Statistics, Data Analysis, and Machine Learning for Physicists
Timothy Brandt
Spring 2020

Lecture 9

We'll start this class with a brief overview of minimization. I won't say anything about maximization: maximizing $f(\mathbf{x})$ is completely equivalent to minimizing $-f(\mathbf{x})$. Minimization of a multi-dimensional function is *hard*–there is generally no way of knowing that you have found the global minimium without an exhaustive brute-force search (and even then, are you *sure*?). Sometimes a function is analytic and you can be certain of finding the global minimum. Linear least-squares is a fortuitous example of this. In general though, you're in for a much harder slog. There are two general classes of algorithms to accomplish minimization:

- Deterministic methods that often need to start from a decent guess and find a minimum. They generally cannot get out of a local minimum and search parameter space more broadly.
- Stochastic methods that inject some amount of randomness in an attempt to explore the entire parameter space.

Both of these classes are available in python. For the second homework, methods in the first class should generally be sufficient; you can usually guess reasonable values of your parameters. You can still get stuck in local minima though, so to get out of them, you can construct your own hybrid of the two methods. For example, you can start a deterministic minimization routine from a random guess, and repeat the procedure with a range of random guesses. This is closely related to the basin hopping technique, described briefly below.

Deterministic minimization routines are available through the function `scipy.optimize.minimize`. You can use a variety of methods with the `method` keyword. You may also be familiar with `scipy.optimize.curve_fit`. The latter routine typically uses a method called *Levenberg-Marquadt* to find a minimum.

I'll briefly describe minimization in one dimension first. This is somewhat similar to the problem of finding a root of a function. If you have a continuous function and points x_1, x_2 such that $f(x_1)$ and $f(x_2)$ have opposite signs, you know that there is a root between x_1 and x_2. You can try to find the root by using the function's derivative (Newton's method), you can cut the interval in half to isolate the root to a smaller interval, or you can try something fancier. Newton's method is sufficiently useful that it's worth a two minute review here. You start with a guess at the zero of a function, call this guess x_0. You then fit a line to the function at x_0 using the derivative $f'(x_0)$:

$$f(x_0 - x) \approx f(x_0) - f'(x_0)\,(x - x_0)\,. \tag{1}$$

We want $f(x_0 - x) = 0$, so we solve to get our next guess for x, call it x_1, as

$$0 = f(x_0) - f'(x_0)(x_1 - x_0) \tag{2}$$

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}. \tag{3}$$

This converges quickly when you are close to a zero, but it can fail catastrophically, and if the function f has many roots, Newton's method might not find the zero you are looking for.

When you are minimizing a function the situation, and the choices, are very similar. There is one difference though: having two points doesn't guarantee a minimum in the middle no matter what the functional values are. You need three points, x_1, x_2, x_3, with $f(x_2) < f(x_1)$ and $f(x_2) < f(x_3)$. As for the root finding case, you can subdivide the interval. For example, choose a point x_4 between x_1 and x_2. If $f(x_4) < f(x_2)$, you have a new best point, and your minimum is between x_1 and x_2. Otherwise, your minimum is between x_4

and x_3. This is much like bisecting to localize a root. If you want something that converges faster (albeit with a risk of catastrophic failure, and difficulty distinguishing a maximum from a minimum), you can fit a parabola to x_1, x_2, and x_3. The vertex of this parabola will be your new guess for the minimum. Practical methods typically use a combination of these ideas.

Minimizing a function in many dimensions basically means minimizing it in one dimension, then another, then another, until you are done. You take a series of steps in an attempt to find the minimum. These steps need not be in the a single variable. For example, if you are minimizing a function of x and y, you are allowed to step in the direction $\hat{\mathbf{x}} + \hat{\mathbf{y}}$. You want to choose the directions and the step sizes to reach the minimum as quickly as possible. It might help to know in which directions your function is decreasing, so the gradient and Hessian can be useful. There are classes of methods that use these and classes that do not. I'll say a few words about a couple of different methods here:

- Levenberg-Marquadt: this is the default used by `scipy.optimize.curve_fit`. You might remember that when we took derivatives of χ^2, we just needed to know the derivatives of our fitting functions with respect to the model parameters. If we have these, then we have the Jacobean. Levenberg-Marquadt uses the specific form of the Jacobean in the least-squares problem to converge to the minimum. Unfortunately, you need to give it an initial guess (or it will use a default guess). If this default guess is close to the best fit, you will generally get there quickly. If it isn't, you might get somewhere completely different.

- Powell's method: I have found that this approach generally works well when I am trying to optimize a general function. It uses only the function, not its derivatives. Powell's method does a sequence of one-dimensional minimizations, and tries to choose these directions in such a way as to find the minimum as quickly as possible.

- Conjugate-gradient methods: These typically use the Jacobean to choose the directions along which to minimize. They are effective, and often fast, when you know the Jacobean, can compute it efficiently, and have a decent starting guess. The idea, in a sense, is to mimic the parabolic fitting in one dimension. The multidimensional analog is a quadratic form. You want to fit a quadratic form to the function and move to its vertex as your guess for the minimum.

