Integer Programming in R

Therese Donovan

2020-04-08

Overview

This script is a translation of the integer programming model, the Endangered Species of Int, introduced by Tony in class. We will be using the two packages we used last week (lpSolve and linprog) to solve the Plants of Erehwon problem, where we had to determine the number of highland hectares and the number of valley hectares to purchase to maximize species conservation. Let's start this week's exercise by reviewing that problem quickly, as we'll need to configure this week's problem in a similar way.

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

```
library(lpSolve)
```

As a refresher, the package 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" – it contains functions that permits R users enter inputs, and the functions call "lp_solve version 5.5" to do the actual work; the returned results are delivered back to R.

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. Never, ever forget that R will use the default values unless you, the user, specify otherwise.

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.obj, f.con = f.con, f.dir = f.dir, f.rhs = f.rhs)
```

In a nutshell, the first part of the problem (above the dashed line) can be written in pencil. Once you have that written out, you need to create R objects that can be fed into the function `lp()`. These objects are called **f.obj** (which gives the coefficients of the objective function), **f.con** (which is a matrix of the coefficients of the constraints, left side), **f.dir** (which gives the directionality of the constraints), and **f.rhs** (which gives the constraint right hand side values). Take a moment to match these objects up with the "pencil" sketch.

Integer Programming in R

We can use `lp()` to solve the Endangered Species of Int problem too. In this problem, you are hired as a consultant to help identify *which* sites (12 possible) should be conserved to ensure the protection of 14 species.

The data to be used for this problem are in a csv file called "SpeciesOfInt.csv". This file provides a listing of each site, the cost of each site, and the occurrence (or absence) of each species of interest across sites.

Let's load this file now into R (assuming this file is located in your working directory).

```
data <- read.csv(file = "SpeciesOfInt.csv", header = T)

data
```

```
##    Site S1 S2 S3 S4 S5 S6 S7 S8 S9 S10 S11 S12 S13 S14 Cost
## 1     1  0  1  1  0  0  0  0  1  0   0   1   0   0   0   10
## 2     2  0  1  0  0  0  0  1  1  0   0   0   1   0   0    6
## 3     3  1  1  1  0  0  1  1  0  1   0   1   0   0   1   15
## 4     4  0  0  1  1  0  1  0  0  1   1   0   0   0   1    5
## 5     5  1  1  0  0  0  1  1  0  1   1   0   1   0   0   17
## 6     6  1  0  1  1  1  0  0  0  0   1   1   1   0   0    7
## 7     7  0  1  1  0  0  0  0  1  1   0   0   0   1   0    5
## 8     8  1  0  1  0  1  0  0  1  0   1   0   0   1   0   11
## 9     9  0  0  1  1  0  0  0  0  1   0   0   0   0   1    8
## 10   10  0  1  1  0  0  0  1  0  1   1   1   0   0   0   18
## 11   11  1  1  0  0  0  0  0  1  0   0   0   1   0   1   12
## 12   12  1  0  1  1  1  0  0  0  0   0   0   0   1   0    9
```

Here, we have 12 sites of interest. These are the rows of the dataframe. Sites are identified as 1-12 in column 1. The next set of columns identifies species of interest, labeled S1-S14. The grid consists of 0's and 1's, where a 1 indicates that a given species is present on the site, while a 0 indicates the species is absent. The final column is named "Cost", which gives the cost of each site in terms of conservation purchase.

Let's split out the pieces of information that we will need soon. First, let's create a cost vector:

```
cost <- data$Cost
cost
```

```
##  [1] 10  6 15  5 17  7  5 11  8 18 12  9
```

Next, let's create a matrix that holds our species presence-absence information. We will need to return all rows, but only columns 2:15:

