Linear Programming in R

Therese Donovan

2020-04-01

Overview

This script is a translation of the linear programming model introduced by Tony in class.

In this problem, you are hired as a consultant for the jurisdiction of "Erehwon" to recommend the purchase of valley hectares or highlight hectares to conserve as many plant species as possible . The valley has 350 unique plant species, while the highlight has 240 unique plant species.

How many hectares of each habitat should be conserved? The decision is not so simple because you have a number of decision *constraints* to contend with. These constraints contain bits of information that need to be included in the decision problem (such as the cost per hectare or the total new plant species that are added with each new hectare conserved).

- Budget: The maximum budget is 24,000 pickles.
- Total unique plant species by habitat:
  - valley = 350
  - highland = 240
- Number of new species conserved for each additional hectare purchased:
  - valley = 5
  - highland = 3
- Edge effects
  - valley = 15
  - highland = 25
- Political Considerations
  - valley = 2
  - highland = 2
- Social upheaval per hectare
  - valley = 5
  - highland = 2
- Cost per hectare
  - valley = 300
  - highland = 200

The decision is to identify how many highland hectares (h) and how many valley hectares (v) shoudl be purchased to maximize the total species conserved, given two additional constraints:

- You total budget is 24000 pickles
- The total social upheaval is

Tony illustrated how to find the solution space for this problem by plotting lines on a graph (hence the name, linear programming). We then showed how Solver could be used to find the solution as well.

Linear Programming in R

As you may have guessed, there are several ways to approach this problem in R. Here, we will follow an example from a post by **Roberto Salazar** in the blog "Towards Data Science":

https://towardsdatascience.com/linear-programming-in-r-444e9c199280

Take a moment to read that post now. Roberto provides a summary of optimization in general as follows:

An optimization model seeks to find the values of the decision variables that optimize (maximize or minimize) an objective function among the set of all values for the decision variables that satisfy the given constraints. Its three main components are:

1. *Objective function: a function to be optimized (maximized or minimized)*
2. *Decision variables: controllable variables that influence the performance of the system*
3. *Constraints: set of restrictions (i.e. linear inequalities or equalities) of decision variables. A non-negativity constraint limits the decision variables to take positive values (e.g. you cannot produce negative number of items x1, x2 and x3).*
4. *The solution of the optimization model is called the optimal feasible solution.*

Roberto then walks us through modeling steps for a generic optimization problem:

Modeling accurately an operations research problem represents the most significant-and sometimes the most difficult-task. A wrong model will lead to a wrong solution, and thus, will not solve the original problem. The following steps should be performed by different team members with different areas of expertise to obtain an accurate and greater view of the model:

- *Problem definition: defining the scope of the project and identifying that the result is the identification of three elements: description of decision variables, determination of the objective and determination of the limitations (i.e. constraints).*
- *Model construction: translating the problem definition into mathematical relationships.*
- *Model solution: using a standard optimization algorithm. Upon obtaining a solution, a sensitivity analysis should be performed to find out the behavior of the solution due to changes in some of the parameters.*
- *Model validity: checking if the model works as it was supposed to.*
- *Implementation: translating the model and the results into the recommendation of a solution.*

I think Roberto would like Tony, don't you?

Finally, Roberto describes linear programming as just one type of optimization problem:

Linear programming (also referred as LP) is an operations research technique used when all the objectives and constraints are linear (in the variables) and when all the decision variables are continuous. In hierarchy, linear programming could be considered as the easiest operations research technique. The lpSolve package from R contains several functions for solving linear programming problems and getting significant statistical analysis.

I would like to fully credit Roberto for providing such a useful posting. He goes on to give an example of the package, lpSolve, and we'll look at this package next.

Package lpSolve

If you haven't installed **lpSolve** on your machine yet, do so now. You can do through the RStudio Packages pain, or type the following in your console:

```
install.packages("lpSolve")
```

To use the functions in the lpSolve package, we must load it with the `library()` function.

```
library(lpSolve)
```

Whenever you work with a new package, the very first thing you should do is look at the package's helpfile. Here, we can use the `help()` function and specify 'lpSolve' as the package to bring up this file.

```
help(package = 'lpSolve')
```

You should see a link for the DESCRIPTION file (a short description of the package), and a list of functions that are included within the package. The description reads:

"Lp_solve is freely available (under LGPL 2) software for solving linear, integer and mixed integer programs. In this implementation we supply a 'wrapper' function in C and some R functions that solve general linear/integer problems, assignment problems, and transportation problems. This version calls lp_solve version 5.5."

