test

Diego Aviles

Wednesday, July 29, 2015

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

Heather?

Heather?!?!?!?

```
install_github("avilesd/productConfig")
library(productConfig)
```

```
head(camera_data)
```

```
##   cid usid round atid selected selectable
## 1  13    6     0    1        3          3
## 2  14    6     1    1        3          3
## 3  15    6     2    1        3         -1
## 4  16    6     3    1        3         -1
## 5  17    6     4    1        0          3
## 6  18    6     5    1        0          3
```

Before calculating the decision matrix, it is necessary to first gather some key data from the `camera_data` provided. This is were the `GetFunctions` come in. For example, it is necessary to know how many attributes there are,

```
get_attrs_ID(camera_data)
```

```
## [1] 1 2 3 4
```

as well as how the possible value each attribute can have:

```
get_attr_values(dataset = camera_data, attrid = 1)
get_attr_values(camera_data, 2)
get_attr_values(camera_data, 3)
get_attr_values(camera_data, 4) ## Price attribute
```

```
## [1] 3 0 2 1
```

```
## [1] 0 3 2 1
```

```
## [1] 0 3 2 1
```

```
## [1]  0.16805556 -0.27777778 -0.12916667  0.01944444  0.46527778  0.31666667
## [7]  0.61388889
```

But even more important, is to know how many `rounds`, i.e. how many rows our decision matrix can have. For the remainder of this example we are going to work with a random picked `userid = 18`.

```
get_rounds_by_ID(camera_data, userid = 18)
```

```
##  [1]  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14
```

The user 18, interacted 15 times with the camera configurator. This means, that there were 15 configurations the user considerated before taking his decision. Note that given the nature of product configurators, it is likely to see duplicate alternatives, i.e. equal configurations.

We know now the number of columns (attributes) and the number of rows (rounds) the decision matrix can have, for our selected user. For the decision matrix the reader has the ability to choose how many columns and rounds he wants to use, this is done through the `attr` and `rounds` parameters, respectively. To calculate the decision matrix using all attributes and rounds, we can find at least three equivalent ways to do it:

```
decision_matrix(data= camera_data, userid= 18,  attr= c(1,2,3,4),
                rounds= c(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14))

decision_matrix(data= camera_data, 18, attr= get_attrs_ID(camera_data),
                rounds = get_rounds_by_ID(camera_data, userid= 18))

decision_matrix(data= camera_data, 18, rounds="all")
```

Returning the full decision matrix for the selected user:

```
##         attr1 attr2 attr3      attr4
## round0      2     0     0 0.31666667
## round1      2     1     0 0.16805556
## round2      2     0     0 0.31666667
## round3      2     0     2 0.01944444
## round4      0     0     2 0.31666667
## round5      0     0     3 0.16805556
## round6      0     0     2 0.31666667
## round7      0     0     1 0.46527778
## round8      0     0     2 0.31666667
## round9      0     2     2 0.01944444
## round10     0     0     2 0.31666667
## round11     1     0     2 0.16805556
## round12     0     0     2 0.31666667
## round13     0     0     3 0.16805556
## round14     0     0     2 0.31666667
```

Note that in the third option, we did not enter any `attr` value. Excluding the `dataset` and `userid` arguments, all arguments are assigned a default value by the functions, unless stated otherwise by the reader. In this case,

the default behavior without `attr` input is to calculate all attributes found in the given data. At this point it is important to note that default values and behavior can be changed, but should be well documented.

As the next step in our procedure comes the calculation of the reference point vector, `refps` in 'productConfig'. Ideally, the length of the `refps` vector is equal to the length of the `attr` vector, although the function `ref_points` has built-in capabilites to handle other scenarios. The package's user may enter its own values. If no value is given, `refps` determines the reference points as the default values for each attribute in the initial configuration. For our selected user they are:

```
ref_points(camera_data, userid= 18)
```

```
##     rp 1      rp 2      rp 3      rp 4
## 2.0000000 0.0000000 0.0000000 0.3166667
```

We discussed earlier the possibility of reference points being defined by the decision-maker as other than default values. There is also extensive literature about the use of multiple reference points, for a brief overview see [11]. At this point of development we worked only with single reference points ,default values, or status quo. It is but a major limitation and it presents great room for future package development in this direction.

