L:dims)])
    dn <- dn[1L:dims]
    z <- if (is.complex(x))
        .Internal(rowSums(Re(x), prod(dn), p, na.rm)) + (0+1i) *
            .Internal(rowSums(Im(x), prod(dn), p, na.rm))
    else .Internal(rowSums(x, prod(dn), p, na.rm))
    if (length(dn) > 1L) {
        dim(z) <- dn
```

The Function Class

Functions are their own **class** in R:

```
> class(sin)
```

```
[1] "function"
```

```
> class(sample)
```

```
[1] "function"
```

```
> resample = function(x) {
+   return(sample(x, size=length(x), replace=TRUE))
+ }
> class(resample)
```

```
[1] "function"
```

Facts About Functions

A call to `function()` creates and returns a function object.

- Can see the body `body()`
- Can see the arguments with `args()`
- Can see the environment with `environment()`

```
> body(resample)
```

```
{
    return(sample(x, size = length(x), replace = TRUE))
}
```

```
> args(resample)
```

```
function (x)
NULL
```

Facts About Functions

```
> environment(resample)
```

```
<environment: R_GlobalEnv>
```

Example: `grad()`

- Many problems in statistics come down to optimization. (So do lots of problems in economics, physics, computer science, etc.)
- Lots of optimization routines require the gradient of the **objective function**—this is the function that is to be minimized or maximized.
- Recall, the gradient of f at $x = (x_1, \ldots x_n)$ is just the vector that collects all the partial derivatives:

$$\nabla f(x) = \begin{pmatrix} \frac{\partial f(x)}{\partial x_1} \\ \vdots \\ \frac{\partial f(x)}{\partial x_n} \end{pmatrix}$$

Example: grad()

- Note that we do basically the same thing to get the gradient of f at x no matter what f is:

```
(1) Find partial derivative of f with
                 respect to each component of x
(2) Return the vector of partial derivatives
```

Example: `grad()`

- Note that we do basically the same thing to get the gradient of f at x no matter what f is:

```
(1) Find partial derivative of f with
                 respect to each component of x
(2) Return the vector of partial derivatives
```

- It makes no sense to rewrite this every time we change f!
- Hence, write code to calculate the gradient of an arbitrary function.
- We could write our own, but there are lots of tricky issues:
    - Best way to calculate partial derivative?
    - What if x is at the edge of the domain of f?
- Fortunately, someone has already done this for us.

Example: grad()

From the package numDeriv:

```
> library(numDeriv)
> args(grad)
```

```
function (func, x, method = "Richardson", side = NULL, method
    ...)
NULL
```

Example: grad()

From the package numDeriv:

```
> library(numDeriv)
> args(grad)
```

```
function (func, x, method = "Richardson", side = NULL, method
    ...)
NULL
```

- func is a function which returns a single floating-point value.
- x is a vector, at which we want to evaluate the derivative of func.
- Extra arguments in ... get passed along to func.
- Other functions in the package for, e.g., the Hessian matrix (matrix of second partial derivatives)

Simple Example

So, does it work as advertized?

```
> simpleFun = function(x) {
+   # x is a vector of length 2
+   return(x[1]^2 + 1/3*x[2]^2)
+ }
> xpt <- runif(n = 2, min = -2, max = 2)
> grad(simpleFun, xpt)
```

