MultivariateOptimizationAlgortihms

February 21, 2015

1 Algorithms for Optimization and Root Finding for Multivariate Problems

1.1 Optimizers

1.1.1 Newton-Conjugate Gradient

First a note about the interpretations of Newton's method in 1-D:

In the lecture on 1-D optimization, Newton's method was presented as a method of finding zeros. That is what it is, but it may also be interpreted as a method of optimization. In the latter case, we are really looking for zeroes of the first derivative.

Let's compare the formulas for clarification:

Finding roots of f	Geometric Interpretation	Finding Extrema of f	Geometric Interpretation
$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$	Invert linear approximation to f	$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$	Use quadratic approximation of f

These are two ways of looking at exactly the same problem. For instance, the linear approximation in the root finding problem is simply the derivative function of the quadratic approximation in the optimization problem.

Hessians, Gradients and Forms - Oh My! Let's review the theory of optimization for multivariate functions. Recall that in the single-variable case, extreme values (local extrema) occur at points where the first derivative is zero, however, the vanishing of the first derivative is not a sufficient condition for a local max or min. Generally, we apply the second derivative test to determine whether a candidate point is a max or min (sometimes it fails - if the second derivative either does not exist or is zero). In the multivariate case, the first and second derivatives are *matrices*. In the case of a scalar-valued function on $\mathbb{R}^n$, the first derivative is an $n \times 1$ vector called the *gradient* (denoted ∇f). The second derivative is an $n \times n$ matrix called the *Hessian* (denoted H)

Just to remind you, the gradient and Hessian are given by:

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

$$H = \begin{pmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \dfrac{\partial^2 f}{\partial x_1\,\partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_1\,\partial x_n} \\ \dfrac{\partial^2 f}{\partial x_2\,\partial x_1} & \dfrac{\partial^2 f}{\partial x_2^2} & \cdots & \dfrac{\partial^2 f}{\partial x_2\,\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial x_n\,\partial x_1} & \dfrac{\partial^2 f}{\partial x_n\,\partial x_2} & \cdots & \dfrac{\partial^2 f}{\partial x_n^2} \end{pmatrix}$$

One of the first things to note about the Hessian - it's symmetric. This structure leads to some useful properties in terms of interpreting critical points.

The multivariate analog of the test for a local max or min turns out to be a statement about the gradient and the Hessian matrix. Specifically, a function $f : \mathbb{R}^n \to \mathbb{R}$ has a critical point at x if $\nabla f(x) = 0$ (where zero is the zero vector!). Furthermore, the second derivative test at a critical point is as follows:

- If $H(x)$ is positive-definite, f has a local minimum at x
- If $H(x)$ is negative-definite, f has a local maximum at x
- If $H(x)$ has both positive and negative eigenvalues, f has a saddle point at x.

1.1.2 Newton CG Algorithm

Features:

- Minimizes a 'true' quadratic on $\mathbb{R}^n$ in n steps
- Does NOT require storage or inversion of an $n \times n$ matrix.

We begin with : $\mathbb{R}^n \to \mathbb{R}$. Take a quadratic approximation to f:

$$f(x) \approx \frac{1}{2}x^T Hx + b^T x + c$$

Note that in the neighborhood of a minimum, H will be positive-definite (and symmetric). (If we are maximizing, just consider $-H$).

This reduces the optimization problem to finding the zeros of

$$Hx = -b$$

This is a linear problem, which is nice. The dimension n may be very large - which is not so nice. Also, *a priori* it looks like we need to know H. We actually don't but that will become clear only after a bit of explanation.

General Inner Product Recall the axiomatic definition of an inner product $<,>_A$:

- For any two vectors v, w we have
$$< v, w >_A = < w, v >_A$$
- For any vector v
$$< v, v >_A \geq 0$$
with equality $\iff$ $v = 0$.
- For $c \in \mathbb{R}$ and $u, v, w \in \mathbb{R}^n$, we have
$$< cv + w, u > = c < v, u > + < w, u >$$

These properties are known as symmetric, positive definite and bilinear, respectively.

