Homework 3: Improving Estimation by Nonlinear Least Squares

36-350, Fall 2012

Due at 11:59 pm on Thursday, 20 September 2012

INSTRUCTIONS: You know the drill by now.

Direct objective: Practice with writing and organizing functions.

Indirect objectives: Fitting statistical models; testing small pieces of the code before trusting them; translating math into code.

BACKGROUND: In the last lab, we estimated the parameter a in a nonlinear model,

$$Y = y_0 N^a + \text{noise} \tag{1}$$

by minimizing the mean squared error

$$\frac{1}{n}\sum_{i=1}^{n}(Y_i - y_0 N_i^a)^2 \tag{2}$$

We did this by approximating the derivative of the MSE, and adjusting a by an amount proportional to that, stopping when the derivative became small. Our procedure assumed we knew y_0. In this assignment, we will see how to estimate two parameters at once.

A function of one variable $f(x)$ is at an extremum when its derivative is zero, $df/dx = 0$. A function of multiple variables, say $f(x_1, x_2)$, is at an extremum when all the partial derivatives are zero, $\partial f/\partial x_1 = \partial f/\partial x_2 = 0$. Remember from calculus that the vector of partial derivatives of f, the **gradient** of f, is written ∇f. The direction of $\nabla f(x)$ is the direction in which f increases most rapidly when starting from x, and the magnitude of the gradient shows how quick that increase is. If x is an extremum, $\nabla f(x) = 0$. Since we want to make our function small, we will try to go *against* the gradient. The **gradient descent** or **steepest descent** method for minimizing a function starts with a guess $x^{(0)}$ and a scale factor r, and updates it by

$$x^{(t+1)} = x^{(t)} - r\nabla f(x^{(t)})$$

until ∇f is close to zero.

You will estimate the power-law scaling model by gradient descent.

1. (10) Write a function, called `mse()`, which calculates the mean squared error of the model in Eq. 1 on a given data set. `mse()` should take three

arguments: a numeric vector of length two, the first component standing for y_0 and the second for a; a numerical vector containing the values of N; and a numerical vector containing the values of Y. The function should return a single numerical value. The latter two arguments should have as the default values the columns `pop` and `pcgmp` (respectively) from the `gmp` data frame in Lab 2. Your function may not use `for()` or any other loop.

Hint: Look at the slides for Lecture 4.

2. (5) Check that, with the default data, you get the following values.

```
> mse(c(6611,0.15))
[1] 207057513
> mse(c(5000,0.10))
[1] 298459915
```

3. (20) Write a function, `mse.grad()`, which approximates the gradient of the mean squared error. It should take five arguments: a vector of length 2 giving the point at which we want the gradient; the increment for y_0; the increment for a; and the vectors containing the values of N and Y. Provide default values for everything except the first vector. `mse.grad()` should return a length two vector containing the gradient. This function must call your `mse()`.

4. (5) Check that you get the following values

```
> mse.grad(c(6611,0.15),10,1e-5)
[1] 1.650303e+05 1.429197e+10
> mse.grad(c(5000,0.10),7,-1e-5)
[1]    -109811.2 -7129496031.7
```

5. (30) Write a function, `plm()`, which estimates the parameters y_0 and a of the model (1) by minimizing the mean squared error, and minimizes the MSE by gradient descent. It should take the following arguments: an initial guess for y_0; an initial guess for a; a vector containing the N values; a vector containing the Y values; the increments for calculating the gradient; the scaling factor r; the threshold below which the gradient is considered effectively zero; and the maximum number of iterations. All arguments except the initial guesses should have suitable default values. It should return a list with the following components: the final guess for y_0; the final guess for a; the final value of the MSE; the final gradient; the number of iterations taken; a flag for whether the function stopped before running out of iterations.

   Your function must call those you wrote in earlier questions, and the appropriate arguments to `plm()` should be passed on to them.

   Hint: See the slides for lecture 4.

6. (10) What parameter estimate do you get when starting from $y_0 = 6611$ and $a = 0.15$? From $y_0 = 5000$ and $a = 0.10$? If these are not the same, why do they differ? Which estimate has the lower MSE? (You may want to experiment with different scale factors.)

7. (5) *Adjusting the step size* One problem with gradient descent is that the different parameters can have very different magnitudes, and have derivatives which differ wildly in size. This makes using the same step scale r for all of the parameters a problem. One way out is to give each parameter its own scaling factor[1]:

$$x^{(t+1)} = x^{(t)} - \begin{bmatrix} r_1 & 0 \\ 0 & r_2 \end{bmatrix} \nabla f(x^{(t)}) \tag{3}$$

   Write a new function, `plm2`, which which estimates the parameters y_0 and a of the model (1) by minimizing the mean squared error, and minimizes the MSE according to (3) rather than simple gradient descent. It should have almost all the same inputs as `plm`, except that it should take a vector of scaling factors, not just one. It should return the same things as `plm`.

8. (5) If both scale factors are the same, $r_1 = r_2$, the new method is just gradient descent. Check that `plm2` works by running it from the same starting positions as in 6, with both scale factors fixed to whatever you used there, and verifying that it gives the same results as `plm`.

9. (10) Set the scale factors to be 10^{-2} for y_0 and 10^{-12} for a. What are the estimates which `plm2` reaches from the two starting positions? Do they agree? Should they? Which of all of the estimates your code has produced seems the best, and why?

[1] Later on, we will see cleverer versions of this idea.