```
pa <- data[,2:15]
pa
```

```
##    S1 S2 S3 S4 S5 S6 S7 S8 S9 S10 S11 S12 S13 S14
## 1   0  1  1  0  0  0  0  1  0   0   1   0   0   0
## 2   0  1  0  0  0  0  1  1  0   0   0   1   0   0
## 3   1  1  1  0  0  1  1  0  1   0   1   0   0   1
## 4   0  0  1  1  0  1  0  0  1   1   0   0   0   1
## 5   1  1  0  0  0  1  1  0  1   1   0   1   0   0
## 6   1  0  1  1  1  0  0  0  0   1   1   1   0   0
## 7   0  1  1  0  0  0  0  1  1   0   0   0   1   0
## 8   1  0  1  0  1  0  0  1  0   1   0   0   1   0
## 9   0  0  1  1  0  0  0  0  1   0   0   0   0   1
## 10  0  1  1  0  0  0  1  0  1   1   1   0   0   0
## 11  1  1  0  0  0  0  0  1  0   0   0   1   0   1
## 12  1  0  1  1  1  0  0  0  0   0   0   0   1   0
```

We know that our variables (the site choice matrix) can only take on values of 0 or 1. That is, they are binary.

The goal of the problem is simple: you have a budget, and must protect all species in at least one site with the smallest budget possible. Which "mix" of sites will accomplish this? This is a "portfolio" problem, much like the challenge of identifying a portfolio of retirement options for investment.

Now, let's sketch out this problem with "pencil" as we did earlier. We wish to minimize our cost function subject to constraints.

```
Set up problem: minimize

   C =  X1*10 + X2*6 + X3*15 + ... + X12*9

 subject to

 X1*0 + X2*0 + . . . + X12*1 >= 1 (species 1 constraint)
 X1*1 + X2*1 + . . . + X12*0 >= 1 (species 2 constraint)
 X1*1 + X2*0 + . . . + X12*1 >= 1 (species 3 constraint)
 X1*0 + X2*0 + . . . + X12*1 >= 1 (species 4 constraint)
 X1*0 + X2*0 + . . . + X12*1 >= 1 (species 5 constraint)
 X1*0 + X2*0 + . . . + X12*0 >= 1 (species 6 constraint)
 X1*0 + X2*0 + . . . + X12*0 >= 1 (species 7 constraint)
 X1*1 + X2*1 + . . . + X12*0 >= 1 (species 8 constraint)
 X1*0 + X2*0 + . . . + X12*0 >= 1 (species 9 constraint)
 X1*0 + X2*0 + . . . + X12*0 >= 1 (species 10 constraint)
 X1*1 + X2*0 + . . . + X12*0 >= 1 (species 11 constraint)
 X1*0 + X2*1 + . . . + X12*0 >= 1 (species 12 constraint)
 X1*0 + X2*0 + . . . + X12*1 >= 1 (species 13 constraint)
 X1*0 + X2*0 + . . . + X12*0 >= 1 (species 14 constraint)

 X1, X2, X3, ... X12 binary
```

We have unknown variables, X1 through X12, that make up our "choice" vector. These must be binary. The total cost of any porfolio is the sumproduct of the choice vector times the cost of each site. I've labeled this "C", and we wish to find the portfolio that minimizes C under certain constraints. Our constraints

Now we just need to figure out how to enter these in R.

```
# first, let's define the coefficients for the function to be optimized
f.obj <- cost
```

```
# look at the cost vector
cost
```

```
##  [1] 10  6 15  5 17  7  5 11  8 18 12  9
```

Notice the cost vector is of length 12. Next, we need a matrix of coefficients for our constraints. In our pencil-sketch, we've identified 14 constraints, so we need 14 equations. The presence-absence data (called pa) consists of 12 rows and 14 columns. We need to transpose this dataset so that it consists of 14 rows and 12 columns (14 rows to match our constraints).

Set matrix corresponding to coefficients of constraints by rows

```
# this means we'll need to transpose our presence-absence dataframe with the t function.
f.con <- t(pa)