```
[1]  1.300305 -1.273476
```

```
> c(2*xpt[1], 2/3*xpt[2])
```

```
[1]  1.300305 -1.273476
```

Basic Optimization

Examples of Optimization Problems

Many we've already seen in this class!

- Minimize mean-squared error of regression surface (Gauss, c. 1800)
- Maximize likelihood of distribution (Fisher, c. 1918)
- Fitting general models by minimizing the training mean-squared error (GMP data)

Formal Optimization Set-up

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

$$\theta^* = \arg\min_{\theta} f(\theta).$$

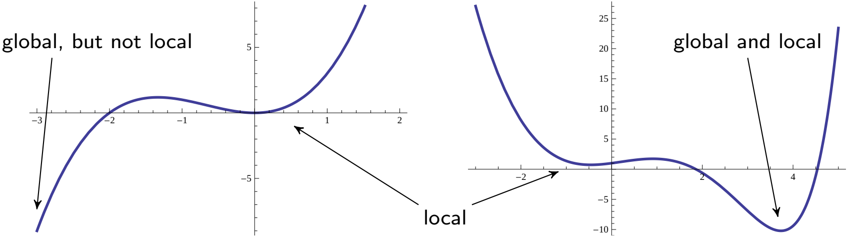
- Basics: maximizing f is minimizing $-f$:

$$\arg\max_{\theta} f(\theta) = \arg\min_{\theta} -f(\theta).$$

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

$$\arg\min_{\theta} f(\theta) = \arg\min_{\theta} h(f(\theta)).$$

Types of Minima



Local and global minima

A minimum of f at θ^* is called:

- **Global** if f assumes no smaller value on its domain.
- **Local** if there is some open neighborhood U of θ^* such that $f(\theta^*)$ is a global minimum of f restricted to U.

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).

- **Approximation**: How close can we get to $f(\theta^*)$, or (better yet) θ^*?
- **Convergence Speed**: How many iterations does the algorithm take to get there? Varies with precision of approximation, niceness of f, size of $\mathcal{D}$, size of data, method...
- **Iteration Cost**: Most optimization algorithms use **successive approximation**, so must consider cost of each iteration when thinking about efficiency.

Analytic criteria for local minima

Recall that one-dimensional x^* is a local minimum of smooth function f if

$$f'(x^*) = 0 \qquad \text{and} \qquad f''(x^*) > 0 \ .$$

Analytic criteria for local minima

Recall that one-dimensional x^* is a local minimum of smooth function f if

$$f'(x^*) = 0 \qquad \text{and} \qquad f''(x^*) > 0\,.$$

This all carries over to multiple dimensions!

For $\theta \in \mathbb{R}^d$,

$$\nabla f(\theta^*) = 0 \qquad \text{and} \qquad H_f(\theta^*) = \left(\frac{\partial f}{\partial\theta_i\partial\theta_j}(\theta^*)\right)_{i,j=1,\ldots,d} \text{ positive definite.}$$

The $d \times d$-matrix $H_f(\theta)$ is called the **Hessian matrix** of f at θ.

You Remember Multivariate Calculus, Right?

Let $f : \mathbb{R}^d \to \mathbb{R}$,

The **gradient** of f, denoted ∇f, is a vector of length n with each element equal to the partial derivative of f. Meaning,

$$\nabla f = \left(\frac{\partial f}{\partial \theta_1}, \frac{\partial f}{\partial \theta_2}, \dots, \frac{\partial f}{\partial \theta_d}\right).$$

You Remember Multivariate Calculus, Right?

Let $f : \mathbb{R}^d \to \mathbb{R}$,

The **gradient** of f, denoted ∇f, is a vector of length n with each element equal to the partial derivative of f. Meaning,

$$\nabla f = \left(\frac{\partial f}{\partial \theta_1}, \frac{\partial f}{\partial \theta_2}, \ldots, \frac{\partial f}{\partial \theta_d} \right).$$

The **Hessian matrix** of f is a square matrix of the second order partial derivatives. Meaning, for $i, j = 1, 2, \ldots, d$,

$$[H_f]_{i,j} = \frac{\partial f}{\partial \theta_i \partial \theta_j}$$

You Remember Multivariate Calculus, Right?

Let $f : \mathbb{R}^d \to \mathbb{R}$,

The **gradient** of f, denoted ∇f, is a vector of length n with each element equal to the partial derivative of f. Meaning,

$$\nabla f = \left(\frac{\partial f}{\partial \theta_1}, \frac{\partial f}{\partial \theta_2}, \ldots, \frac{\partial f}{\partial \theta_d}\right).$$

The **Hessian matrix** of f is a square matrix of the second order partial derivatives. Meaning, for $i, j = 1, 2, \ldots, d$,

$$[H_f]_{i,j} = \frac{\partial f}{\partial \theta_i \partial \theta_j}$$

The Hessian matrix is **positive definite** if $z^T H_f z > 0$ for all non-zero vectors $z \in \mathbb{R}^n$. (Positive semidefinite if $z^T H_f z \geq 0$.) $H_f(\theta^*)$ positive definite roughly means that the function f is "curved upwards" at θ^*.

You Remember Multivariate Calculus, Right?

Example

Let $f : \mathbb{R}^2 \to \mathbb{R}$ be the function $f(\theta_1, \theta_2) = \theta_1^2\theta_2$,

- Find the gradient of f and find $\nabla f(2, 3)$.
- Find the Hessian matrix of f and also the Hessian matrix evaluated at point $(2, 3)$.

You Remember Multivariate Calculus, Right?

Example

Let $f : \mathbb{R}^2 \to \mathbb{R}$ be the function $f(\theta_1, \theta_2) = \theta_1^2\theta_2$,

- Find the gradient of f and find $\nabla f(2, 3)$.
- Find the Hessian matrix of f and also the Hessian matrix evaluated at point $(2, 3)$.

Gradient

$$\frac{\partial f}{\partial \theta_1} = 2\theta_1\theta_2 \text{ and } \frac{\partial f}{\partial \theta_2} = \theta_1^2,$$

so

$$\nabla f = (2\theta_1\theta_2,\ \theta_1^2),$$

and

$$\nabla f(2, 3) = (12, 4).$$

You Remember Multivariate Calculus, Right?

Example

Let $f : \mathbb{R}^2 \to \mathbb{R}$ be the function $f(\theta_1, \theta_2) = \theta_1^2 \theta_2$,

- Find the gradient of f and find $\nabla f(2, 3)$.
- Find the Hessian matrix of f and also the Hessian matrix evaluated at point $(2, 3)$.

Hessian

$$\frac{\partial f}{\partial \theta_1} = 2\theta_1\theta_2 \quad \text{and} \quad \frac{\partial f}{\partial \theta_2} = \theta_1^2,$$

$$\frac{\partial f}{\partial \theta_1^2} = 2\theta_2, \quad \frac{\partial f}{\partial \theta_1 \partial \theta_2} = 2\theta_1, \quad \frac{\partial f}{\partial \theta_2 \partial \theta_1} = 2\theta_1, \quad \text{and} \quad \frac{\partial f}{\partial \theta_2^2} = 0,$$

so

$$H_f = \begin{Bmatrix} 2\theta_2 & 2\theta_1 \\ 2\theta_1 & 0 \end{Bmatrix}, \quad \text{and} \quad H_f(2, 3) = \begin{Bmatrix} 6 & 4 \\ 4 & 0 \end{Bmatrix},$$

Optima

Numerical methods

All numerical minimization methods perform roughly the same steps:

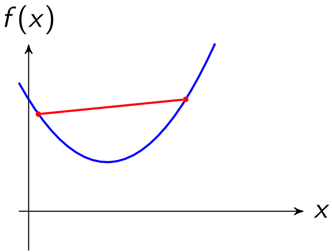
- Start with some point x_0.
- Our goal is to find a sequence $x_0, \ldots, x_m$ such that $f(x_m)$ is a minimum.
- At a given point x_n, compute properties of f (such as $f'(x_n)$ and $f''(x_n)$).
- Based on these values, choose the next point x_{n+1}.

The information $f'(x_n)$, $f''(x_n)$ etc is always *local at* x_n, and we can only decide whether a point is a local minimum, not whether it is global.

Convex Functions

Definition

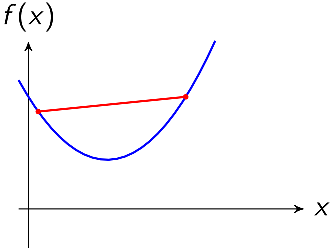
A function f is **convex** if every line segment between function values lies above the graph of f.



Convex Functions

Definition

A function f is **convex** if every line segment between function values lies above the graph of f.



Analytic criterion

A twice differentiable function is convex if $f''(x) \geq 0$ (or $H_f(\theta)$ positive semidefinite) for all x.

Convex Functions

Implications for optimization

If f is convex, then:

- $f'(x) = 0$ is a sufficient criterion for a minimum.
- Local minima are global.
- If f is **strictly convex** ($f'' > 0$ or H_f positive definite), there is only one minimum (which is both gobal and local).

Gradients Tell Us How f Changes

Recall

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

Gradients Tell Us How f Changes

Recall

$$f'(x_0) = \frac{\partial f}{\partial x}\bigg|_{x=x_0} = \lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0}.$$

Therefore,

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

Locally, the function looks linear!

To minimize a linear function move down the slope.

Gradients Tell Us How f Changes

Recall

$$f'(x_0) = \frac{\partial f}{\partial x}\bigg|_{x=x_0} = \lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0}.$$

Therefore,

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

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

Algorithm

Gradient descent searches for a minimum of f.

1. Start with an initial guess for θ and a step size $\eta > 0$.
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 approximation of θ^*.

Gradient Descent

Algorithm

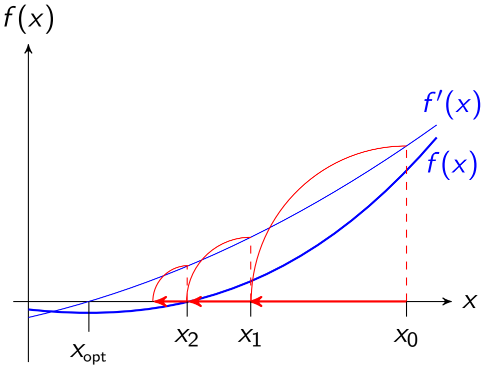
Gradient descent searches for a minimum of f.

1. Start with an initial guess for θ and a step size $\eta > 0$.
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 approximation of θ^*.

Variations

Adaptively adjust η to ensure improvement or search along the gradient direction for minimum.

Gradient Descent



Pros and Cons

Pro:

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

Con:

- 'Small enough' η could be *very* small.
- Slowness or zig-zagging if components of ∇f are very different sizes.

Gradient Descent

Pros and Cons

Pro:

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

Con:

- 'Small enough' η could be *very* small.
- Slowness or zig-zagging if components of ∇f are very different sizes.

How much work does gradient descent require to find the minimum?

Scaling

Big-O Notation

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

means

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

for some $c \neq 0$.

Scaling

Big-O Notation

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

means

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

for some $c \neq 0$.

Examples

- $5x^2 - 200x + 36 = O(x^2)$.
- $\frac{e^x}{1-e^x} = O(1)$.

We often talk about scaling in this way: useful to look at big picture and hide the details.

How Much Work is Gradient Descent?

Pro:

- **Number of Iterations**: For nice f, $f(\theta) \leq f(\theta^*) + \epsilon$ in $O(\epsilon^{-2})$ iterations.
  - A litte faster – $O(\log \epsilon^{-1})$ – for *very* nice f.
- **Cost of Each Iteration**: For $\theta \in \mathbb{R}^d$, to get $\nabla f(\theta)$, must take d derivatives. Cost is $O(d)$.

Con:

- Taking derivative can slow down as data grows – each iteration might really be $O(nd)$.

Gradient Descent

With our knowledge of `grad()`, we can now write a very useful function for peforming gradient descent.

Code Example

Gradient Descent

Let's try it out on our simple example!

