mcld.co.uk

blog

A very simple toy problem for matching pursuits

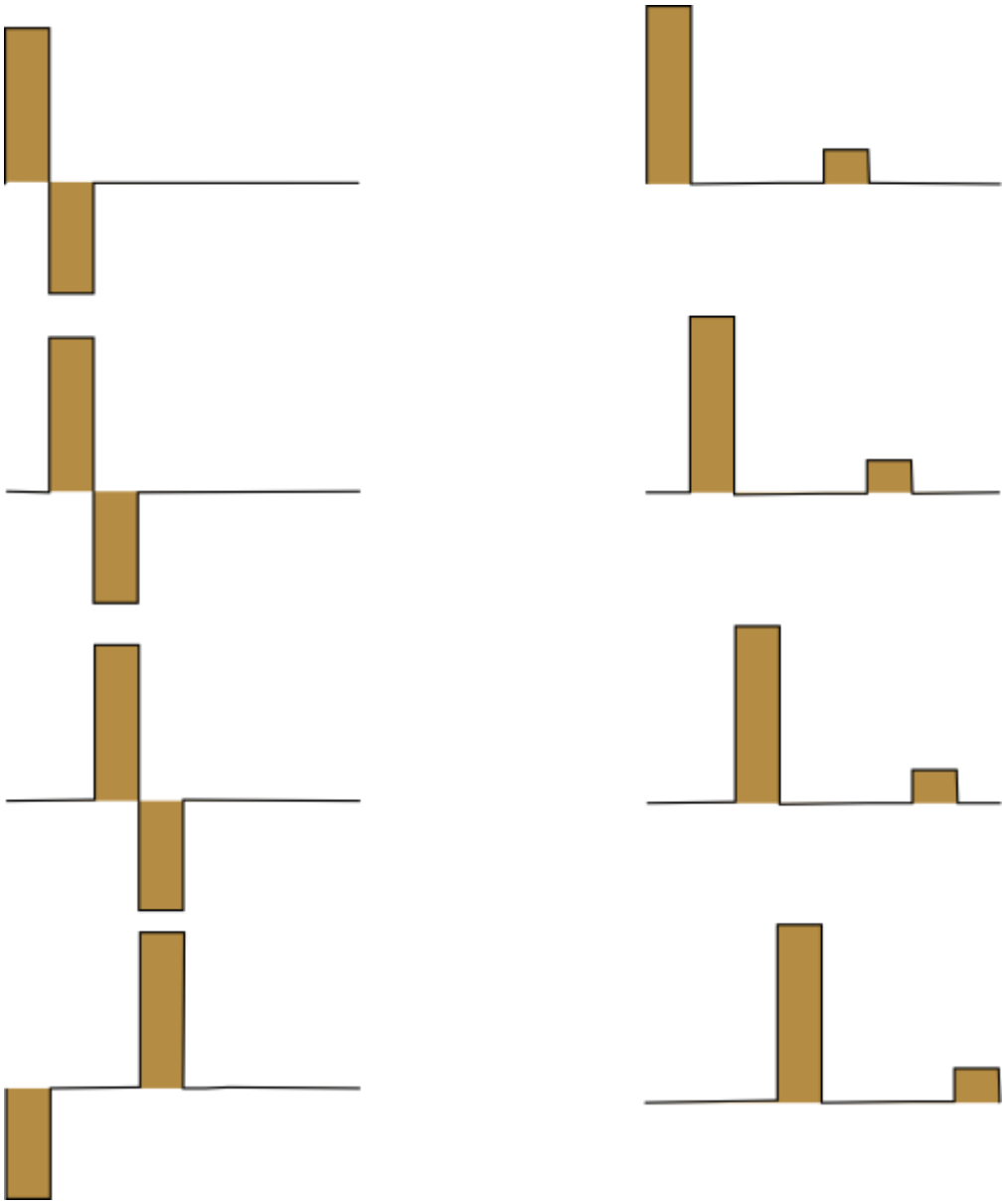
To help me think about how and why **matching pursuits** fail, here's a very simple toy problem which defeats matching pursuit (MP) and orthogonal matching pursuit (OMP). *[[NOTE: It doesn't defeat OMP actually - see comments.]]*