# look at the transposed p-a data
f.con
```

```
##     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
## S1     0    0    1    0    1    1    0    1    0     0     1     1
## S2     1    1    1    0    1    0    1    0    0     1     1     0
## S3     1    0    1    1    0    1    1    1    1     1     0     1
## S4     0    0    0    1    0    1    0    0    1     0     0     1
## S5     0    0    0    0    0    1    0    1    0     0     0     1
## S6     0    0    1    1    1    0    0    0    0     0     0     0
## S7     0    1    1    0    1    0    0    0    0     1     0     0
## S8     1    1    0    0    0    0    1    1    0     0     1     0
## S9     0    0    1    1    1    0    1    0    1     1     0     0
## S10    0    0    0    1    1    1    0    1    0     1     0     0
## S11    1    0    1    0    0    1    0    0    0     1     0     0
## S12    0    1    0    0    1    1    0    0    0     0     1     0
## S13    0    0    0    0    0    0    1    1    0     0     0     1
## S14    0    0    1    1    0    0    0    0    1     0     1     0
```

Next, we need a vector of length 14 that gives the inquality direction. We need for each species to occur in at least one site, so our direction is ">=".

```
# Set inequality signs
f.dir <- rep(">=", times = 14)

# Look at the direction vector
f.dir
```

```
##  [1] ">=" ">=" ">=" ">=" ">=" ">=" ">=" ">=" ">=" ">=" ">=" ">=" ">=" ">="
```

Finally, we need to set the right hand side coefficients for our 14 constraints. We need for each species to occur in at least 1 site:

```
# Set right hand side coefficients
f.rhs <- rep(1, times = 14)

# Look at the right hand side coefficients
f.rhs
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Now we can send these objects to the `lp()` function. We need to make sure that "all.bin" argument is set to TRUE to enforce that our variables only take on values of 0 or 1.

```
# Now run, make sure the all.bin argument is TRUE
lp(direction = "min",
   objective.in = f.obj,
   const.mat = f.con,
   const.dir = f.dir,
   const.rhs = f.rhs,
   all.bin = TRUE
   )
```

```
## Success: the objective function is 23
```

The returned answer is 23 – same as we got on our spreadsheet. We can purchase a set of sites for a cost of 23 units and ensure that each of the 14 species is conserved at at least 1 site. Surely there is more information than just this! We need to know *which* sites were selected in the final portfolio. Let's run this again, but save the result as an R object so we can dig a bit deeper:

```
# Now run again, saving the results as an R object called results
results <- lp(direction = "min",
   objective.in = f.obj,
   const.mat = f.con,
   const.dir = f.dir,
   const.rhs = f.rhs,
   all.bin = TRUE
   )
```

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 0
##  $ x.count         : int 12
##  $ objective       : int [1:12] 10 6 15 5 17 7 5 11 8 18 ...
##  $ const.count     : int 14
##  $ constraints     : num [1:14, 1:14] 0 0 1 0 1 1 0 1 0 0 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:14] "" "" "" "" ...
##   .. ..$ : chr [1:14] "S1" "S2" "S3" "S4" ...
##  $ int.count       : int 0
##  $ int.vec         : int 0
##  $ bin.count       : int 12
##  $ binary.vec      : int [1:12] 1 2 3 4 5 6 7 8 9 10 ...
##  $ num.bin.solns   : int 1
##  $ objval          : num 23
##  $ solution        : num [1:12] 0 1 0 1 0 1 1 0 0 0 ...
##  $ 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. For example, we can look at the solution as follows:

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

```
##  [1] 0 1 0 1 0 1 1 0 0 0 0 0
```

This means that the final portfolio includes sites 2, 4, 6, and 7. This is the same solution that Solver found using the Simplex method. 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

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

```
library(linprog)
```

You may recall 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)
```

Let's show the function's arguments (and make sure to read the description in the helpfile):