```
> x0 <- c(-1.9, -1.9)
> gd <- grad.descent(simpleFun, x0)
> gd$x
```

```
[1] -5.437919e-07 -1.490486e-02
```

```
> gd$k
```

```
[1] 144
```

Note: the minimum here is achieved at (0,0), so this is right

Gradient Descent

Example: Linear Regression

Let's set up a linear regression simulation:

```
> n <- 100
> p <- 2
> pred <- matrix(rnorm(n*p), n, p)
> beta <- c(1, 4)
> resp <- pred %*% beta + rnorm(n)
> lm.coefs <- coef(lm(resp ~ pred + 0))
> lm.coefs
```

```
    pred1     pred2
0.8931815 3.9234320
```

Gradient Descent

Example: Linear Regression

Let's now try out gradient descent:

```
MSE <- function(beta) {
  return(sum((resp - pred %*% beta)^2))
}
grad.descent(MSE, x0 = c(0,0), step.size = 0.05,
             max.iter = 200)

Error in grad.default(f, xmat[, k - 1], ...) :
  function returns NA at 1.20302765942042e+150
  7.12716833990127e+148 distance from x.
```

Gradient Descent

Example: Linear Regression

Let's now try out gradient descent:

```
MSE <- function(beta) {
  return(sum((resp - pred %*% beta)^2))
}
grad.descent(MSE, x0 = c(0,0), step.size = 0.05,
             max.iter = 200)

Error in grad.default(f, xmat[, k - 1], ...) :
  function returns NA at 1.20302765942042e+150
  7.12716833990127e+148 distance from x.
```

What happened??

You should practice your debugging skills to confirm this, but the step size is simply too large, and so gradient descent is not converging.

Gradient Descent

Example: Linear Regression

A simple fix is just to take a smaller step size

```
> out = grad.descent(MSE, x0 = c(0,0), step.size = 1e-3,
+                    max.iter = 200)
> out$k
```

