Chapter_8

Henry Bourne

2022-12-12

Numerical Optimization

An optimization problem can be described as trying to find the best solution from all feasible solutions. Generally an optimization problem is either discrete or continuous, we will focus on the latter and in the case where we are trying to minimize (as opposed to maximize) a function. We will also mainly discuss non-constrained optimization, however, we note that the L-BFGS-B method in *optim* (which we will discuss) can deal with **box constraints** (where the only constraints are upper and lower bounds on the parameters we are optimizing). Also note we can use the constrOptim function to deal with **constrained optimization** problems.

One-Dimensional Optimization

We will start by looking at how we can perform one-dimensional optimization. The R function **optimize()** can be used to carry out one-dimensional optimization. It uses **golden section search** which works by dividing the search domain into sections according to the golden ratio and uses this to find the minimum. More specifically, the golden ratio splits the domain into three, it then evaluates the function at the four points that define the sections (Left-most value, left of middle section, right of middle section and right-most value). If the minimum is at the left-most or left of the middle section we make the left-most value to the right of the middle our new domain where the left-most value remains the left-most value, the right of the middle becomes the right-most value and the left of the middle value becomes the right of the middle and we select a new left of the middle value. We do something similar if the right of the middle value or the right-most value are the minimum. This optimization method works best for a uni-modal function (one-dimensional) where the minimum is within the search interval (obviously). It is not guaranteed to converge and we can set a termination condition (by using the *tol* parameter which will stop searching once this tolerance is reached) which uses the gaps between the section values, however, an advantage of this method is we don't need to compute any derivatives.

Another method we could use is **Newtons method**. Newtons method aims to find the root of some function, g, by assuming it has a Taylor expansion at some value x_0, $g(x) \approx g(x_0) + g'(x_0)(x - x_0)$. If we set $x = 0$ we get $x = x_0 - \frac{g(x_0)}{g'(x_0)}$, we can then use this expression to find our next "best guess" for what value of x we have $g(x) = 0$. So in the context of trying to find the minimum of some function f we use Newtons method to solve for $f(x) = 0$. To carry out Newtons method we can use the help of the **D()** function to compute (one-dimensional) derivatives (it computes derivatives symbolically, ie. analytically, and algorithmically, ie. numerically). eg.

```r
f_sym <- expression(cos(x) + cos(2*x) + sin(3*x))
f1_sym <- D(f_sym, 'x'); f1 <- function(x) eval(f1_sym)
f2_sym <- D(f1_sym, 'x'); f2 <- function(x) eval(f2_sym)
x <- 2
for (i in 1:6) {
  x <- x - f1(x) / f2(x); cat(x, '\n')
}
```

```
## 1.371591
## 1.686732
## 1.647531
## 1.648062
## 1.648062
## 1.648062
```

Here we quickly converge on a local minimum. We could also use the *newton.method()* function from the *animation* package. Note that we can also use Newtons method in the multi-variate case. Problems associated with newtons method are that it can converge to local maxima aswell as minima and that we need to calculate the second derivative.

Multi-Dimensional Optimization

Most optimization problems we come across will be multi-dimensional, so what are some good multi-dimensional optimization techniques? and how do we implement them in R?

- We can divide common optimization techniques into three categories:
- Simplex methods: where we only use the value of the function.
- Gradient type methods; where we use the value of the function and it's gradient.
- Newton type methods: which use the value of the function, it's gradient vector and its second derivitive (it's hessian).

We've already seen a Newton type method (Newtons method) and a simplex method (golden section search) in the one dimensional case. In R there are two main functions used for multi-dimensional optimization. There is *nlm()* (which stands for non-linear minimization) which is a Newton-type algorithm and *optim* which has a selection of algorithms we can use including: Nelder-Mead, BFGS (Broyden, Fletcher, Goldfarb and Shanno), CG (conjugate gradient), L-BFGS-B (Byrd et al's version of BFGS that has low memory requirements and allows box constraints), SANN (simulated annealing), and Brent (which uses the one-dimensional *optimize* function discussed in the previous section).

Simplex methods

The **Nelder-Mead** algorithm, which can be implemented by using *optim* with *method="Nelder-Mead"*, also known as the downhill simplex method is the most well-known simplex method. It works by comparing values of the function at various points and a big advantage of using it is that it doesn't require any derivatives. However, it's a heuristic method, it only converges to local minima and may converge to non-stationary points and can be very slow. It works by maintaining a simplex, where we evaluate the vertices of the simplex and aim to replace the worst vertex with a new point which depends on the value of the worst point and the center of the other vertices. In particular we maintain a simplex of size n+1, where we are working in $\mathbb{R}^n$ space. We update our worst vertex by using a set of rules that determine how to reflect, expand, contract or shrink the simplex and we do this until we satisfy the local minimum conditions.