```
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
```

The argument names for `solveLP()` are very different than the `lp()` function, but they are basically the same thing. We need an objective function, we need to specify if we want to minimize or maximize this function, we need constraint coefficients, directions, and right-hand values. We're looking for a way to specify that we want to minimize the objective function (which we can do by setting the **maximum** argument to

FALSE). But we're also looking for a way to force the variables to be binary. I don't see this though! Let's try this anyway and see what happens:

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

```
##
##
## Results of Linear Programming / Linear Optimization
##
## Objective function (Minimum): 23
##
## Iterations in phase 1: 17
## Iterations in phase 2: 7
## Solution
##    opt
## 1    0
## 2    1
## 3    0
## 4    1
## 5    0
## 6    1
## 7    1
## 8    0
## 9    0
## 10   0
## 11   0
## 12   0
##
## Basic Variables
##      opt
## 2      1
## 3      0
## 4      1
## 6      1
## 7      1
## S 1    0
## S 2    1
## S 3    2
## S 4    1
## S 6    0
## S 8    1
## S 9    1
## S 10   1
## S 12   1
##
## Constraints
##    actual dir bvec free dual    dual.reg
## 1       1  >=    1    0    0 1.66533e-15
## 2       2  >=    1    1    0 1.00000e+00
## 3       3  >=    1    2    0 2.00000e+00
## 4       2  >=    1    1    0 1.00000e+00
```

```
## 5       1  >=    1    0    3 8.88178e-16
## 6       1  >=    1    0    0          NA
## 7       1  >=    1    0    6 4.09418e+14
## 8       2  >=    1    1    0 1.00000e+00
## 9       2  >=    1    1    0 1.00000e+00
## 10      2  >=    1    1    0 1.00000e+00
## 11      1  >=    1    0    4 1.00000e+00
## 12      2  >=    1    1    0 1.00000e+00
## 13      1  >=    1    0    5         Inf
## 14      1  >=    1    0    5         Inf
##
## All Variables (including slack variables)
##      opt cvec min.c max.c marg    marg.reg
## 1      0   10  99.0  77.0    6 0.00000e+00
## 2      1    6  -9.0   7.0   NA          NA
## 3      0   15 -16.0  18.0   NA          NA
## 4      1    5  -8.0   6.0   NA          NA
## 5      0   17  99.0  77.0   11 1.00000e+00
## 6      1    7 -10.0   8.0   NA          NA
## 7      1    5 -10.0   6.0   NA          NA
## 8      0   11  99.0  77.0    3 5.00000e-01
## 9      0    8  99.0  77.0    3 1.00000e+00
## 10     0   18  99.0  77.0    8 0.00000e+00
## 11     0   12  99.0  77.0    7 1.00000e+00
## 12     0    9  99.0  77.0    1 5.00000e-01
## S 1    0    0  -1.0   6.0    0          NA
## S 2    1    0  -5.0   1.0    0          NA
## S 3    2    0  -3.0   1.0    0          NA
## S 4    1    0  -1.5   1.0    0          NA
## S 5    0    0  -3.0   Inf    3 0.00000e+00
## S 6    0    0  -5.0   3.0    0          NA
## S 7    0    0  -6.0   Inf    6 4.09418e+14
## S 8    1    0  -3.0   0.5    0          NA
## S 9    1    0  -5.0   1.0    0          NA
## S 10   1    0  -1.5   0.5    0          NA
## S 11   0    0  -4.0   Inf    4 1.00000e+00
## S 12   1    0  -1.5   0.5    0          NA
## S 13   0    0  -5.0   Inf    5         Inf
## S 14   0    0  -5.0   Inf    5         Inf
```

It appears that the binary constraint was (somehow) adhered to. I'm not sure how! Under the section, Solution, you can see the solution. Under the Section, "All Variables", you can see each of the final solution in rows 1-12, followed by the slack variables for each of the 14 species 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.

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

Last week, we mentioned a blog written by **Roberto Salazar** in the blog "Towards Data Science":

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

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. Let's not lose sight of this!