We have a signal which is a sequence of eight numbers. It's very simple, it's four "on" and then four "off". The "on" elements are of value 0.5 and the "off" are of value 0; this means the L2 norm is 1, which is convenient.

```
signal = array([0.5, 0.5, 0.5, 0.5, 0, 0, 0, 0])
```

Now, we have a dictionary of 8 different atoms, each of which is again a sequence of eight numbers, again having unit L2 norm. I'm deliberately constructing this dictionary to "outwit" the algorithms - not to show that there's anything wrong with the algorithms (because we know the problem in general is NP-hard), but just to think about what happens. Our dictionary consists of four up-then-down atoms wrapped round in the first half of the support, and four double-spikes:

```
dict = array([
   [0.8, -0.6, 0, 0, 0, 0, 0, 0],
   [0, 0.8, -0.6, 0, 0, 0, 0, 0],
   [0, 0, 0.8, -0.6, 0, 0, 0, 0],
   [-0.6, 0, 0, 0.8, 0, 0, 0, 0],
   [sqrt(0.8), 0, 0, 0, sqrt(0.2), 0, 0, 0],
   [0, sqrt(0.8), 0, 0, 0, sqrt(0.2), 0, 0],
   [0, 0, sqrt(0.8), 0, 0, 0, sqrt(0.2), 0],
   [0, 0, 0, sqrt(0.8), 0, 0, 0, sqrt(0.2)],
]).transpose()
```



BTW, I'm writing my examples as very simple Python code with Numpy (assuming you've run "from numpy import *"). We can check that the atoms are unit norm, by getting a list of "1"s when we run:

```
sum(dict ** 2, 0)
```

So, now if you wanted to reconstruct the signal as a weighted sum of these eight atoms, it's a bit obvious that the second lot of atoms are unappealing because the sqrt(0.2) elements are sitting in a space that we want to be zero. The first lot of atoms, on the other hand, look quite handy. In fact an equal portion of each of those first four can be used to reconstruct the signal exactly:

```
sum(dict * [2.5, 2.5, 2.5, 2.5, 0, 0, 0, 0], 0)
```

That's the unique exact solution for the present problem. There's no other way to reconstruct the signal exactly.

So now let's look at "greedy" matching pursuits, where a single atom is selected one at a time. The idea is that we select the most promising atom at each step, and the way of doing that is by taking the **inner product** between the signal (or the residual) and each of the atoms in turn. The one with the

highest inner product is the one for which you can reduce the residual energy by the highest amount on this step, and therefore the hope is that it typically helps us toward the best solution.

What's the result on my toy data?

- For the first lot of atoms the inner product is (0.8 * 0.5) + (-0.6 * 0.5) which is of course 0.1.
- For the second lot of atoms the inner product is (sqrt(0.8) * 0.5) which is about 0.4472.

To continue with my Python notation you could run "sum(dict.T * signal, 1)". The result looks like this:

```
array([ 0.1,  0.1,  0.1,  0.1,  0.4472136,  0.4472136,  0.4472136,  0.4472136])
```

So the first atom chosen by MP or OMP is definitely going to be one of the evil atoms - more than four times better in terms of the dot-product. (The algorithm would resolve this tie-break situation by picking one of the winners at random or just using the first one in the list.)

What happens next depends on the algorithm. In MP you subtract (winningatom * winningdotproduct) from the signal, and this residual is what you work with on the next iteration. For my purposes here it's irrelevant: both MP and OMP are unable to throw away this evil atom once they've selected it, which is all I needed to show. There exist variants which are allowed to throw away dodgy candidates even after they've picked them (such as "cyclic OMP").

NOTE: see the comments for an important proviso re MP.

Fri 13 July 2012 | **science** | **Permalink**

The blog is made using **Pelican**. It doesn't have a "comments" function so email me or tweet at me...