```
[1] 62
```

```
> out$x
```

```
[1] 0.8931187 3.9233875
```

```
> lm.coefs
```

```
    pred1     pred2
0.8931815 3.9234320
```

Gradient Descent

Up Next

We perhaps don't want to fiddle around with the step size manually.

- What's the problem with this?
- What's the problem with taking it just to be super tiny, so that we always converge?

Now we'll learn a more principled strategy

Taylor Series

The Taylor series gives us a way to estimate the value of a function f near x_0 if we know the derivatives of f at x_0.

A one-dimensional **Taylor series** around a point $x = x_0$ is given by

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{1}{2!}(x - x_0)^2 f''(x_0) + \frac{1}{3!}(x - x_0)^2 f'''(x_0) + \ldots + \frac{1}{n!}(x - x_0)^n f^{(n)}(x_0) + \ldots$$

Taylor Series

The Taylor series gives us a way to estimate the value of a function f near x_0 if we know the derivatives of f at x_0.

A one-dimensional **Taylor series** around a point $x = x_0$ is given by

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{1}{2!}(x - x_0)^2 f''(x_0) + \frac{1}{3!}(x - x_0)^2 f'''(x_0) + \ldots + \frac{1}{n!}(x - x_0)^n f^{(n)}(x_0) + \ldots$$

Taylor's theorem states that any function satisfying certain conditions can be expressed as a Taylor series.

Taylor Series

The degree-n **Taylor polynomial** approximation to f at point $x = x_0$ is given by

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

Taylor Series

Let's consider a quadratic approximation to f:

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

Taylor Series

Let's consider a quadratic approximation to f:

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

Assume x_0 is a minimum, then $f'(x_0) = 0$ and the above simplifies to

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

Near a minimum, a smooth function looks like a parabola!

Taylor Series

The same is true in the multivariate case: for a minimizing value θ_0,

$$\begin{aligned} f(\theta) &\approx f(\theta_0) + (\theta - \theta_0)\nabla f(\theta_0) + \frac{1}{2}(\theta - \theta_0)^T H_f(\theta_0)(\theta - \theta_0) \\ &= f(\theta_0) + \frac{1}{2}(\theta - \theta_0)^T H_f(\theta_0)(\theta - \theta_0). \end{aligned}$$

Near a minimum, a smooth function looks like a parabola!

Minimizing a Quadratic Function

If we know,

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

we can minimize exactly:

$$f'(x) = 2ax^* + b = 0$$
$$x^* = -\frac{b}{2a}.$$

Minimizing a Quadratic Function

If we know,

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

we can minimize exactly:

$$f'(x) = 2ax^* + b = 0$$
$$x^* = -\frac{b}{2a}.$$

If we know,

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

then

$$f'(x) = a(x^* - x_0) + b = 0$$
$$x^* = x_0 - a^{-1}b.$$

Newton's Method

Let θ_0 be our current guess at a minimizing value. We find a quadratic approximation of f at θ_0:

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

Then we update our guess by minimizing the above with

$$\nabla f(\theta) = \nabla f(\theta_0) + H_f(\theta_0)(\theta^* - \theta_0) = 0$$
$$\theta^* = \theta_0 - (H_f(\theta_0))^{-1}\nabla f(\theta_0).$$

Newton's Method

Let θ_0 be our current guess at a minimizing value. We find a quadratic approximation of f at θ_0:

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

Then we update our guess by minimizing the above with

$$\nabla f(\theta) = \nabla f(\theta_0) + H_f(\theta_0)(\theta^* - \theta_0) = 0$$
$$\theta^* = \theta_0 - (H_f(\theta_0))^{-1}\nabla f(\theta_0).$$

- If f is exactly quadratic (and $H_f(\theta)^{-1}$ exists), this finds the minimizer exactly.
- If f isn't quadratic, keep pretending it is until we get close to θ^*, when it's nearly true.

Newton's Method

Algorithm

Gradient descent searches for a minimum of f.

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

Newton's Method

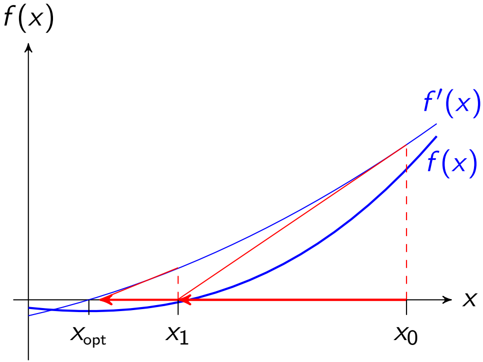
Algorithm

Gradient descent searches for a minimum of f.

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

- Like gradient descent, but with inverse Hessian as the step size.
- Inverse Hessian tells you how far you can go in the current gradient direction.

Newton's Method



Newton's Method: Multiple Dimensions

Newton's Method for Minima

$$\text{Update: } \theta_1 \leftarrow \theta_0 - H_f(\theta_0)^{-1}\nabla f(\theta_0).$$

- The inverse of $H_f(\theta)$ exists only if the matrix is positive definite (not if it is only semidefinite), i.e. f has to be strictly convex.
- The Hessian measures the curvature of f.

Newton's Method: Multiple Dimensions

Newton's Method for Minima

$$\text{Update: } \theta_1 \leftarrow \theta_0 - H_f(\theta_0)^{-1} \nabla f(\theta_0).$$

- The inverse of $H_f(\theta)$ exists only if the matrix is positive definite (not if it is only semidefinite), i.e. f has to be strictly convex.
- The Hessian measures the curvature of f.

Effect of the Hessian

Multiplication by H_f^{-1} in general changes the direction of $\nabla f(\theta_0)$. The correction takes into account how $\nabla f(\theta)$ changes away from θ_0, as estimated using the Hessian at θ_0.

Newton's Method

Pros and Cons

Pro:

- Step-size chosen adaptively through second derivatives, much harder to get zig-zagging, over-shooting, etc.
- Always converges if $f''(x) > 0$ (or H_f positive definite).
- Also guaranteed to need $O(\epsilon^{-2})$ steps to get within ϵ of the optimum.
- Only $O(\log\log\epsilon^{-1})$ steps for *very* nice functions.
- Typically many fewer steps than gradient descent.

Newton's Method

Pros and Cons

Pro:

- Step-size chosen adaptively through second derivatives, much harder to get zig-zagging, over-shooting, etc.
- Always converges if $f''(x) > 0$ (or H_f positive definite).
- Also guaranteed to need $O(\epsilon^{-2})$ steps to get within ϵ of the optimum.
- Only $O(\log\log\epsilon^{-1})$ steps for *very* nice functions.
- Typically many fewer steps than gradient descent.

Con:

- Hopeless if H_f doesn't exist or isn't invertible.
- Need to take $O(d^2)$ second derivatives plus d first derivatives.
- Inverting H_f is $O(d^3)$, which beomes unwieldy in high dimensions.

Getting Around the Hession

Want to use Hessian to improve convergence, but don't want to keep computing it at each step.

Approaches

- Use knowledge of the system to get approximations of the Hessian, instead of taking derivatives. ("Fisher scoring").
- Use only diagonal entries (d unmixed second derivatives).
- Use $H_f(\theta_0)$ at initial guess θ_0 and hope H_f doesn't change that much with θ.
- Recompute $H_f(\theta)$ only every k steps for $k > 1$.
- Fast, approximate updates to the Hessian at each step (BFGS).

Other Methods

There are lots...

For example, "Nedler-Mead" (a.k.a. the simplex method) or coordinate descent.

Curve Fitting

Curve Fitting by Optimizing

Set-up

- We have data $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.
- We also have possible curves, which we label $r(x;\theta)$. For example,
  - $r(x) = \theta \cdot x$
  - $r(x) = \theta_0 x^{\theta_1}$
  - $r(x) = \sum_{j=1}^{q} \theta_j b_j(x)$ for fixed basis functions b_j.

Curve Fitting by Optimizing

Least squares curve fitting:

$$\hat{\theta} = \arg\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} (y_i - r(x_i; \theta))^2.$$

'Robust' curve fitting:

$$\hat{\theta} = \arg\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} \psi(y_i - r(x_i; \theta)),$$

where ψ is a 'robust' loss function.

Optimization in R: `optim()`

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

- `par`: Inital parameter guess; mandatory.
- `fn`: Function to be minimized; 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, step size, tolerance for convergence, etc.
- `hessian`: Should the final Hessian be returned? Default is FALSE.

Returns the location (`$par`) and the value (`$val`) of the optimum, diagnositics, possibly `$hessian`.

Optimization in R: optim()

GMP Example