- Quasi-Newton methods: (BFGS and variants are available in `scipy.optimize.minimize`, as are the previous two methods. Like the conjugate-gradient approach, you generally want to move toward lower values of the function, and fit a quadratic form when you get close.

Hopefully the analogy with one-dimensional optimization and root finding is at least a little bit clear. In one dimensional root-finding, you can bracket the solution and move a bit in the direction of zero by following the gradient. When you get close, Newton's Method will generally work well. In multi-dimensional optimization, bracketing is harder but still possible. The principles of minimization are then the same: move downhill, and when you get close to a minimum, fit a quadratic form (rather than a line, since the gradient is zero at a minimum) and move to, or in the direction of, the vertex.

In practice, you can try out these different methods. Some will be more suited to a particular problem than others. They all have a very important drawback though. **None of these guarantee that you will find the global minimum, the one you presumably want**. Local minima are the bane of function optimization. And when you are fitting a model with many free parameters, your goodness-of-fit function χ^2 will typically have many local minima. **It is something of a miracle that the linear, least-squares model has a unique local, and therefore global, minimum**.

I'll spend a bit of time now talking about stochastic methods. These can be good alternatives to deterministic methods when you want to make sure that you didn't get stuck in a local minimum. The simplest thing you can do is to pick a bunch of random starting points as guesses and apply a deterministic optimization routine to each one. In other words, throw a ton of guesses at the χ^2 surface and see which minima they roll into. Then pick the lowest minimum and hope it's the global minimum. In general, you will have no guarantee that your model is the global best fit–this is hard.

I'll discuss one other stochastic method in a little detail–simulated annealing. Simulated annealing is fairly closely related to Markov Chain Monte Carlo, something that we will cover later in the course. Basin hopping is a simple generalization of simulated annealing along the preceding idea of throwing a bunch of random guesses at your function. The core idea behind simulated annealing is to:

1. Propose a random step in parameter space;
2. Always keep the step if it improves the function; and
3. Sometimes keep it the step it makes the function worse.

This is the same basic strategy used in Markov Chain Monte Carlo. In simulated annealing, there is a temperature factor added, so that the probability of keeping a step that makes f larger is

$$p(\text{keep}) = \min\left(1,\, \exp\left[-\frac{f(\mathbf{x}_{\text{proposed}}) - f(\mathbf{x}_{\text{old}})}{T}\right]\right). \tag{4}$$

Higher temperatures mean that you are more likely to accept an unfavorable step and explore parameter space, while lower temperatures keep you moving toward the nearest local minimum. By alternately raising and lowering this temperature you can settle into a minimum and then get out of the minimum to look for another one. This heating/cooling procedure is similar to what is done for metals to get them to settle into a perfect crystalline structure ("annealing"), hence the name of the technique.

In python, simulated annealing is implemented through a function called `scipy.optimize.dual_annealing`. A related function, `scipy.optimize.basin_hopping`, makes one major change to the method outlined above. It replaces the cooling step, in which you settle into a local minimum, with a call to a deterministic minimization routine. In this way, basin hopping mimics a sort of random sampling of starting guesses for deterministic minimization. It then keeps the best minimum. Unfortunately, there is still no guarantee that this represents the global minimum.

We'll finish with some words about numerical issues. Your tools for much of the work in this class have been those of linear algebra. You have been solving and approximating matrix equations, and characterizing both your data and your model parameters with covariance matrices.

When you do floating point arithmetic on a computer, the computer only stores a limited number of decimal places. The typical data type you use is called *double precision floating point*, which uses 64 bits (0 or 1) to represent a number. One of these is allotted to the sign, 11 to the exponent, and the rest (52) to the base number. This means that if you add something smaller than 2^{-52} to a number (about 2×10^{-16}, it might not change its value. For example,

```
1 + 1e-17 = 1.0
1 + 1e-17 - 1 = 0.0
```

This sometimes causes problems when you are combining numbers of vastly different magnitudes. When you add and subtract in the context of matrix inversion, this can get you into trouble. Roughly speaking, you are in trouble when the **condition number** of your matrix, the ratio of the largest to the smallest singular value (eigenvalue for an invertible square matrix) is within spitting distance of machine precision.

Let's see how this can happen in practice. Suppose I want to fit a model

$$y_i = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 + a_6 x^6, \tag{5}$$

just a simple polynomial. Suppose also that x runs from 50 to 200. I can naïvely set up my matrix for SVD:

```
x = np.arange(50., 201)
y = x**1.5
A = np.zeros((len(y), 7))
```

```
for i in range(A.shape[1]):
    A[:, i] = x**i
```

This will give a solution vector and a set of residuals. The residuals won't look that good. In fact, they will be much better if I make the following simple change:

```
for i in range(A.shape[1]):
    A[:, i] = (x/100)**i
```

(putting the factor of 100 back in later). What happened? Well, I was combining a constant term with 200^6, which is getting a bit close to machine precision. The condition number of the matrix `A`, in the first definition, was about 10^{16}. In the second definition, it was about 10^5, which is now a good ten orders of magnitude away from roundoff error problems.

In general, when you fit functions, you are better off choosing orthogonal functions for this reason. If you are fitting polynomials, Legendre polynomials are a popular choice. You can see an example of that in the example Jupyter notebook supplied with these notes.