SDS 385 Exercise Set 01

Kevin Song

September 19, 2016

1 The Wolfe Conditions

The Wolfe conditions have two components: one makes sure that we have searched far enough, and the second makes sure that we do not search too far.

The condition that we search far enough is given by the first Wolfe condition, referred to by Nodecal and Wright as the *sufficient decrease* condition.

The sufficient decrease condition says that the step size taken must result in a decrease in the value of the function that would be greater than following some multiple of the derivative at that point, that is

$$f(\vec{x_0} + r\vec{s}) \leq f(\vec{x_0}) + c_1 \nabla f(\vec{x})^T \vec{s}$$

The *curvature condition* says that the interval must have a directional derivative along the search direction which is greater than some multiple of the original directional derivative along the search direction. This is used to rule out steps that are too small. Mathematically, this is expressed as:

$$\nabla f(\vec{x_0} + r\vec{s}) \geq c_2 \nabla f(\vec{x_0})^T$$

We can satisfy the first condition with an incredibly simple backtracking search that simply chooses the first step size in a geometric series that satisfies the sufficient decrease conditions. The code for this in python is provided below:

```python
import numpy as np

def backtracking_search(grad_func, obj_func, guess, searchDir):
    ”””
    Uses backtracking search to find a good step size.

    grad_func:  (function) calculates the gradient
    obj_func:   (function) calculates the objective value
    guess:      (cvector)  current guess (beta)
    searchDir:  (cvector)  direction to search along
    ”””

    ratio = 0.65 # How much do we decrease the step size each time?
    step = 0.5     # Initial trial value (almost certainly wrong)
    c1 = 1e 6     # Value suggested by book


    while obj_func(guess + searchDir * step) > \
          obj_func(guess) + c1 * step * np.dot(grad_func(guess).T, searchDir):

        step *= ratio

    return step
```

../code/backtrack.py

This code, even though it does not incorporate the curvature conditions, already represents a huge improvement from constant-size search. Whereas the previous search would take it's full 10000 steps without converging, with intelligent step sizes, steepest descent converges in under 300 iterations, even at a tolerance of 10^{-5}, using a ratio (upon step size failure) of 0.55.

Examination of the step sizes produced by the code shows that while most of the steps are the same order of magnitude, they vary by up to a factor of 8, which corresponds to 3 shortenings of the interval.

2 Quasi-Newton and BGFS

Computing the Hessian in Newton's method is a time-consuming task, and solving it is an $O(n^3)$ procedure.

Instead, we can create an approximation to the Hessian and solve with the approximation. If the approximation is done well, we can still get superlinear convergence for less cost.

By expanding the Taylor approximation near a local minimum of the objective function, we find that

$$\nabla f_{k+1}(x_{k+1} - x_k) \approx \nabla f_k + 1 - \nabla f_k$$

This suggests to us that for our approximation, B_{k+1}, to be valid, it should satisfy the rule

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

or, defining $s_k = x_{k+1} - x_k$, and $y_k = \nabla f_k + 1 - \nabla f_k$, this can be written

$$B_{k+1} s_k = y_k$$

The rule we use to generate the approximation is the Broyden-Fletcher-Goldfarb-Shanno method (BGFS), which states that B should be updated according to the following rule:

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k}$$

Now, in theory, we can solve the the equation $B_{k+1} s_{k+1} = y_{k+1}$ to find the search direction. However, in practice, since the Hessian changes at every iteration, we would need to spend $O(n^3)$ operations to factor it before solving. Since we are working with an approximation of the Hessian anyways, it is simpler to compute an approximation of the *inverse* of the Hessian.

If we define $H_k = B_K^{-1}$, the BGFS update rule becomes

$$H_{k+1} = \left(I - \rho_k s_k y_k^T\right) H_k (I - \rho_k y_k s_k^T) + \rho_k s_k s_k^T$$

where $\rho_k = 1/y_k^T s_k$.

The code for BFGS can be found in `code/bfgs.py` in the top-level of this exercises directory. It is not listed here for brevity.

The code uses a few iterations of steepest descent to burn in the Hessian (until the condition number is sufficiently small), then uses the computed inverse along with BGFS to do updates.

This code converges incredibly quickly: within 7 iterations, it converges. The objective function values match with previous solutions, suggesting that this is not a spurious convergence.