So, the R package **lpSolve** is a "wrapper function" – it is a function that let's R users enter inputs, and the function itself will call "lp_solve version 5.5" to do the actual work. After lp_solve version 5.5 does its work, the authors of the package (Michel Berkelaar and others) collect the output, and return it as an R object that can be further processed.

The actual lp_solve source code can be found at https://sourceforge.net/projects/lpsolve/files/lpsolve/5.5.2.0/. The summary tab at this website tells us what it does: "Mixed Integer Linear Programming (MILP) solver lp_solve solves pure linear, (mixed) integer/binary, semi-cont and special ordered sets (SOS) models. lp_solve is written in ANSI C and can be compiled on many different platforms like Linux and WINDOWS."

This package has just a few functions, and the one that we will use is called `lp()`. Let's dig into that function's helpfile:

```
help('lp')
```

The function's description indicates that the `lp()` function is an interface to lp_solve linear/integer programming system. This is the package's main function. Let's look at the function's arguments:

```
args(lp)
```

```
## function (direction = "min", objective.in, const.mat, const.dir,
##     const.rhs, transpose.constraints = TRUE, int.vec, presolve = 0,
##     compute.sens = 0, binary.vec, all.int = FALSE, all.bin = FALSE,
##     scale = 196, dense.const, num.bin.solns = 1, use.rw = FALSE)
## NULL
```

This function has many arguments, some of which have default values. You need to develop some good habits for working with R, and one important habit is to always read a function's helpfile before using a function for the first time. Let's review the arguments that we need to solve our plant conservation problem:

- **direction**: Character string giving direction of optimization: "min" (default) or "max."
- **objective.in**: Numeric vector of coefficients of objective function
- **const.mat**: Matrix of numeric constraint coefficients, one row per constraint, one column per variable (unless transpose.constraints = FALSE; see below).
- **const.dir**: Vector of character strings giving the direction of the constraint: each value should be one of "<," "<=," "=," "==," ">," or ">=". (In each pair the two values are identical.) = **const.rhs**: Vector of numeric values for the right-hand sides of the constraints.

The very best way to learn about how to use a function in R is to head to the Examples section in the helpfile (after reading the description and have a look at the argument definitions, of course.) Examples are meant to be copied and pasted into your R console. This way, you can see what each line of code produces by looking in your Global Environment. The example looks like this:

```
Set up problem: maximize
   x1 + 9 x2 +   x3 subject to
   x1 + 2 x2 + 3 x3  <= 9
 3 x1 + 2 x2 + 2 x3 <= 15

f.obj <- c(1, 9, 1)
f.con <- matrix (c(1, 2, 3, 3, 2, 2), nrow = 2, byrow = TRUE)
f.dir <- c("<=", "<=")
f.rhs <- c(9, 15)

# Now run
lp ("max", f.obj, f.con, f.dir, f.rhs)
```

In a nutshell, the first part of the problem can be written in pencil. Once this is done, the next four pieces should fall into place, which involves creating R objects that can be passed to the `lp()` function.

- **f.obj** specify the coefficients of the objective function
- **f.con** is a matrix of the coefficients of the constraint equations
- **f.dir** provides the directionality of the constraint equations
- **f.rhs** is the right hand side of the constraint equations.

Have another look at the `lp()` helpfile, and pay attention to the default arguments. These are the arguments that will be used by the function unless you, the coder, tell it otherwise.