```
> gmp     <- read.table("gmp.txt", header = TRUE)
> gmp$pop <- gmp$gmp/gmp$pcgmp
> # install.packages("numDeriv")
> library(numDeriv)
> mse <- function(theta) {
+   mean((gmp$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)
```

GMP Example

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

```
$par
[1] 6493.2563739    0.1276921

$value
[1] 61853983

$counts
function gradient
      63       11
```

GMP Example

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

```
$convergence
[1] 0

$message
NULL

$hessian
             [,1]         [,2]
[1,]      52.5021      4422071
[2,] 4422071.3594 375729087979
```

Optimization in R: `nls()`

- `optim()` is a general-purpose optimizer.
- So is `nls()` – try them both if one doesn't work!
- `nls()` is for non-linear least squares.
- The default optimization method for `nls()` is a version of Newton's method.

Optimization in R: `nls()`

```
nls(formula, data, start, control, [lot of others...])
```

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

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

Optimization in R: nls()

GMP Example

```
> fit2 <- nls(pcgmp ~ theta0*pop^theta1, data = gmp,
+             start = list(theta0 = 5000, theta1 = 0.10))
```

Optimization in R: nls()

GMP Example

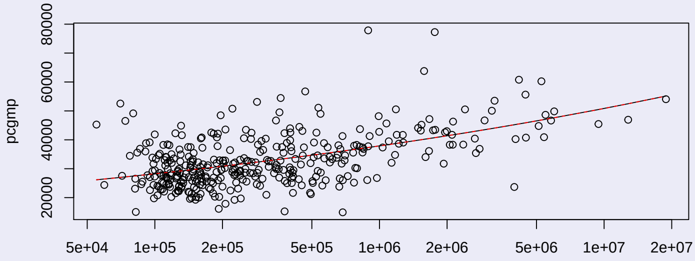
```
> summary(fit2)
```

```
Formula: pcgmp ~ theta0 * pop^theta1

Parameters:
        Estimate Std. Error t value Pr(>|t|)
theta0 6.494e+03  8.565e+02   7.582 2.87e-13 ***
theta1 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
```

Optimization in R: nls()

GMP Example