Fact: If we denote the standard inner product on $\mathbb{R}^n$ as $<,>$ (this is the 'dot product'), any symmetric, positive definite $n \times n$ matrix A defines an inner product on $\mathbb{R}^n$ via:

$$< v, w >_A = < v, Aw > = v^T Aw$$

Just as with the standard inner product, general inner products define for us a notion of 'orthogonality'. Recall that with respect to the standard product, 2 vectors are orthogonal if their product vanishes. The same applies to $<,>_A$:

$$< v, w >_A = 0$$

means that v and w are orthogonal under the inner product induced by A. Equivalently, if v, w are orthogonal under A, we have:

$$v^T Aw = 0$$

This is also called *conjugate* (thus the name of the method).

Conjugate Vectors Suppose we have a set of n vectors $p_1, ..., p_n$ that are mutually conjugate. These vectors form a basis of $\mathbb{R}^n$. Getting back to the problem at hand, this means that our solution vector x to the linear problem may be written as follows:

$$x = \sum_{i=1}^{n} \alpha_i p_i$$

So, finding x reduces to finding a conjugate basis and the coefficients for x in that basis.

Note that:

$$p_k^T b = p_k^T A x$$

and because $x = \sum_{i=1}^{n} \alpha_i p_i$, we have:

$$p^T A x = \sum_{i=1}^{n} \alpha_i p^T A p_i$$

we can solve for α_k:

$$\alpha_k = \frac{p_k^T b}{p_k^T A p_k} = \frac{\langle p_k, b \rangle}{\langle p_k, p_k \rangle_A} = \frac{\langle p_k, b \rangle}{\|p_k\|_A^2}.$$

Now, all we need are the p_k's.

A nice initial guess would be the gradient at some initial point x_1. So, we set $p_1 = \nabla f(x_1)$. Then set:

$$x_2 = x_1 + \alpha_1 p_1$$

This should look familiar. In fact, it is gradient descent. For p_2, we want p_1 and p_2 to be conjugate (under A). That just means orthogonal under the inner product induced by A. We set

$$p_2 = \nabla f(x_1) - \frac{p_1^T A \nabla f(x_1)}{p_1^T A p_1} p_1$$

I.e. We take the gradient at x_1 and subtract its projection onto p_1. This is the same as Gram-Schmidt orthogonalization.

The k^{th} conjugate vector is:

$$p_{k+1} = \nabla f(x_k) - \sum_{i=1}^{k} \frac{p_i^T A \nabla f(x_k)}{p_i^T A p_i} p_i$$

The 'trick' is that in general, we do not need all n conjugate vectors.

Convergence rate is dependent on sparsity and condition number of A. Worst case is n^2.

1.1.3 BFGS - Broyden–Fletcher–Goldfarb–Shanno

BFGS is a 'quasi' Newton method of optimization. Such methods are variants of the Newton method, where the Hessian H is replaced by some approximation. We we wish to solve the equation:

$$B_k p_k = -\nabla f(x_k)$$

for p_k. This gives our search direction, and the next candidate point is given by:

$$x_{k+1} = x_k + \alpha_k p_k$$

.

where α_k is a step size.

At each step, we require that the new approximate H meets the secant condition:

$$B_{k+1}(x_{k+1} - x_k) = \nabla f(x_{k+1}) - \nabla f(x_k)$$

There is a unique, rank one update that satisfies the above:

$$B_{k+1} = B_k + c_k v_k v_k^T$$

where

$$c_k = -\frac{1}{(B_k(x_{k+1} - x_k) - (\nabla f(x_{k+1}) - \nabla f(x_k))^T (x_{k+1} - x_k)}$$

and

$$v_k = B_k(x_{k+1} - x_k) - (\nabla f(x_{k+1}) - \nabla f(x_k))$$

Note that the update does NOT preserve positive definiteness if $c_k < 0$. In this case, there are several options for the rank one correction, but we will not address them here. Instead, we will describe the BFGS method, which almost always guarantees a positive-definite correction. Specifically:

$$B_{k+1} = B_k + b_k g_k g_k^T + c_k B_k d_k d_k^T B_k$$

where we have introduced the shorthand:

$$g_k = \nabla f(x_{k+1}) - \nabla f(x_k) \qquad \text{and} \qquad d_k = x_{k+1} - x_k$$

If we set:

$$b_k = \frac{1}{g_k^T d_k} \qquad \text{and} \qquad c_k = \frac{1}{d_k^T B_k d_k}$$

we satisfy the secant condition.