Having determined the decision matrix and the reference points vector, we now calculate the normalized gain and loss matrices. After the next code sample we leave the attr and refps command out, since their default values calculate the same values we are using.

```
norm_g_l_matrices(data= camera_data, userid= 18,
                              attr = get_attrs_ID(camera_data),
                              rounds = "all",
                              refps = NULL)
```

```
## $ngain
##       [,1] [,2]      [,3] [,4]
##  [1,]    0  0.0 0.0000000  0.0
##  [2,]    0  0.5 0.0000000  0.0
##  [3,]    0  0.0 0.0000000  0.0
##  [4,]    0  0.0 0.6666667  0.0
##  [5,]    0  0.0 0.6666667  0.0
##  [6,]    0  0.0 1.0000000  0.0
##  [7,]    0  0.0 0.6666667  0.0
##  [8,]    0  0.0 0.3333333  0.5
##  [9,]    0  0.0 0.6666667  0.0
## [10,]    0  1.0 0.6666667  0.0
## [11,]    0  0.0 0.6666667  0.0
## [12,]    0  0.0 0.6666667  0.0
## [13,]    0  0.0 0.6666667  0.0
## [14,]    0  0.0 1.0000000  0.0
## [15,]    0  0.0 0.6666667  0.0
##
## $nloss
##       [,1] [,2] [,3]          [,4]
##  [1,]  0.0    0    0  0.000000e+00
##  [2,]  0.0    0    0 -5.000000e-01
##  [3,]  0.0    0    0  0.000000e+00
##  [4,]  0.0    0    0 -1.000000e+00
##  [5,] -1.0    0    0 -3.810036e-14
##  [6,] -1.0    0    0 -5.000000e-01
```

```
##  [7,] -1.0    0    0 -3.810036e-14
##  [8,] -1.0    0    0  0.000000e+00
##  [9,] -1.0    0    0 -3.810036e-14
## [10,] -1.0    0    0 -1.000000e+00
## [11,] -1.0    0    0 -3.810036e-14
## [12,] -0.5    0    0 -5.000000e-01
## [13,] -1.0    0    0 -3.810036e-14
## [14,] -1.0    0    0 -5.000000e-01
## [15,] -1.0    0    0 -3.810036e-14
```

For the final step of calculating the prospect values, we are only missing the attribute weights. The `WeightFunctions` provides different functions to calculate the weights. It is structured in such a way that it does this with what we call an interface function, named `get_attr_weight`. The idea is to simplify the calculation by providing the interface function with the necessary parameters and the name of the function you want to use. Nevertheless at the time of the publication of this seminar paper, only one weight function was deemed mature enough to include.

Weight functions are also a point to reflect on. To the best of my knowledge, there is no uniformly consensus on how to determine them. We are working in some creative ways to calculate them using the dataset the reader provides. However, we encourage more thoughts on this subject, to advance the status of this package. The standard function used at the moment `weight_higher_sum_value` weights each attribute according to the relative size of the sum of its values across all rounds.

```
weights_demo <- get_attr_weight(dataset= camera_data, userid= 18,
                weight= NULL,
                rounds= "all")
weights_demo
```

```
##      attr1      attr2      attr3      attr4
## 0.11841074 0.03947025 0.32891873 0.51320027
```

As you can observe, all weights are smaller than 1 and their sum `sum(weights_demo)` returns 1.

We have gone through all the necessary functions that lead to the higher level cluster `ProspectValueFunctions`. Within this group of functions we find two important steps, (1)the calculation of the value matrix and the following (2) determination of the overall prospect values for each one of the fifteen rounds.

Value matrix

```
pvalue_matrix(dataset= camera_data, userid= 18,
                rounds= "all",
                alpha= 0.88, beta= 0.88, lambda= 2.25)  ## attr, refps default
```

```
##            [,1]      [,2]      [,3]          [,4]
##  [1,]  0.000000 0.0000000 0.0000000  0.000000e+00
##  [2,]  0.000000 0.5433674 0.0000000 -1.222577e+00
##  [3,]  0.000000 0.0000000 0.0000000  0.000000e+00
##  [4,]  0.000000 0.0000000 0.6999060 -2.250000e+00
##  [5,] -2.250000 0.0000000 0.6999060 -3.494622e-12
##  [6,] -2.250000 0.0000000 1.0000000 -1.222577e+00
##  [7,] -2.250000 0.0000000 0.6999060 -3.494622e-12
##  [8,] -2.250000 0.0000000 0.3803061  5.433674e-01
##  [9,] -2.250000 0.0000000 0.6999060 -3.494622e-12
## [10,] -2.250000 1.0000000 0.6999060 -2.250000e+00
```

```
## [11,] -2.250000 0.0000000 0.6999060 -3.494622e-12
## [12,] -1.222577 0.0000000 0.6999060 -1.222577e+00
## [13,] -2.250000 0.0000000 0.6999060 -3.494622e-12
## [14,] -2.250000 0.0000000 1.0000000 -1.222577e+00
## [15,] -2.250000 0.0000000 0.6999060 -3.494622e-12
```