```
> plot(pcgmp ~ pop, data = gmp, log = 'x')
> 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 = "red")
```



Summary

Summary

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

Constrained Optimization

Up Next: Constrained Optimization

Set-up

A **constrained** optimization problem adds an additional requirement on θ.

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

$$\theta^* = \arg\min_\theta f(\theta),$$

$$\text{subject to} \quad \theta \in \mathcal{G},$$

  where $\mathcal{G} \subset \mathcal{D}$ is called the **feasible set**.

- The set $\mathcal{G}$ is often defined by equations, e.g.

$$\theta^* = \arg\min_\theta f(\theta),$$

$$\text{subject to} \quad g(\theta) \geq 0.$$

  The equation g is called a **constraint**.

Next: Constrained Optimization

So far

- If f is differentiable, we can search for local minima using gradient descent.
- If f is sufficiently nice (convex and twice differentiable), we know how to speed up the search process using Newton's method.

Next: Constrained Optimization

Constrained problems

- Numerical minimizers (like those discussed) use the criterion $\nabla f(\theta) = 0$ for the minimum.
- In a constrained problem, the minimum is *not* identified by this criterion.

Next: Constrained Optimization

Constrained problems

- Numerical minimizers (like those discussed) use the criterion $\nabla f(\theta) = 0$ for the minimum.
- In a constrained problem, the minimum is *not* identified by this criterion.

Next steps

We will figure out how the constrained minimum can be identified. We have to distinguish two cases:

- Problems involving only equalities as constraints, i.e. subject to $g(\theta) = 0$ (sometimes easy).
- Problems also involving inequalities, i.e. subject to $g(\theta) \geq 0$ (a bit more complex).

Example: Maximum Likelihood of a Multinomial Distribution

Example: MLE for the Multinomial

I roll a single six-sided die n times with n_1, n_2, n_3, n_4, n_5, and n_6 counting the outcomes.

Likelihood and Log-likelihood

$$L(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6) = \frac{n!}{n_1!n_2!n_3!n_4!n_5!n_6!} \prod_{i=1}^{6} \theta_i^{n_i}$$

$$\ell(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6) = \log \frac{n!}{n_1!n_2!n_3!n_4!n_5!n_6!} + \sum_{i=1}^{6} n_i \log(\theta_i).$$

Example: MLE for the Multinomial

I roll a single six-sided die n times with n_1, n_2, n_3, n_4, n_5, and n_6 counting the outcomes.

Likelihood and Log-likelihood

$$L(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6) = \frac{n!}{n_1!n_2!n_3!n_4!n_5!n_6!} \prod_{i=1}^{6} \theta_i^{n_i}$$

$$\ell(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6) = \log \frac{n!}{n_1!n_2!n_3!n_4!n_5!n_6!} + \sum_{i=1}^{6} n_i \log(\theta_i).$$

So let's find the optimizing $\theta = (\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6)$ by setting the gradient (of the log-likelihood) equal to 0: for $i = 1, 2, 3, 4, 5, 6$,

$$\frac{\partial \ell}{\partial \theta_i} = \frac{n_i}{\theta_i} = 0, \quad \therefore \theta_i = \infty.$$

Example: MLE for the Multinomial

We forgot that $\sum_{i=1}^{6} \theta_i = 1$!

Use the above constraint to eliminate one of the variables: $\theta_6 = 1 - \sum_{i=1}^{5} \theta_i$. Rewrite the log-likelihood:

$$\ell(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5) = \log \frac{n!}{n_1! n_2! n_3! n_4! n_5! n_6!} + \sum_{i=1}^{5} n_i \log(\theta_i) + n_6 \log(1 - \sum_{j=1}^{5} \theta_j).$$

Example: MLE for the Multinomial

We forgot that $\sum_{i=1}^{6} \theta_i = 1$!

Use the above constraint to eliminate one of the variables: $\theta_6 = 1 - \sum_{i=1}^{5} \theta_i$. Rewrite the log-likelihood:

$$\ell(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5) = \log \frac{n!}{n_1!n_2!n_3!n_4!n_5!n_6!} + \sum_{i=1}^{5} n_i \log(\theta_i) + n_6 \log(1 - \sum_{j=1}^{5} \theta_j).$$

To find a minimizer, solve the following equations for $i = 1, 2, 3, 4, 5$:

$$\frac{\partial \ell}{\partial \theta_i} = \frac{n_i}{\theta_i} - \frac{n_6}{1 - \sum_{j=1}^{5} \theta_j} = 0.$$

Example: MLE for the Multinomial

We forgot that $\sum_{i=1}^{6} \theta_i = 1$!

Use the above constraint to eliminate one of the variables: $\theta_6 = 1 - \sum_{i=1}^{5} \theta_i$. Rewrite the log-likelihood:

$$\ell(\theta_1, \theta_2, \theta_3, \theta_4, \theta_5) = \log \frac{n!}{n_1!n_2!n_3!n_4!n_5!n_6!} + \sum_{i=1}^{5} n_i \log(\theta_i) + n_6 \log(1 - \sum_{j=1}^{5} \theta_j).$$

To find a minimizer, solve the following equations for $i = 1, 2, 3, 4, 5$:

$$\frac{\partial \ell}{\partial \theta_i} = \frac{n_i}{\theta_i} - \frac{n_6}{1 - \sum_{j=1}^{5} \theta_j} = 0.$$

Usually elminating a variable with a constraint doesn't work out so nicely!

Lagrange Multipliers

$$\text{Constraint: } g(\theta) = c \implies g(\theta) - c = 0.$$

Define a Lagrangian

$$\mathcal{L}(\theta, \lambda) = f(\theta) - \lambda(g(\theta) - c).$$

Note the above equals f when the constraint is satisfied.

Lagrange Multipliers

$$\text{Constraint: } g(\theta) = c \implies g(\theta) - c = 0.$$

Define a Lagrangian

$$\mathcal{L}(\theta, \lambda) = f(\theta) - \lambda(g(\theta) - c).$$

Note the above equals f when the constraint is satisfied.

Now do *unconstrained* optimization over θ and λ:

$$\begin{aligned} \nabla_\theta \mathcal{L}|_{\theta^*,\lambda^*} &= \nabla f(\theta^*) - \lambda^* \nabla g(\theta^*) = 0 \\ \frac{\partial \mathcal{L}}{\partial \lambda}|_{\theta^*,\lambda^*} &= g(\theta^*) - c = 0. \end{aligned}$$