1.1.4 Nelder-Mead Simplex

While Newton's method is considered a 'second order method' (requires the second derivative), and quasi-Newton methods are first order (require only first derivatives), Nelder-Mead is a zero-order method. I.e. NM requires only the function itself - no derivatives.

For $f : \mathbb{R}^n \to \mathbb{R}$, the algorithm computes the values of the function on a simplex of dimension n, constructed from $n + 1$ vertices. For a univariate function, the simplex is a line segment. In two dimensions, the simplex is a triangle, in 3D, a tetrahedral solid, and so on.

The algorithm begins with $n+1$ starting points and then the follwing steps are repeated until convergence:

- Compute the function at each of the points
- Sort the function values so that
$$f(x_1) \leq ... \leq f(x_{n+1})$$
- Compute the centroid x_c of the n-dimensional region defined by $x_1, ..., x_n$
- Reflect x_{n+1} about the centroid to get x_r
$$x_r = x_c + \alpha(x_c - x_{n+1})$$
- Create a new simplex according to the following rules:
  - If $f(x_1) \leq f(x_r) < f(x_n)$, replace x_{n+1} with x_r
  - If $f(x_r) < f(x_1)$, expand the simplex through x_r:
$$x_e = x_c + \gamma(x_c - x_{n+1})$$
    If $f(x_e) < f(x_r)$, replace x_{n+1} with x_e, otherwise, replace x_{n+1} with x_r
  - If $f(x_r) \geq f(x_n)$, compute $x_p = x_c + \rho(x_c - x_{n+1})$. If $f(x_p) < f(x_{n+1})$, replace x_{n+1} with x_p
  - If all else fails, replace *all* points except x_1 according to
$$x_i = x_1 + \sigma(x_i - x_1)$$

The default values of α, γ, ρ and σ in scipy are not listed in the documentation, nor are they inputs to the function.

1.1.5 Powell's Method

Powell's method is another derivative-free optimization method that is similar to conjugate-gradient. The algorithm steps are as follows:

Begin with a point p_0 (an initial guess) and a set of vectors $\xi_1, ..., \xi_n$, initially the standard basis of $\mathbb{R}^n$.

- Compute for $i = 1, ..., n$, find λ_i that minimizes $f(p_{i-1} + \lambda_i \xi_i)$ and set $p_i = p_{i-1} + \lambda_i \xi_i$
- For $i = 1, ..., n-1$, replace ξ_i with ξ_{i+1} and then replace ξ_n with $p_n - p_0$
- Choose λ so that $f(p_0 + \lambda(p_n - p_0)$ is minimum and replace p_0 with $p_0 + \lambda(p_n - p_0)$

Essentially, the algorithm performs line searches and tries to find fruitful directions to search.

1.2 Solvers

1.2.1 Levenberg-Marquardt (Damped Least Squares)

Recall the least squares problem:

Given a set of data points (x_i, y_i) where x_i's are independent variables (in $\mathbb{R}^n$ and the y_i's are response variables (in $\mathbb{R}$), find the parameter values of β for the model $f(x; \beta)$ so that

$$S(\beta) = \sum_{i=1}^{m} (y_i - f(x_i; \beta))^2$$

is minimized.

If we were to use Newton's method, our update step would look like:

$$\beta_{k+1} = \beta_k - H^{-1} \nabla S(\beta_k)$$

Gradient descent, on the other hand, would yield:

$$\beta_{k+1} = \beta_k - \gamma \nabla S(\beta_k)$$

Levenberg-Marquardt adaptively switches between Newton's method and gradient descent.

$$\beta_{k+1} = \beta_k - (H + \lambda I)^{-1} \nabla S(\beta_k)$$

When λ is small, the update is essentially Newton-Gauss, while for λ large, the update is gradient descent.

1.2.2 Newton-Krylov

The notion of a Krylov space comes from the Cayley-Hamilton theorem (CH). CH states that a matrix A satisfies its characteristic polynomial. A direct corollary is that A^{-1} may be written as a linear combination of powers of the matrix (where the highest power is $n - 1$).

