Quasi-Newton Methods: L-BFGS

Giri Iyengar

December 10, 2014

Quasi-Newton Methods: L-BFGS

- Often, in Data Analytics, you end up with a function minimization problem
- You have to find the *minimum* or *maximum* of a function in a neighborhood
- E.g. Least-squares fit for a given set of data. You are minimizing the sum of squared errors

Newton's method for function approximation

- Most optimization algorithms are iterative in nature
- Newton's method is one such. You start with an initial guess and successively refine it
- Suppose you have a guess x_n. Now, the goal is to find x_{n+1} such that $f(x_{n+1}) < f(x_n)$

Newton's method

- Assume $x \in R^n$ and $f(x) : R^n \rightarrow R$
- Assuming $f(x)$ is twice-differentiable (i.e. has first and 2nd derivatives), let's do Taylor series expansion
- $f(x + \Delta x) \approx f(x) + \Delta x^T \nabla f(x) + \frac{1}{2} \Delta x^T (\nabla^2 f(x)) \Delta x$
- $\nabla f(x)$ is the *Gradient* vector and $\nabla^2 f(x)$ is the *Hessian* matrix

Newton's method

- Rewriting in terms of a function of Δx, for the n-th iteration, we get
- $h_n(\Delta x) = f(x_n) + \Delta x^T g_n + \frac{1}{2}\Delta x^T H_n \Delta x$
- g_n is the gradient at x_n and H_n is the Hessian at x_n
- We want to choose that Δx which minimizes this local expansion
- $\frac{\partial h_n(\Delta x)}{\partial \Delta x} = g_n + H_n \Delta x$
- This suggests that the optimal step is: $\Delta x = -H_n^{-1} g_n$
- In practice, we set $x_{n+1} = x_n - \alpha(H_n^{-1} g_n)$

Iterative implementation of Newton's method

- Start with an initial guess of x
- Compute g_n and H_n^{-1} at that x
- Estimate the next x, with a suitable learning rate, α
- Iterate till the function is converged. If function is *convex*, we are guaranteed a global minimum

Problem with Newton's method

- Most routinely encountered Data Analytics problems have dozens to hundreds of variables
- Hessian quickly becomes huge
- Computing the inverse of the Hessian at each iteration becomes quite expensive
- Luckily, the technique still works even if you use a *approximate* Hessian. Can we construct a cheap approximate Hessian?
- Notice that instead of Hessian, if we substitute with an identity matrix, we get plain old **Gradient** descent

BFGS method

- Broyden, Fletcher, Goldfarb, and Shanno *independently* discovered this algorithm
- Directly computes an approximate H_n^{-1}
- Computationally efficient as you don't have to invert any matrices

BFGS method

- We want an approximate Hessian such that
- It is invertible (any positive semi-definite matrix)
- The gradient of h_n matches the real gradient of the function
- That is $\nabla h_n = g_n$ and $\nabla h_{n-1} = g_{n-1}$
- We are choosing a Hessian such that the gradients of the quadratic approximation agree at the various iteration steps
- This might result in a good approximation of the actual Hessian

BFGS method

- Turns out, this results in: $H_n^{-1}(g_n - g_{n-1}) = x_n - x_{n-1}$
- Let $y_n = g_n - g_{n-1}$ and $s_n = x_n - x_{n-1}$
- The *best*, in a least-squares sense, H_{n+1}^{-1} turns out be determined completely by H_n^{-1}, y_n, and s_n
- This gives us an iterative algorithm. Start with any H_0, say Identity matrix and successively refine H_n in terms of all the previous Hessian approximations

L-BFGS method

- Limited memory variant of BFGS
- Only retain m past values of y_n and s_n
- Relatively fast
- However, for really large data sets, online methods such as SGD often perform better than L-BFGS