Optimizing the **Lagrange multiplier** $\mathcal{L}$ enforces the constraint, but more constraints mean more multipliers.

Example: MLE for the Multinomial

We try the dice example again.

$$\mathcal{L}(\theta, \lambda) = \log \frac{n!}{\prod_{i=1}^{6} n_i!} + \sum_{i=1}^{6} n_i \log(\theta_i) - \lambda \left(\sum_{i=1}^{6} \theta_i - 1 \right)$$

$$\frac{\partial \mathcal{L}}{\partial \theta_i}\Big|_{\theta^*, \lambda^*} = \frac{n_i}{\theta_i^*} - \lambda^* = 0 \implies \frac{n_i}{\lambda^*} = \theta_i^*.$$

Now we enforce the constraint:

$$1 = \sum_{i-1}^{6} \theta_i^* = \sum_{i=1}^{6} \frac{n_i}{\lambda^*} \implies \lambda^* = \sum_{i=1}^{6} n_i.$$

Therefore,

$$\theta_i^* = \frac{n_i}{\lambda^*} = \frac{n_i}{\sum_{i=1}^{6} n_i}.$$

Inequality Constraints

Constraint:

$$h(\theta) \leq d \implies h(\theta) - d \leq 0.$$

The region where the constraint is satisfied is the **feasible set**.

Inequality Constraints

Constraint:

$$h(\theta) \leq d \implies h(\theta) - d \leq 0.$$

The region where the constraint is satisfied is the **feasible set**.

Roughly two cases:

1. Unconstrained optimum is inside the feasible set $\implies$ constraint is **inactive**.
2. Optimum is outside the feasible set; constraint is **active** or **binding**. The *constrained* optimum is usually on the boundary.

Inequality Constraints

Constraint:

$$h(\theta) \leq d \implies h(\theta) - d \leq 0.$$

The region where the constraint is satisfied is the **feasible set**.

Roughly two cases:

1. Unconstrained optimum is inside the feasible set $\implies$ constraint is **inactive**.
2. Optimum is outside the feasible set; constraint is **active** or **binding**. The *constrained* optimum is usually on the boundary.

Strategy

Add a Lagrange multiplier and then $\lambda \neq 0 \implies$ constraint binds.

Optimization Under Constraints

Objective

Want to recover minimizing θ^*:

$$\theta^* = \arg\min_{\theta} f(\theta),$$

$$\text{subject to} \quad g(\theta) = 0.$$

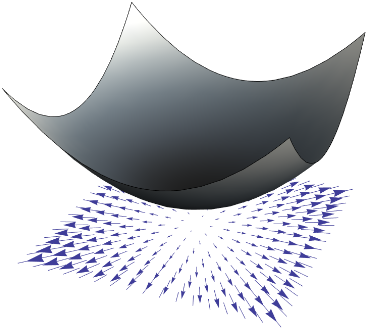
Idea

- The **feasible set** is the set of points θ which satisfy $g(\theta) = 0$,

$$G := \{\theta \,|\, g(\theta) = 0\} \; .$$

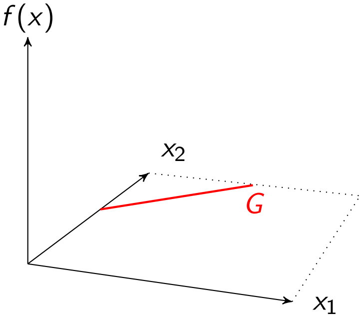
  If g is reasonably smooth, G is a smooth surface in $\mathbb{R}^d$.
- Restrict f to this surface and call the restricted function f_g.
- **Constrained** optimization: looking for the minimum of f_g.

Optimization Under Constraints



$$f(x) = x_1^2 + x_2^2$$

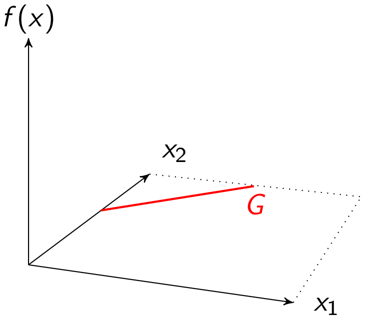
The blue arrows are the gradients $\nabla f(x)$ at various values of x.

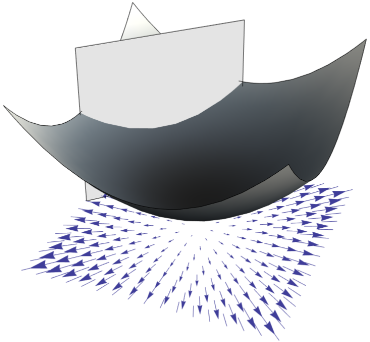


Constraint g.

Here, g is linear, so the graph of g is a (sloped) affine plane. The intersection of the plane with the x_1-x_2-plane is the set G of all points x with $g(x) = 0$.

Optimization Under Constraints





- Make the function f_g given by the constraint $g(x) = 0$ visible by placing a plane vertically through G. The graph of f_g is the intersection of the graph of f with the plane.
- Here, f_g has parabolic shape.
- The gradient of f at the miniumum of f_g is *not* 0.

Mathematical Programming

Problems of this sort have been studied for a long time.

Linear Programming

Optimize $f(\theta)$ subject to $g(\theta) = c$ and $h(\theta) \leq d$.

- Both g, h linear in θ.
- Here θ^* always at a corner of the feasible set.

An Example

Linear Programming

A factory makes cars and trucks, using labor and steel.

- A car takes 40 hours of labor and 1 ton of steel,
- A truck takes 60 hours and 3 tons of steel,
- Total resources: 1600 hours of labor and 70 tons of steel each week.

Total revenue is 13K per car and 27K per truck.

An Example

Linear Programming

A factory makes cars and trucks, using labor and steel.

- A car takes 40 hours of labor and 1 ton of steel,
- A truck takes 60 hours and 3 tons of steel,
- Total resources: 1600 hours of labor and 70 tons of steel each week.

Total revenue is 13K per car and 27K per truck.

Linear Programming

Let c be the number of cars and t the number of trucks. Maximize revenue:

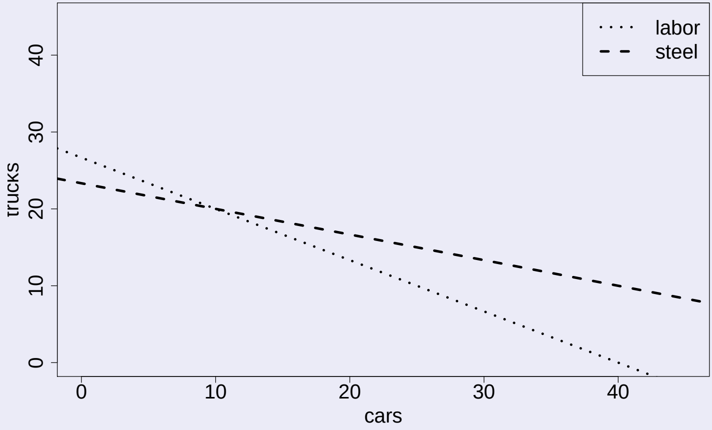
$$f(c, t) = 13c + 27t,$$

subject to

- $40c + 60t < 1600$.
- $c + 3t < 70$.

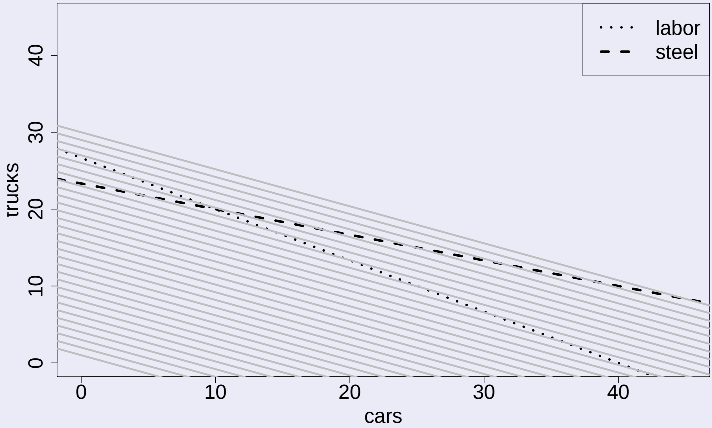
An Example

The feasible region



An Example

The feasible region, plus lines of equal profit



Maybe a More Interesting Example

- **Given:** Expected returns $r_1, \ldots, r_p$ among p financial assets, their $p \times p$ matrix of variances and covariances Σ.
- **Find:** The portfolio shares $\theta_1, \ldots, \theta_p$ which maximize expected returns.
- **Such that**: total variance is below some limit, covariance with other stocks or portfolios is under some limit.

- Expected returns $f(\theta) = r \cdot \theta$.
- Constraints:
  - $\sum_{i=1}^{p} \theta_i = 1$, $\theta_i > 0$.
  - Covariance constraints linear in θ.
  - Variance constraint is quadratic, over-all variance $\theta^T \Sigma \theta$.

A.K.A. "interior point", "central path", etc.

- Having constraints switch on and off abruptly is annoying, especially with gradient methods.
- Fix $\mu > 0$ and try minimizing

$$f(\theta) - \mu \log (d - h(\theta))$$

- The above "pushes away" from the barrier – more and more weakly as $\mu \to 0$

Barrier Methods

Algorithm

- Choose initial guess of θ in the feasible set and initial μ.
- While ((not too tired) and (making adequate progress))
  - Minimize $f(\theta) - \mu \log(d - h(\theta))$,
  - Reduce μ.
- Return final θ as approximation of θ^*.

R Implementation

`constrOptim()` implements the barrier method.