Now we can apply this example to our Saving the Species of Erehwon problem. We have two variables, v and h.

```
Set up problem: maximize
   5v +   3h subject to
   1v +   0h   >= 15    (edge effects for valley)
   0v +   1h   >= 25    (edge effects for highland)
   1v -   2h   <= 0     (politics constraining valley purchase)
  -2v +   1h   <= 0     (politics constraining highland purchase)
   5v +   2h   <= 400   (social upheaval constraint)
 300v + 200h   <= 24000 (budge constraint)
   1v +   0h   <= 70    (species richness valley constraint)
   0v +   1h   <= 60    (species richness highland constraint)
```

```r
# first, let's define the coefficients for the function to be optimized
f.obj <- c(5, 3)

# Set matrix corresponding to coefficients of constraints by rows
# Do not consider the non-negative constraint; it is automatically assumed
f.con <- matrix(c(1, 0, 0, 1, 1, -2, -1, 1, 5, 2, 300, 200, 1, 0, 0, 1),
         nrow = 8,
         byrow = TRUE)

# Set unequality signs
f.dir <- c(">=", ">=", "<=", "<=", "<=", "<=", "<=", "<=")

# Set right hand side coefficients
f.rhs <- c(15, 25, 0, 0, 400, 24000, 70, 60)

# Now run
lp(direction = "max",
   objective.in = f.obj,
   const.mat = f.con,
```

```
   const.dir = f.dir,
   const.rhs = f.rhs)
```

```
## Success: the objective function is 390
```

The returned answer is 390 – same as Tony got with his polygon drawing. We can save 390 species. Certainly there is more information than just this. Let's run this again, but save the result as an R object so we can dig a bit deeper:

```
# Now run
results <- lp(direction = "max",
   objective.in = f.obj,
   const.mat = f.con,
   const.dir = f.dir,
   const.rhs = f.rhs,
   all.int = T
  )
```

As with so many R analyses, the object that is output is rich with information. First, let's see how this object is structured:

```
str(results)
```

```
## List of 28
##  $ direction       : int 1
##  $ x.count         : int 2
##  $ objective       : num [1:2] 5 3
##  $ const.count     : int 8
##  $ constraints     : num [1:4, 1:8] 1 0 2 15 0 1 2 25 1 -2 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:4] "" "" "const.dir.num" "const.rhs"
##   .. ..$ : NULL
##  $ int.count       : int 2
##  $ int.vec         : int [1:2] 1 2
##  $ bin.count       : int 0
##  $ binary.vec      : int 0
##  $ num.bin.solns   : int 1
##  $ objval          : num 390
##  $ solution        : num [1:2] 60 30
##  $ presolve        : int 0
##  $ compute.sens    : int 0
##  $ sens.coef.from  : num 0
##  $ sens.coef.to    : num 0
##  $ duals           : num 0
##  $ duals.from      : num 0
##  $ duals.to        : num 0
##  $ scale           : int 196
##  $ use.dense       : int 0
##  $ dense.col       : int 0
##  $ dense.val       : num 0
##  $ dense.const.nrow: int 0
##  $ dense.ctr       : num 0
##  $ use.rw          : int 0
##  $ tmp             : chr "Nobody will ever look at this"
##  $ status          : int 0
##  - attr(*, "class")= chr "lp"
```

As you can see, the output is structured as a list, but is also has a class of "lp". This means we can use list indexing to look at certain parts of the output, but the author of the package may have written other functions (methods) that will work on objects of class "lp". For example, we can look at the solution as follows:

```
# Variables final values
results$solution
```

```
## [1] 60 30
```

I have not been able to figure out how to return the slack values from this analysis (if you figure it out, please let me know!). It appears that the beefier package, lpSolveAPI, will allow this.

So..... we'll try this same analysis in a different R package, **linprog**, which will provide the full simplex output. Luckily, we will be able to re-use some R objects we already made, namely f.obj (our objective function variables), f.con (our constraints matrix), f.dir (our vector of inequalities), and f.rhs (the right-hand values).

Package linprog

The package **linprog** can also be used solve a linear programming problem. This package may be of interest to those who want to dig deeper and see the actual results of the simplex algorithm.

https://www.rdocumentation.org/packages/linprog/versions/0.9-2/topics/solveLP

Let's install this package:

```
install.packages("linprog")
```

And next let's call it into R so we can use its functions:

```
library(linprog)
```

As always, start your exploration of a new package by looking at the package index helpfiles.

```
help(package = 'linprog')
```

Here, you shoudl see that the package author is Arne Henningsen. The description file says "This package can be used to solve Linear Programming / Linear Optimization problems by using the simplex algorithm." This is the same algorithm used by Excel (Excel's optimizer is actually written by Frontline Solver. If you are serious about linear or integer programming, I highly, highly recommend that you look at their fantastic tutorials. See https://www.solver.com/.)