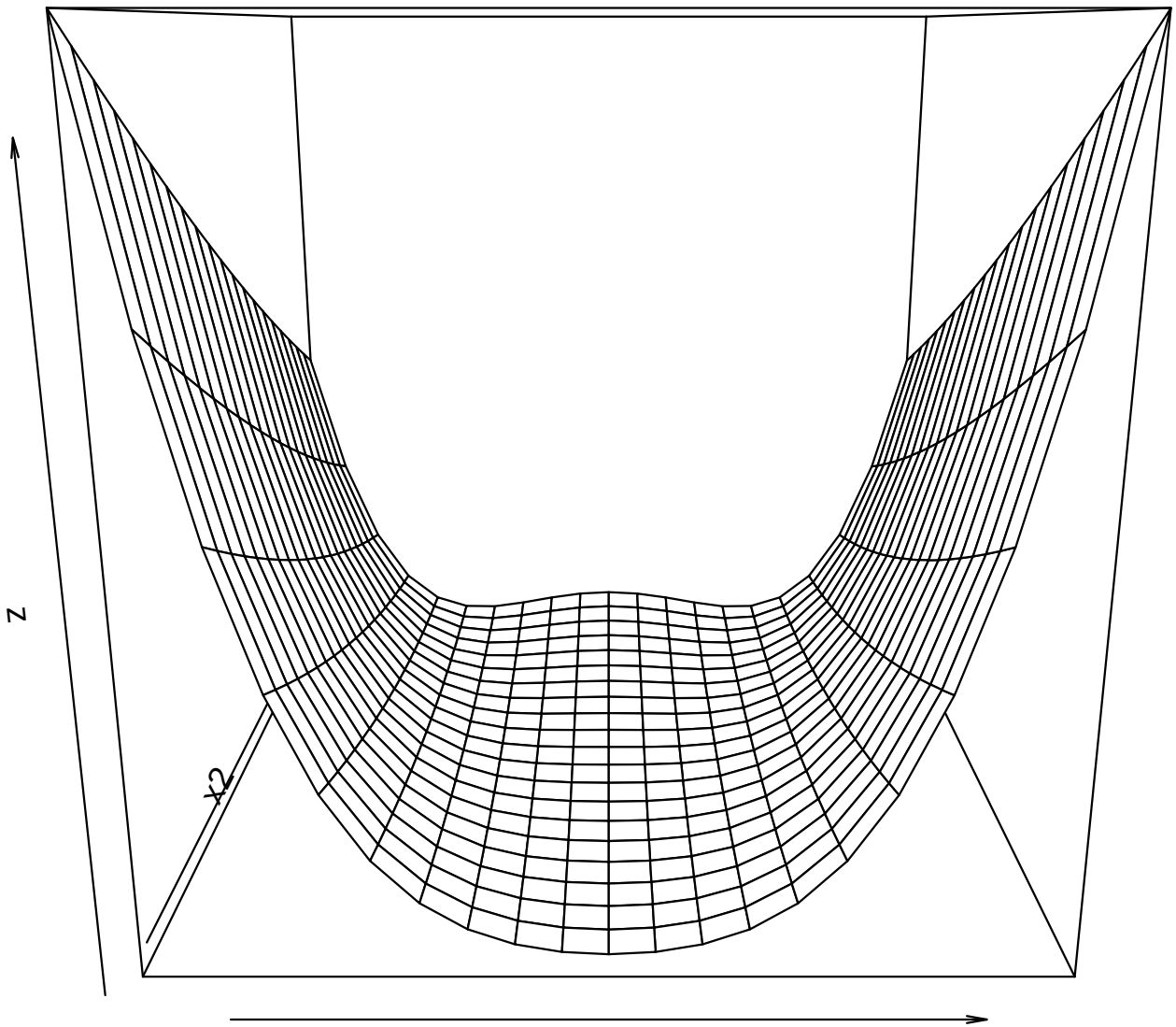
L-BFGS example in R

```
library(grid)
library(graphics)
objective <- function(x)
  {100*(x[2]-x[1]^2)^2 + (1 - x[1])^2}
gradient <- function(x)
  {c(-400 * x[1] * (x[2] - x[1]^2) - 2 * (1 - x[1]),
    200 * (x[2] - x[1]^2))}

x1 <- seq(-5,5,0.5); x2 <- seq(-5,5,0.5)
xygrid <- expand.grid(x1=x1,x2=x2)
z <- matrix(data=objective(xygrid),
            nrow=length(x1),ncol=length(x2))
```

Objective function to be optimized by L-BFGS

```
persp(x1,x2,z)
```



L-BFGS Solution of the minima

```
library(lbfgs)
out.lbfgs <- lbfgs(objective, gradient, c(-10, 10),invisible=1
out.lbfgs$value
```

```
## [1] 1.596243e-15
```

```
out.lbfgs$par
```

```
## [1] 1.0000000 0.9999999
```

```
# Contrast this with gradient descent via CG
out.optim1 <- optim(c(-10, 10), objective, gradient,method="CG
out.optim2 <- optim(c(-1, 1), objective, gradient,method="CG",
```

Conjugate Gradient Solution

```
out.optim1$par
```

```
## [1] -2.752043  7.580342
```

```
out.optim2$par
```

```
## [1] 1.036727 1.074907
```

```
out.optim2$counts
```

```
## function gradient
##     3997     1001
```

L-BFGS Solution for the minima

- When we examine this objective function we notice the following
- L-BFGS converges to the correct solution in about 60 iterations. You can see this by removing the parameter: **invisible** from lbfgs invocation
- Conjugate Gradient (CG) method doesn't converge in more than 100 iterations
- In the case where we start much closer to the real solution $(1, 1)$, CG comes close after 1000 iterations but it is not quite there