Neighbourhood Functions for Local-Search Algorithms

Package version 0.1-0

Enrico Schumann
es@enricoschumann.net

The function `neighbourfun` constructs a neighbourhood function, i.e. a function that maps a given solution into a randomly-chosen neighbour. This vignette provides several examples of how the function can be used. We start by attaching the package and setting a seed.

```
> library("neighbours")
> set.seed(3477)
```

In the examples that follow, we will use a simple optimisation algorithm, a (stochastic) Local Search. If package `NMOF` is available, function `LSopt` from that package is used. If not, we use a simple replacement, taken from Gilli et al. [2019, Chapter 13].

```
> LSopt <- if (requireNamespace("NMOF")) {
               NMOF::LSopt
           } else
               function(OF, algo = list(), ...) {
                   xc  <- algo$x0
                   xcF <- OF(xc, ...)
                   for (s in seq_len(algo$nI)) {
                       xn <- algo$neighbour(xc, ...)
                       xnF <- OF(xn, ...)
                       if (xnF <= xcF) {
                           xc  <- xn
                           xcF <- xnF
                       }
                   }
                   list(xbest = xc, OFvalue = xcF)
               }
```

Example: Selecting elements of a list

We are given a numeric vector y and also a matrix X, which has as many rows as y has elements. The aim now is to find a subset of columns of X whose average is as lowly correlated with y as possible. Let us create random data.

```
> ny <- 50     ## length of y, number of rows of X
> nx <- 500    ## number of columns of X
> y <- runif(ny)
> X <- array(runif(ny * nx), dim = c(ny, nx))
```

We'll try a (stochastic) Local Search to compute a solution. There may be other, perhaps better heuristics for the job. But a Local Search will compute a good solution, as we will see; and it is simple, which is a good idea for an example. See Gilli et al. [2019, Chapter 13] for a tutorial on Local Search.
Suppose we want a solution to include between 10 and 20 columns. A valid candidate solution would be the first 15 columns of X.

```
> x0 <- logical(nx)
> x0[1:15] <- TRUE
> head(x0, 20)
```

```
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
[10]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
[19] FALSE FALSE
```

It's probably not a very good solution. We write an objective function to compute the actual quality of the solution `x0`.

```
> column_cor <- function(x, X, y)
      cor(rowMeans(X[, x]), y)
```

With this objective function we can evaluate the quality of `x0`.

```
> column_cor(x0, X, y)
```

```
[1] -0.0621
```

To run a Local Search, we need a neighbourhood function. Calling `neighbourfun` will create such a function, taking as inputs our constraints:† at least 10, no more than 20 columns.

† Gentleman and Ihaka [2000] show that R's scoping rules are particularly convenient for creating the ingredients of optimisation, such as the objective function or, as here, a neighbourhood function.

```
> nb <- neighbourfun(type = "logical", kmin = 10, kmax = 20)
```

It remains to run the Local Search.

```
> sol.ls <- LSopt(column_cor,
                  list(neighbour = nb,
                       x0 = x0,      ## initial solution
                       nI = 3000,    ## number of iterations
                       printBar = FALSE),
                  X = X, y = y)
```

```
Local Search.
Initial solution:  -0.0621
Finished.
Best solution overall: -0.824
```

Let us evaluate the final solution.

```
> column_cor(sol.ls$xbest, X, y)
```

```
[1] -0.824
```

And we check the constraints: how many columns are in the solution?

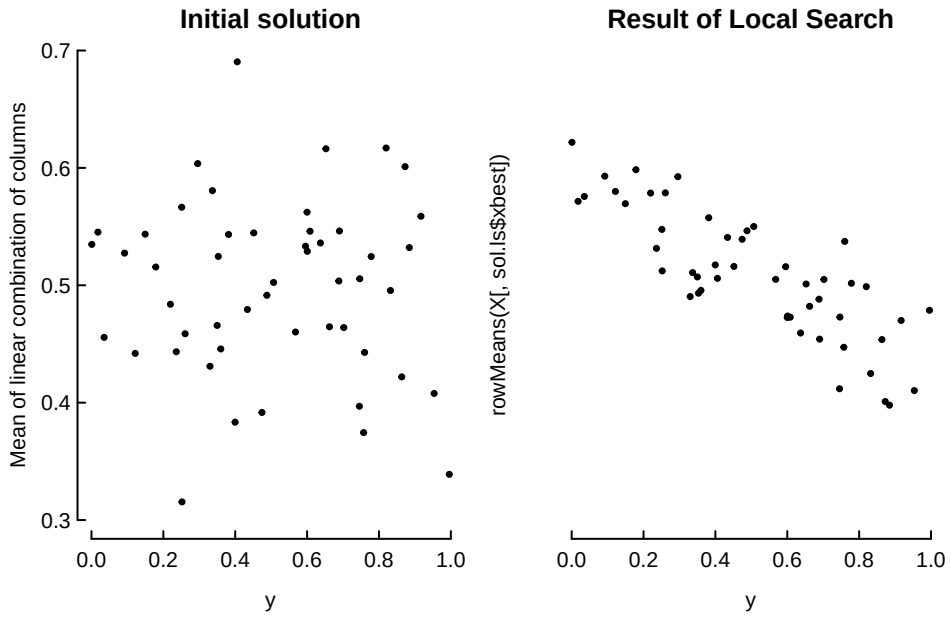
```
> sum(sol.ls$xbest)
```

```
[1] 20
```