The main solving function in this package is called `solveLP()`. Let's have a look at its helpfile:

```
help(solveLP)
```

"This function minimizes (or maximizes) c'x, subject to A x <= b and x >= 0.

Note that the inequality signs <= of the individual linear constraints in A x <= b can be changed with argument const.dir."

Let's look at the function's arguments:

```
args(solveLP)
```

```
## function (cvec, bvec, Amat, maximum = FALSE, const.dir = rep("<=",
##     length(bvec)), maxiter = 1000, zero = 1e-09, tol = 1e-06,
##     dualtol = tol, lpSolve = FALSE, solve.dual = FALSE, verbose = 0)
## NULL
```

As with the first package (lpSolve) and its LP solving function (`lp()`), we need to provide the function `solveLP()` the pieces of the linear progamming problem. Most fortunately, we can use the objects we created for the `lp()` function, and recycle them as inputs to the `solveLP()` function.

```
solveLP(cvec = f.obj,
         bvec = f.rhs,
         Amat = f.con,
         maximum = TRUE,
         const.dir = f.dir)
```

```
##
##
## Results of Linear Programming / Linear Optimization
##
## Objective function (Maximum): 390
##
## Iterations in phase 1: 3
## Iterations in phase 2: 1
## Solution
##   opt
## 1  60
## 2  30
##
## Basic Variables
##     opt
## 1    60
## 2    30
## S 1  45
## S 2   5
## S 4  30
## S 5  40
## S 7  10
## S 8  30
##
## Constraints
##   actual dir  bvec free    dual dual.reg
## 1     60  >=    15   45 0.00000       45
## 2     30  >=    25    5 0.00000        5
## 3      0  <=     0    0 0.12500       48
## 4    -30  <=     0   30 0.00000       30
## 5    360  <=   400   40 0.00000       40
## 6  24000  <= 24000    0 0.01625     4000
## 7     60  <=    70   10 0.00000       10
## 8     30  <=    60   30 0.00000       30
##
## All Variables (including slack variables)
##     opt cvec      min.c     max.c     marg marg.reg
## 1    60    5   4.500000       Inf       NA       NA
## 2    30    3 -10.000000  3.333333       NA       NA
## S 1  45    0  -0.500000       Inf  0.00000       NA
## S 2   5    0 -13.000000  0.333333  0.00000       NA
## S 3   0    0       -Inf  0.125000 -0.12500       48
## S 4  30    0  -0.200000       Inf  0.00000       NA
## S 5  40    0         NA  0.250000  0.00000       NA
## S 6   0    0       -Inf  0.016250 -0.01625     4000
```

```
## S 7  10    0         NA  0.500000  0.00000       NA
## S 8  30    0  -0.333333 13.000000  0.00000       NA
```

Now we're talking. Under the section, Basic Variables, you can see the solution, and 6 of the slack constraints. These should match up with your spreadsheet (please double-check this).