The Krylov space of order r generated by an $n \times n$ matrix A and an n-dimensional vector b is given by:

$$\mathcal{K}_r(A, b) = \operatorname{span}\{b, Ab, A^2 b, \ldots, A^{r-1} b\}$$

Thes are actually the subspaces spanned by the conjugate vectors we mentioned in Newton-CG, so, technically speaking, Newton-CG is a Krylov method.

Now, the scipy.optimize newton-krylov solver is what is known as a 'Jacobian Free Newton Krylov'. It is a very efficient algorithm for solving *large* $n \times n$ non-linear systems. We won't go into detail of the algorithm's steps, as this is really more applicable to problems in physics and non-linear dynamics.

1.3 GLM Estimation and IRLS

Recall generalized linear models are models with the following components:

- A linear predictor $\eta = X\beta$
- A response variable with distribution in the exponential family
- An invertible 'link' function g such that

$$E(Y) = \mu = g^{-1}(\eta)$$

We may write the log-likelihood:

$$\ell(\eta) = \sum_{i=1}^{m} (y_i \log(\eta_i) + (\eta_i - y_i) \log(1 - \eta_i)$$

where $\eta_i = \eta(x_i, \beta)$.
Differentiating, we obtain:

$$\frac{\partial L}{\partial \beta} = \frac{\partial \eta}{\partial \beta}^T \frac{\partial L}{\partial \eta} = 0$$

Written slightly differently than we have in the previous sections, the Newton update to find β would be:

$$-\frac{\partial^2 L}{\partial \beta \beta^T} (\beta_{k+1} - \beta_k) = \frac{\partial \eta}{\partial \beta}^T \frac{\partial L}{\partial \eta}$$

Now, if we compute:

$$-\frac{\partial^2 L}{\partial \beta \beta^T} = \sum \frac{\partial L}{\partial \eta_i} \frac{\partial^2 \eta_i}{\partial \beta \beta^T} - \frac{\partial \eta}{\partial \beta}^T \frac{\partial^2 L}{\partial \eta \eta^T} \frac{\partial \eta}{\partial \beta}$$

Taking expected values on the right hand side and noting:

$$E\left(\frac{\partial L}{\partial \eta_i}\right) = 0$$

and

$$E\left(-\frac{\partial^2 L}{\partial \eta \eta^T}\right) = E\left(\frac{\partial L}{\partial \eta} \frac{\partial L}{\partial \eta}^T\right) \equiv A$$

So if we replace the Hessian in Newton's method with its expected value, we obtain:

$$\frac{\partial \eta}{\partial \beta}^T A \frac{\partial \eta}{\partial \beta} (\beta_{k+1} - \beta_k) = \frac{\partial \eta}{\partial \beta}^T \frac{\partial L}{\partial \eta}$$

Now, these actually have the form of the normal equations for a weighted least squares problem.

$$\min_{\beta_{k+1}} \left(A^{-1} \frac{\partial L}{\partial \eta} + \frac{\partial \eta}{\partial \beta} (\beta_{k+1} - \beta_k)\right)^T A \left(A^{-1} \frac{\partial L}{\partial \eta} + \frac{\partial \eta}{\partial \beta} (\beta_{k+1} - \beta_k)\right)$$

A is a weight matrix, and changes with iteration - thus this technique is *iteratively reweighted least squares.*

1.3.1 Constrained Optimization and Lagrange Multipliers

Often, we want to optimize a function subject to a constraint or multiple constraints. The most common analytical technique for this is called 'Lagrange multipliers'. The theory is based on the following:

If we wish to optimize a function $f(x, y)$ subject to the constraint $g(x, y) = c$, we are really looking for points at which the gradient of f and the gradient of g are in the same direction. This amounts to:

$$\nabla_{(x,y)} f = \lambda \nabla_{(x,y)} g$$

(often, this is written with a (-) sign in front of λ). The 2-d problem above defines two equations in three unknowns. The original constraint, $g(xy,) = c$ yields a third equation. Additional constraints are handled by finding:

$$\nabla_{(x,y)} f = \lambda_1 \nabla_{(x,y)} g_1 + ... + \lambda_k \nabla_{(x,y)} g_k$$

The generalization to functions on $\mathbb{R}^n$ is also trivial:

$$\nabla_x f = \lambda \nabla_x g$$

```
In [ ]:
```