Overall prospect values

```
overall_pv(dataset= camera_data, userid= 18,
               rounds = "all",
               alpha = 0.88, beta = 0.88, lambda = 2.25)  ## attr, refps, weight default
```

```
##  [1]  0.00000000 -0.60597986  0.00000000 -0.92448843 -0.03621199
##  [6] -0.56493215 -0.03621199  0.13752194 -0.03621199 -1.15144236
## [11] -0.03621199 -0.54198074 -0.03621199 -0.56493215 -0.03621199
```

Although this walk-through has allowed us to ilustrate some functionality of our package, we capitalize this moment to show off the reasoning behind the strutcuring of 'productConfig'. Since the higher level function `overall_pv` is independent from the steps shown above, we can reproduce the exact same result just by running a simple command.

```
overall_pv(camera_data, 18, rounds = "all")
```

```
##  [1]  0.00000000 -0.60597986  0.00000000 -0.92448843 -0.03621199
##  [6] -0.56493215 -0.03621199  0.13752194 -0.03621199 -1.15144236
## [11] -0.03621199 -0.54198074 -0.03621199 -0.56493215 -0.03621199
```

One problem the reader might suggest is that the results are only for one user. This is where the `powerful_function` is particularly useful.

```
powerful_function(camera_data,
                  userid= get_all_userids(camera_data),
                  FUN= overall_pv,
                  rounds = "all")
```

For space purposes we will only show the results of four randomly picked users.

```
powerful_function(camera_data,
                  userid= c(54, 20, 6, 16),
                  FUN= overall_pv,
                  rounds = "all")
```

```
## $usid54
## [1]  0.0000000 -0.2930264  0.0000000 -0.2930264  0.0000000 -0.2930264
## [7]  0.0000000
##
## $usid20
##  [1]  0.0000000000 -0.0024465102 -0.3477875648 -0.0024465102 -0.3711682414
##  [6] -0.0024465102 -0.0009304227 -0.5092419678 -0.0009304227 -0.1411575478
## [11] -0.1117492972 -0.6200608423 -0.2430846082 -0.6200608423 -0.2430846082
## [16] -0.5089079865 -0.2430846082
```

```
##
## $usid6
##   [1]  0.00000000 -0.52240541 -0.36563466 -0.19867397 -0.21957955
##   [6] -0.24646294 -0.29983047 -0.21571471 -0.29983047 -0.21571471
##  [11] -0.43520366 -0.21571471  0.00000000 -0.52240541 -0.36563466
##  [16] -0.19867397 -0.11402736 -0.19867397 -0.36563466 -0.19867397
##  [21] -0.36563466 -0.52240541 -0.36563466 -0.19867397 -0.21957955
##  [26] -0.29983047 -0.21957955 -0.19867397 -0.36563466 -0.19867397
##  [31] -0.16168421 -0.50327560 -0.67023629 -0.50327560 -0.82700704
##  [36] -0.50327560 -0.30460163 -0.56527808 -0.15098001  0.00000000
##  [41] -0.25015722 -0.27106280 -0.25015722 -0.27106280 -0.43520366
##  [46] -0.21571471  0.00000000 -0.21571471  0.00000000 -0.36563466
##  [51] -0.19867397 -0.16168421 -0.20985313 -0.42990489 -0.15098001
##  [56] -0.21571471 -0.29983047 -0.54998768 -0.29983047 -0.42990489
##  [61] -0.20985313 -0.46001035 -0.24515720 -0.50327560 -0.30460163
##  [66] -0.56527808 -0.29983047 -0.54998768 -0.29983047 -0.42990489
##  [71] -0.20985313 -0.15098001 -0.30460163 -0.34121144 -0.58805432
##  [76] -0.34121144 -0.42362901 -0.57441052 -0.42362901 -0.08203762
##  [81] -0.21571471 -0.29983047 -0.24646294 -0.58805432 -0.32993592
##  [86] -0.46001035 -0.32993592 -0.58805432 -0.32993592 -0.46001035
##  [91] -0.20985313 -0.46001035 -0.21316746 -0.46001035 -0.20985313
##  [96] -0.42990489 -0.15098001 -0.21571471 -0.15098001 -0.21571471
##
## $usid16
## [1]  0.0000000 -0.4119789 -0.4724162 -0.2561427 -0.3307635  0.0000000
## [7] -0.3878132
```