CS 4: Lecture 23
Monday, April 17, 2006

The Nelder-Mead Simplex Algorithm

This algorithm (not to be confused with the even more famous simplex algorithm for linear programming) is a local search algorithm like grid search, but improves upon it in several ways. First, it adapts its step sizes as it searches, and can make them longer as well as shorter. Second, it saves time by performing fewer evaluations of the objective function f per iteration.

A _simplex_ is a simple geometric shape defined by connecting together d + 1 vertices in d-dimensional space. For example, a 1D simplex is an edge; a 2D simplex is a triangle; a 3D simplex is a tetrahedron. You can have simplices of any dimension. If we're optimizing a function on d parameters, then we're searching in a d-dimensional parameter space, and our simplex has d + 1 vertices.

The Nelder-Mead simplex algorithm always maintains a set of d + 1 vertices, and the values of the objective function at those vertices. Each iteration repositions one (or occasionally most) of the vertices. The simplex must never become _degenerate_. That means no three vertices can lie on a common line; no four vertices can lie on a common plane; and so on.

We begin by choosing d + 1 vertices to start with. For example, if each parameter ranges from -5 to 5, we could try

```
x_0 = (0, 0, ..., 0),
x_1 = (1, 0, ..., 0),
x_2 = (0, 1, ..., 0),
...
x_d = (0, 0, ..., 1).
```

Evaluate the objective function f(x_i) at each vertex x_i, 0 <= i <= d. Let k be the index of the _worst_ vertex; that is, f(x_k) is as large or larger than the objective values at the other vertices.

The simplex algorithm repeats the following loop.

1 Let x_k be the worst vertex of the simplex, and let y be the centroid (average) of all the vertices _except_ x_k.

```
y = sum   x_i.
   i != k
```

2 Let z = 2y - x_k be the _reflection_point_ of x_k through the centroid. For example, in the following figure, x_2 is the worst vertex, y is the centroid of x_1 and x_3, and z is the reflection of x_2 through y.

```
z<-_    O x_3                             z O-------O x_3
    -_                                       \  new |\
      -_                                      \\    | \\
        y-_                                     \   |   \
           -_                                    \\ |    \\
             -_                                    \| old  \
        O      -O                                   O-------O
       x_1     x_2                                 x_1     x_2
```

3 If z is better than x_k--in other words, if f(z) < f(x_k)--then we replace x_k by z. In this manner, the simplex "walks" toward better and better positions in the parameter space. Notice that a reflection step replaces the simplex with a rotated version of the simplex, but doesn't change its shape.

However, if z is the _best_ vertex--in other words, if f(z) < f(x_i) for every vertex x_i of the simplex--then we might have found an unusually good direction, and we should see if the objective function will get even better if we continue in that direction. So we compute an _expansion_vertex_ w = 2z - y, and check the objective function at w.

```
w<-_                                w O--__
    -_                                 \_  --___
      -_                                 \_     --__
        z<-_    O x_3                      \_       --O x_3
            -_                               \_  new  |\
              -_                               \_     | \\
                y-_                              \_   |   \
                   -_                              \_ |    \\
                     -_                              \| old  \
                O      -O                             O-------O
               x_1     x_2                           x_1     x_2
```

If f(w) < f(z), we replace x_k with w (instead of z). Otherwise, we replace x_2 with z as usual. In the former case, the new simplex has a different shape than the old one--it's elongated in the successful search direction.

If either w or z is better than x_k, this iteration is over, and we start the loop from the beginning.

4 If z is not better than x_k, the far size of y might just be bad territory for searching. Instead, we compute a _contraction_vertex_ v = 0.5 (x_k + y), halfway between the worst vertex x_k and the centroid y. If f(v) < f(x_k), we replace x_k with v, end the iteration, and start the loop over.

```
O x_3                                     O x_3
                                          |\
                                          | \\
y                                         |   \
    v                                     |  _O_\
     -_                                   |_-new-\
O      -O                                 O-------O
x_1     x_2                              x_1     x_2
```

5 If we still haven't found a point better than x_k, we perform a _shrink_step_ like in the grid search algorithm. Let x_j be the _best_ vertex of the simplex--the one whose objective value f(x_j) is least. We shrink the simplex by moving every vertex halfway toward x_j--that is, we replace each x_i with 0.5 (x_i + x_j). The vertex x_j doesn't move, because it's the best vertex the search algorithm has ever found, and we don't want to forget it until we find a better one.

```
 O x_3                                     o
 |\                                        |
 | \\                                      v
 |   \                 ===>                O x_3
 |    \\                                   |\_
 |      \                                  |  \
 O-------O                                 O---O<--o
x_1     x_2                               x_1 x_2
```

The idea of the shrink step is that we're probably near a local minimum, so we refine the simplex so we're searching a smaller region around x_j.

The simplex algorithm terminates when the shortest edge of the simplex falls below a certain size.

If the search space is bounded (not infinite in all directions), the easiest way to prevent the simplex algorithm from searching outside the domain is to have f(x) return a very bad (large) value for any point x outside the domain.

An interesting property of this algorithm is that the expansion and contraction steps can make the simplex long and thin. In this way, the algorithm can adapt the simplex so its size along the different axes is tuned to the different parameters. This is particularly useful if the parameters have different units of measurement--for example, engine temperature, air/gasoline mixture, and engine RPMs. With grid search, you have to guess in advance how to scale the different axes to make the objective function "isotropic". But the simplex algorithm can often figure it out for you.

Combined Global & Local Search

An important observation is that local search algorithms like grid walk and simplex often find different local minima if you start them from different starting points. Global search algorithms like random search and grid search are good at finding starting points, but they're bad at homing in on local minima. So let's combine them. Here are some ways to do so.

- Use grid search (with a very coarse grid) to find a good starting point. Then run grid walk from there, with an initial step size of half the original grid size.

- Use grid search to find 20 good starting points. (20 is just a number; pick a number based on how long you're willing to wait.) You could take the 20 best points on the grid, but it's probably better to try to space the points out by choosing grid points that aren't near an even better grid point. Then do simplex search once from each of those 20 points, with each starting simplex being half the size of the grid. Choose the overall winner out of 20 runs of the simplex algorithm.