```
> factory.n <- list(c("labor","steel"), c("car","truck"))
> factory   <- matrix(c(40, 1, 60, 3), nrow = 2,
+                       dimnames = factory.n)
> available         <- c(1600, 70)
> names(available) <- rownames(factory)
> prices            <- c(car = 13, truck = 27)
> revenue <- function(output) {return(-output %*% prices)}
> plan <- constrOptim(theta = c(5, 5), f = revenue,
+                       grad = NULL, ui = -factory,
+                       ci = -available, meth = "Nelder-Mead")
> plan$par
```

```
[1]  9.999896 20.000035
```

`constrOptim()` only works with constraints like $\mathbf{u}\theta \geq c$, so minus signs.

Penalties vs. Constraints

$$\arg \min_{\theta : h(\theta) \le d} f(\theta) \quad \leftrightarrow \quad \arg\min_{\theta, \lambda} f(\theta) - \lambda(h(\theta) - d).$$

Note that d plays no role in minimizing θ.

Penalties vs. Constraints

$$\arg \min_{\theta : h(\theta) \leq d} f(\theta) \quad \leftrightarrow \quad \arg \min_{\theta, \lambda} f(\theta) - \lambda(h(\theta) - d).$$

Note that d plays no role in minimizing θ.

We could just as well minimize

$$f(\theta) - \lambda h(\theta).$$

- **Constrained optimization** uses a constraint level d.
- **Penalized optimization** uses a penalty factor λ.

Statistical Applications of Penalization

Mostly in this class you've seen unpenalized estimates (least squares, maximum likelihood) but lots of modern advanced methods rely on penalties.

- For when the direct estimate is too unstable.
- For handling high-dimensional cases.
- For handling non-parametrics.

Ordinary Least Squares

No penalization; minimize MSE of the linear function $\beta \cdot x$:

$$\hat{\beta} = \arg\min_{\beta} \frac{1}{n} \sum_{i=1}^{n} (y_i - \beta \cdot x_i)^2 = \arg\min_{\beta} MSE(\beta).$$

Closed form solution if we can invert matrices:

$$\hat{\beta} = (X^T X)^{-1} X^T y,$$

where X is the $n \times p$ design matrix of predictors and y is the $n \times 1$ vector of responses.

Ridge Regression

Now put a penalty on the *magnitude* of the coefficient vector:

$$\tilde{\beta} = \arg\min_{\beta} MSE(\beta) + \mu \sum_{j=1}^{p} \beta_j^2 = \arg\min_{\beta} MSE(\beta) + \mu \|\beta\|_2^2.$$

Penalizing β this way makes the estimate more *stable*; especially useful for

- Lots of noise.
- Collinear data (X not of "full rank").
- High-dimensional, $p > n$ data (which implies collinearity).

This is called **ridge regression**. Closed form solution:

$$\tilde{\beta} = (X^T X + \mu I)^{-1} X^T y$$

The LASSO

Now put a penalty on the sum of the coefficient's absolute values:

$$\beta^\dagger = \arg\min_\beta MSE(\beta) + \mu \sum_{j=1}^{p} |\beta_j| = \arg\min_\beta MSE(\beta) + \mu \|\beta\|_1.$$

This is called **the LASSO**.

- Also stabilizes (like ridge)
- Also handles high-dimensional data (like ridge)
- Enforces **sparsity**: it likes to drive small coefficients exactly to 0.

No closed form, but very efficient interior-point algorithms (e.g., `lars` package)

Spline Smoothing

Minimize the MSE of a smooth, nonlinear function, plus a penalty on curvature:

$$\hat{f} = \arg\min_f \frac{1}{n} \sum_{i=1}^{n} (y_i - f(x_i))^2 + \int (f''(x))^2 dx$$

This fits smooth regressions without assuming any specific functional form.

- Lets you check linear models.
- Makes you wonder why you bother with linear models.

Many different `R` implementations, starting with `smooth.spline`.

How Big a Penalty?

Remember cross-validation?

Rarely know the constraint level or the penalty factor λ. Lots of ways of picking, but often **cross-validation** works well:

- Divide the data into parts.
- For each value of λ, estimate the model on one part of the data.
- See how well the models fit the other part of the data.
- Use the λ which extrapolates best on average.

Summary

- We use Lagrange multipliers to turn constrained optimization problems into unconstrained but penalized ones.
- The nature of the penalty term reflects the sort of constraint we put on the problem:
    - Shrinkage?
    - Sparsity?
    - Smoothness?

Coordinate Descent

Coordinate Descent

Gradient descent and Newton's method adjust all coordinates of θ at once.
This gets harder as the number of dimensions d grows.

Coordinate Descent

Gradient descent and Newton's method adjust all coordinates of θ at once. This gets harder as the number of dimensions d grows.

Coordinate Descent

Never do more than one-dimensional optimization!

- Start with an initial guess θ.
- While ((not too tired) and (still making adequate progress))
  - For $i \in (1 : p)$
    1. Do one-dimensional optimization over the i^{th} coordinate of θ holding all others fixed.
    2. Update i^{th} coordinate to this optimal value.
- Return final value of θ as approximation of θ^*.

Coordinate Descent

Pros and Cons

Pro:

- Can be extremely fast and simple.

Con:

- Must have a good one-dimensional optimizer.
- Can bog down for very tricky functions, especially with lots of interactions among variables.