We can visualise the initial and the final solution. The negative correlation is clearly visible.

```
> par(mfrow = c(1, 2), las = 1, bty = "n",
      mar = c(3, 3, 1, 0.5), mgp = c(1.75, 0.25, 0),
      tck = 0.02, cex = 0.7)
> plot(y, rowMeans(X[, x0]),
       main = "Initial solution",
       pch = 19, cex = 0.5,
```

```
       ylim = c(0.3, 0.7),
       xlab = "y",
       ylab = "Mean of linear combination of columns")
> par(yaxt = "n")
> plot(y, rowMeans(X[, sol.ls$xbest]),
       main = "Result of Local Search",
       pch = 19, cex = 0.5,
       ylim = c(0.3, 0.7),
       xlab = "y")
> axis(4)
```



More restrictions

The neighbourhood function we used in the previous section included constraints: it would include no fewer than 10 or no more than 20 `TRUE` values. Note that the neighbourhood function required a valid `x` as input.[†] We may also set `kmin` and `kmax` to the same integer, so the number of `TRUE` values is fixed. (In this case, a slightly-different algorithm will be used.)

We can also add a constraint about elements not to touch. Suppose the initial solution to a model (not the example we use previously) is a logical vector of length 9.

[†] For invalid `x`, the result is undefined. Neighbourhood functions should not check the validity of their inputs, because of speed: the functions are called thousands of times during an optimisation run, and so every fraction of a second matters.

```
> x <- logical(9L)
> x[4:6] <- TRUE
> compare_vectors(x)
```

```
000111000
```

We restrict the changes that can be made to the solution: the first three elements must not be touched; only the remaining elements may change, i.e. are `active`.

```
> active <- c(rep(FALSE, 3),
              rep(TRUE, length(x) - 3))
> active
```

```
[1] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
> nb <- neighbourfun(type = "logical", kmin = 3, kmax = 3, active = active)
```

Let us take a few random steps: the function will never touch the first three elements.

```
000111000
     |  |
000110001
    | |
000100101
   | |
000001101
    | |
000011001
      | |
000011100
   ||
000101100
    ||
000110100
     ||
000111000
     |  |
000110001
    ||
000101001
```

Another example: minimising portfolio risk

Suppose we are given a matrix R and aim to find a vector x such that the variance of the elements in the product Rx is minimised. This is a common problem in finance, in which R could be a matrix of asset-price returns, with each column holding the returns of one asset, and x a vector of portfolio weights.
We'll solve this problem, as in the previous example, with Local Search. The solution now is a numeric vector x. Again we need two functions – the objective function and the neighbourhood function – to find the optimal (or at least a good) vector.
Start with the objective function. For this particular goal – the variance of returns –, we can first compute the variance–covariance matrix Σ of R, and then minimise $x'\Sigma x$. That is, we could write an objective function as follows:

```
> variance <- function(x, S, ...)
      x %*% S %*% x
```

(In the code, `S` stands for the variance–covariance matrix Σ.)
An alternative way to write the objective function is the following.

```
> variance2 <- function(x, R, ...)
      var(R %*% x)
```

The disadvantage of the second version is efficiency: R might have many rows, and then computing the product Rx in every iteration would be more expensive than using S, which essentially is the crossproduct of R. But as we shall see below, this inefficiency can be remedied.
Suppose we start with an equal-weight portfolio.

```
> if (!requireNamespace("NMOF")) {
      R <- array(rnorm(120*20, sd = 0.03), dim = c(120, 20))
  } else
      R <- NMOF::randomReturns(na = 20, 120, 0.03, rho = 0.6)
> S <- cov(R)  ## Sigma
> x0 <- rep(1/ncol(R), ncol(R))
```

When the argument `type` is set to `numeric`, the resulting neighbourhood function will randomly select elements and change them slightly by adding or subtracting real numbers. The size of those numbers is controlled by argument `stepsize`, and with argument `random` set to `TRUE` (the default), the step sizes will vary randomly. When argument `sum` is `TRUE`, the function will add and subtract from chosen elements in such a way that the sum over all elements remains unchanged. (That is a typical restriction in portfolio-selection models.)

```
> nb <- neighbourfun(type = "numeric",
                     min = 0, max = 0.2,
                     stepsize = 0.005)
```