Gradient type methods

The most well-know gradient type method is **Steepest Descent/ Gradient Descent (GD)**, this method works by using the derivative of the function we are trying to minimize to guide our "direction" in the parameter space. Mathematically we iteratively take $x \leftarrow x - \alpha \cdot \nabla f(x)$ where α is the step-size and is a hyper parameter which we select. What we are doing here is using the derivative to tell us the direction in the parameter space which will lead to the largest decrease in the function based on the value of our function for our current value for the parameters. To select α we can either just set it to some constant or use a line-search algorithm. The drawback of this method is that it is relatively slow, can "zig-zag" in the parameter space and be computationally inefficient. A variation on this method is called **Stochastic Gradient Descent (SGD)** which works in a similar way and is useful in scenarios where we are trying to minimize some loss function that measures how well a model fits some data. It works in the same way, however, it only compute the derivative of the loss function based on a sample of the data as opposed to the whole dataset. It's advantages over GD include that it is more computationally efficient (as we only need to compute the derivative of the loss function over some sub-sample of the data) and is more robust in regards to local minima (there is a chance that the stochasticity will "bump" it out of a local minima). Disadvantages include that it "zig-zags" even more than GD and can take many more iterations. To implement GD or SGD we can use the **deriv()** function which uses the D function we mentioned earlier to compute all the partial derivatives (as we are in the multi-variate case here). We can also use the help of the *gradDescentR* and *sgd* R packages.

Another gradient type method is the **Conjugate Gradient (CG)** method, the main idea is that the search direction at each step should be conjugate toward search directions in the previous step. We can use it in the *optim* function by selecting the CG method. By conjugate we mean the direction vectors for each step are orthogonal to one another in respect to an inner-product (usually the dot product). This prevents the "zig-zag" effect we mentioned occurred in GD (and SGD). We have that if the function we are trying to minimize is quadratic that the CG method is guaranteed to reach the minimum in exactly n steps if the input to the function we are trying to minimize is in $\mathbb{R}^n$. It tends to perform better than GD even on general non-linear functions, but not as well as Newton-type methods. Note that this method and all gradient type methods require first derivatives (its in the name!).

Newton type methods

We discussed newtons method earlier in the one-dimensional case earlier, but what about in the multi-dimensional case? In the multi-dimensional case we update x as follows: $x \leftarrow x - H(f(x))^{-1}\nabla f(x)$, where $H(f(x))$ is the hessian of f. To compute the hessian and its inverse can be very expensive in multi-dimensional problems, as the hessian may be very large. **Quasi-Netwon methods** aim to fix this problem. The most common Quasi-Newton method is **Broyden, Fletcher, Goldfarb and Shanno (BFGS)** method which works by replacing the inverse Hessian with an estimate. It uses the current gradient and difference between the current and previous estimates of the solution to calculate a rank-2 update of the Hessian approximation. A rank-2 update is calculated as follow: we first compute the difference between the two gradient vectors, denote this as dg, then calculate the difference between the current and previous estimates of the solution, denote this as dx, we then compute a scalar value $a = \frac{dg \cdot dx}{dg \cdot H \cdot dg}$ (where H is our current approximation of the Hessian matrix), we then update our Hessian as follows $H \leftarrow H + a \cdot dx \cdot dx^T - (H \cdot dx)(dx^T \cdot H)/(dg \cdot H \cdot dg)$. We can use the BFGS method by using the *optim* function with the method set to BFGS. The disadvantage of this method is that it stores a dense matrix (the hessian approximation) aswell as the current and previous gradients and solutions to the function which can take up alot of memory if we are working in a space with high-dimension.

The low-memory version of BFGS is L-BFGS which reduces memory by only storing a few vectors that represent the approximate Hessian. We can use it by again using *optim* but selecting the L-BFGS method.

Simulates Annealing

Unlike the three type of methods discussed so far **Simulated Annealing (SA)** find the global minimum (as opposed to just a local minimum) and only uses the value of the function (like the simplex methods) meaning no derivatives are required. However, it is very slow and actually often fails to find the global minimum. It is a heuristic method that is based on the physical process of annealing which is where a material is slowly cooled (after being heated up) in order to reduce its defects and increase its structural integrity. The way it works is by iteratively choosing some point (by adding a small random perturbation to the current point based on some step-size), evaluating it and then calculating the increase or decrease in "energy". Typically we calculate the energy by simply calculating the difference between the value given by the function at the two points. If the proposed step will decrease the energy then we accept it, if it doesn't decrease the energy we accept it with some probability p, we then decrease p over time. What this means is we start by exploring the parameter space and then as time goes on become more exploitative. In the *optim* function we can use the **SANN** method: a simulated annealing method that uses the Metropolis function for the acceptance probability and where the next step proposal is generated from a Gaussian Markov kernel with scale proportional to the temperature (the acceptance probability).

Conclusion

Generally speaking Newton-type methods perform better than gradient type methods, with CG performing better than GD but worse than Newton-type methods. Although simulated annealing methods make bold-claims about finding the global minima they tend to be lackluster when actually used and the fact that other optimization methods are widely used says something about the effectiveness of these methods (ie. they can't be that effective otherwise we wouldn't use other methods).