3.3 Illustration of the productConfig package

Diego Aviles

2016-03-30

First, let us look at the data:

```
tail.matrix(camera_data)
```

```
##       cid usid round atid   selected selectable
## 1823 1835   62     1    4 0.01944444          1
## 1824 1836   62     2    4 0.16805556          1
## 1825 1837   63     0    1 1.00000000          2
## 1826 1838   63     0    2 1.00000000          1
## 1827 1839   63     0    3 1.00000000          1
## 1828 1840   63     0    4 0.16805556          1
```

As you can see our data displays 1828 rows with around 63 different users in a rather complex format which makes it practically difficult to work with. This is the reason we need the basic function cluster `GetFunctions`. For example, it is quite necessary to know how many attributes there are in out data:

```
get_attrs_ID(dataset=camera_data)
```

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

Given that our functions are mostly vectorized and assuming all users have the same attribtues, we can ask for the unique values of each `attr`.

```
getAttrValues(dataset=camera_data, attr = c(1,2,3,4))
```

```
lapply(temp, unique)
```

```
## $`1`
## [1] 3 0 2 1
##
## $`2`
## [1] 0 3 2 1
##
## $`3`
## [1] 0 3 2 1
##
## $`4`
## [1]  0.16805556 -0.27777778 -0.12916667  0.01944444  0.46527778  0.31666667
## [7]  0.61388889
```

Now that we know how many attributes there are, we also know how many columns the decision matrices are going to have. The number of rows depends on how much each user interacted with the product configurator and again, since functions are vectorised we can calculate the number of rows for all users using `getRoundsById`.

```r
all.rounds <- getRoundsById(camera_data, userid = getAllUserIds(camera_data))
head(all.rounds, 3) # To display only the results for the first three users
```

```
## $`6`
##   [1]  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
##  [24] 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
##  [47] 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68
##  [70] 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91
##  [93] 92 93 94 95 96 97 98 99
##
## $`9`
## [1] 0 1 2 3
##
## $`10`
## [1] 0 1 2 3 4
```

We can now easily observe that user 10 interacted four times with the configurator four times before making a decision.

The three functions presented above are necessary to create more complex structures, such as the decision matrix. To build it, we just need to use the right function with the right parameters. At mentioned earlier, the fourth parameter `attr=4` is price, which means it is a cost attribute (lower values are better). To handle this we input the correspondent attribute ID in `cost_ids`. Choosing any random user from our table, we calculate its decision matrix.

```r
decisionMatrix(camera_data, 33, rounds="all", cost_ids=4)
```

```
## $`33`
##        attr1 attr2 attr3       attr4
## 0round     1     1     1 -0.16805556
## 1round     1     1     2 -0.01944444
## 2round     1     0     2 -0.16805556
## 3round     2     0     2 -0.01944444
## 4round     1     0     2 -0.16805556
## 5round     1     0     3 -0.01944444
```

Notice how we did not specify the `attr` argument. As suggested before, aside from `dataset` and `userid` almost all arguments have a default value and perform a default behavior. When no input is entered `attr` calculates using all recognized attributes and `rounds` with the first and the last, which is why we explicitly specified `"all"`. Our next step is to determine the reference points. For the `refps` of PT we will use the default settings of user `33` which are:

```r
decisionMatrix(camera_data, 33, rounds="first", cost_ids=4)
```

```
## $`33`
##        attr1 attr2 attr3      attr4
## 0round     1     1     1 -0.1680556
```

This result should correspond to and validate our PT-reference-point function `referencePoints`.

```r
referencePoints(camera_data, 33, cost_ids=4)
```

```
## $`33`
##       rp 1       rp 2       rp 3       rp 4
##  1.0000000  1.0000000  1.0000000 -0.1680556
```

Now that we have determined the decision matrix and the reference points for user 33, we can proceed to compute the following steps.

[Insert quick figure]

However, since we have demonstrated how the functions build on each other and to avoid repetitiveness, we will calculate these matrices using only one function.

```r
pvMatrix(camera_data, 33, attr=1:4, rounds="all", cost_ids = 4,
         alpha = 0.88, beta=0.88, lambda=2.25)
```

```
## $`33`
##      [,1]  [,2]      [,3]         [,4]
## [1,]    0  0.00 0.0000000 0.000000e+00
## [2,]    0  0.00 0.5433674 1.000000e+00
## [3,]    0 -2.25 0.5433674 2.129512e-12
## [4,]    1 -2.25 0.5433674 1.000000e+00
## [5,]    0 -2.25 0.5433674 2.129512e-12
## [6,]    0 -2.25 1.0000000 1.000000e+00
```

Finally, for the final step of calculating the overall prospect values, we need to determine the attribute weights using the `WeightFunctions` cluster. Within it we have three weighting functions to our disposal, each mirroring a method described in section **??**: `differenceToIdeal, entropy, highAndStandard`. Using the entropy method we first need to normalize the *decision matrix* and pass it on to the respective function.

```r
norm.DM33 <- normalize.sum(decisionMatrix33[[1]])
```

```r
entropy(norm.DM33)
```

```
##     attr1     attr2     attr3     attr4
## 0.1962875 0.3616283 0.1962424 0.2458418
```

Since most functions are vectorialized, they return a list. For this reason, we use `[[1]]` to subset the matrix from the list, which can be then passed on to `normalize.sum`. Keeping such details in mind in addition to having to know which normalizing function to use with each weight method can be quite unefficient. Alternatively, we can computate the above results with an ‘interface’ function called `weight.entropy`, which also functions for the above functions, with their corresponding names.

```r
weight.entropy(camera_data, 33, attr=1:4, rounds="all", cost_ids = 4)
```

```
## $`33`
##     attr1     attr2     attr3     attr4
## 0.1962875 0.3616283 0.1962424 0.2458418
```

Later for the sake of consistency, to compare to DRP and TRP do input a `refps` for PT.