We can solve this problem with Local Search with the first version of the objective function.

```
> sol.ls <- LSopt(variance,
                  list(x0 = x0,
                       neighbour = nb,
                       nI = 2000,
                       printBar = FALSE),
                  S = S)
```

```
Local Search.
Initial solution:  0.000651
Finished.
Best solution overall: 0.000561
```

```
> sol.qp <- if (requireNamespace("NMOF"))
                round(NMOF::minvar(S, wmin = 0, wmax = 0.2), 2) else NA
> data.frame(LS = round(sol.ls$xbest, 2)[1:10],
             QP = sol.qp[1:10])
```

```
     LS   QP
1  0.00 0.00
2  0.00 0.00
3  0.00 0.00
4  0.12 0.12
5  0.20 0.20
6  0.02 0.02
7  0.00 0.00
8  0.11 0.11
9  0.04 0.04
10 0.00 0.00
```

When we feed the second version of the objective function to `LSopt`, we arrive at the same solution.

```
> sol.ls2 <- LSopt(variance2,
                   list(x0 = x0,
                        neighbour = nb,
```

```
                    nI = 2000,
                    printBar = FALSE),
                R = R)
```

```
Local Search.
Initial solution:  0.000651
Finished.
Best solution overall: 0.000561
```

```
> data.frame(LS2 = round(sol.ls2$xbest, 2)[1:10],
             QP = sol.qp[1:10])
```

```
    LS2   QP
1  0.00 0.00
2  0.00 0.00
3  0.00 0.00
4  0.12 0.12
5  0.20 0.20
6  0.02 0.02
7  0.00 0.00
8  0.11 0.11
9  0.04 0.04
10 0.00 0.00
```

This second objective function is, as described above, less efficient than the first. But it is much more flexible: only for minimising variance could we take the shortcut via the variance–covariance matrix. But for other measures of risk, we cannot do that. One example is the so-called semi-variance, defined as

$$\frac{1}{m}\sum_{i=1}^{m}\min(R_i x - \frac{1}{m}\iota'(Rx), 0)^2 \tag{1}$$

All we have to do now is to exchange objective functions, and Local Search will find the corresponding portfolio.

```
> semivariance <- function(x, R, ...) {
      Rx <- R %*% x
      Rx.lower <- pmin(Rx - mean(Rx), 0)
      sum(Rx.lower)
  }
```

Updating

In the example in the previous section, but also in many other cases when doing data analysis, the solution x is used to compute a matrix/vector product Ax, in which A is a m times n matrix, and x is of length n.
If we change only few elements in the solution, then we do not need to compute Ax in every iteration. Instead, we can update the product and save computing time (the longer x is, the more time we can save).
Let x^c denote the current solution and x^n the neighbour solution, produced by adding element-wise the vector x^Δ to x^c. If only few elements change, then x^Δ will be relatively sparse, i.e. many of its elements are zero.

$$x^n = x^c + x^\Delta .$$

Then we have:

$$Ax^{\mathrm{n}} = A(x^{\mathrm{c}} + x^{\Delta}) = \underbrace{Ax^{\mathrm{c}}}_{\text{known}} + Ax^{\Delta}.$$

So we only need to compute Ax^{Δ} in every iteration, in which many columns of A are ignored because the corresponding elements of x^{Δ} are zero.
Let us solve the risk-minimisation model one more time. First, we add the initial product Ax as an attribute to the solution.

```
> attr(x0, "Ax") <- R %*% x0
```

The objective function now has less work to do: it does not compute Ax, but only its variance.

```
> variance3 <- function(x, ...)
      var(attr(x, "Ax"))
```

The final ingredient is the neighbourhood function.

```
> nb_upd <- neighbourfun(type = "numeric",
                         min = 0, max = 0.2,
                         stepsize = 0.005,
                         update = "Ax", A = R)
```

It remains to call `LSopt`.

```
> sol.ls3 <- LSopt(variance3,
                   list(x0 = x0,
                        neighbour = nb_upd,
                        nI = 2000,
                        printBar = FALSE))
```

```
Local Search.
Initial solution:  0.000651
Finished.
Best solution overall: 0.000561
```

```
> data.frame(LS = round(sol.ls$xbest,  2)[1:10],
             LS2 = round(sol.ls2$xbest, 2)[1:10],
             LS3 = round(sol.ls3$xbest, 2)[1:10],
             QP = sol.qp[1:10])
```

```
     LS  LS2  LS3   QP
1  0.00 0.00 0.00 0.00
2  0.00 0.00 0.00 0.00
3  0.00 0.00 0.00 0.00
4  0.12 0.12 0.12 0.12
5  0.20 0.20 0.20 0.20
6  0.02 0.02 0.02 0.02
7  0.00 0.00 0.00 0.00
8  0.11 0.11 0.11 0.11
9  0.04 0.04 0.04 0.04
10 0.00 0.00 0.00 0.00
```

The solution remains the same, but in particular for large matrices A, the optimisation can become much faster.

References

Robert Gentleman and Ross Ihaka. Lexical scope and statistical computing. *Journal of Computational and Graphical Statistics*, 9(3):491–508, 2000.

Manfred Gilli, Dietmar Maringer, and Enrico Schumann. *Numerical Methods and Optimization in Finance*. Elsevier/Academic Press, 2nd edition, 2019. URL `http://enricoschumann.net/NMOF`.