The returned results must be saved as an R object if you want to look more deeply at what the function has to offer. Here, we'll save our new output as an object called "new.results", and then look at the structure of this object so that we can understand exactly what it is.

```
new.results <- solveLP(cvec = f.obj,
         bvec = f.rhs,
         Amat = f.con,
         maximum = TRUE,
         const.dir = f.dir)

str(new.results)
```

```
## List of 12
##  $ status    : num 0
##  $ opt       : num 390
##  $ iter1     : num 3
##  $ iter2     : num 1
##  $ allvar    :'data.frame':  10 obs. of  6 variables:
##   ..$ opt     : num [1:10] 60 30 45 5 0 30 40 0 10 30
##   ..$ cvec    : num [1:10] 5 3 0 0 0 0 0 0 0 0
##   ..$ min.c   : num [1:10] 4.5 -10 -0.5 -13 -Inf ...
##   ..$ max.c   : num [1:10] Inf 3.333 Inf 0.333 0.125 ...
##   ..$ marg    : num [1:10] NA NA 0 0 -0.125 ...
##   ..$ marg.reg: num [1:10] NA NA NA NA 48 NA NA 4000 NA NA
##  $ basvar    : num [1:8, 1] 60 30 45 5 30 40 10 30
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:8] "1" "2" "S 1" "S 2" ...
##   .. ..$ : chr "opt"
##  $ solution  : Named num [1:2] 60 30
##   ..- attr(*, "names")= chr [1:2] "1" "2"
##  $ con       :'data.frame':  8 obs. of  6 variables:
##   ..$ actual  : num [1:8] 60 30 0 -30 360 24000 60 30
##   ..$ dir     : chr [1:8] ">=" ">=" "<=" "<=" ...
##   ..$ bvec    : num [1:8] 15 25 0 0 400 24000 70 60
##   ..$ free    : num [1:8] 45 5 0 30 40 0 10 30
##   ..$ dual    : num [1:8] 0 0 0.125 0 0 ...
##   ..$ dual.reg: num [1:8] 45 5 48 30 40 4000 10 30
##  $ maximum   : logi TRUE
##  $ lpSolve   : logi FALSE
##  $ solve.dual: logi FALSE
##  $ maxiter   : num 1000
##  - attr(*, "class")= chr "solveLP"
```

The returned object is another list, but it is also of class "solveLP". We can dig into this object with list indexing, as the following example shows:

```
new.results$allvar
```

```
##     opt cvec       min.c      max.c     marg marg.reg
## 1    60    5   4.5000000        Inf       NA       NA
## 2    30    3 -10.0000000  3.3333333       NA       NA
```

```
## S 1  45    0  -0.5000000        Inf  0.00000       NA
## S 2   5    0 -13.0000000  0.3333333  0.00000       NA
## S 3   0    0        -Inf  0.1250000 -0.12500       48
## S 4  30    0  -0.2000000        Inf  0.00000       NA
## S 5  40    0          NA  0.2500000  0.00000       NA
## S 6   0    0        -Inf  0.0162500 -0.01625     4000
## S 7  10    0          NA  0.5000000  0.00000       NA
## S 8  30    0  -0.3333333 13.0000000  0.00000       NA
```

The "Details" of the `solveLP()` function is enlightening. The author basically says that the package lpSolve (our first package) is much faster, but that this package produces more detailed results (as we hoped!).

This function uses the Simplex algorithm of George B. Dantzig (1947) and provides detailed results (e.g. dual prices, sensitivity analysis and stability analysis).

Solving the Linear Programming problem by the package lpSolve (of course) requires the installation of this package, which is available on CRAN (http://cran.r-project.org/src/contrib/PACKAGES.html#lpSolve). Since the lpSolve package uses C-code and this (linprog) package is not optimized for speed, the former is much faster. However, this package provides more detailed results (e.g. dual values, stability and sensitivity analysis).

Conclusion

Roberto offers these final thoughts at the end of his blog, and gives a really nice nod to his professional mentor:

Linear programming represents a great optimization technique for better decision making. The lpSolve R package allows to solve linear programming problems and get significant statistical information (i.e. sensitivity analysis) with just a few lines of code. While there are other free optimization software out there (e.g. GAMS, AMPL, TORA, LINDO), having stored a linear optimization R code in your personal code library could save you a significant amount of time by not having to write the formulation from scratch, but instead by only having to change the coefficients and signs of the corresponding matrices. Special thanks to Dr. Édgar Granda, Ph.D, for all his teachings during the Linear Programming course and during my academic career as an industrial and systems engineer undergraduate student.

But now I'd like to give a nod to my professional mentor, Tony Starfield. In our class video, Tony states that he doesn't use LP to find *the* optimum solution. That makes Tony cringe because the optimum solution depends on the conditions that you apply. It is a model world concept, and the real world is a lot more complicated. Tony uses the LP technique to explore and to interpret back from the model world to the real world, in the hopes that a really good solution can be found.