sdcTable Vignette

Bernhard Meindl <bernhard.meindl@statistik.gv.at>

February 23, 2017

Abstract

The purpose of the sdcTable vignette is to show how to get up and running with sdcTable; for details, including a complete list of options, consult the help pages or the manual for the following main functions of the package:

- makeProblem using e.g: help('makeProblem')
- primarySuppression using e.g: help('primarySuppression')
- protectTable using e.g: help('protectTable')
- setInfo using e.g: help('setInfo')
- getInfo using e.g: help('getInfo')

Contents

1 Introduction to `sdcTable`

R-package `sdcTable` is free and open source software that provides methods to generate instances of multidimensional, hierarchical table structures, identify primary sensitive table cells within such objects and finally protect primary sensitive table cells by solving the secondary cell suppression problem with currently 3 implemented algorithms.

2 How to protect data - An overview

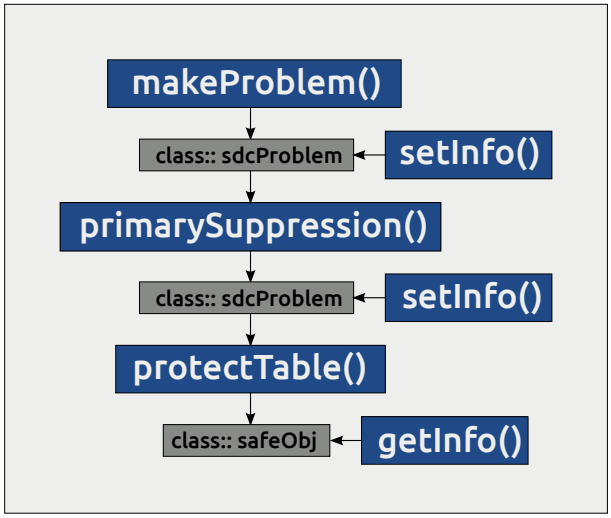

Figure 1: `sdcTable` - overview of exported functions

The main functions that are exported to users are shown in Figure (1).

Function `makeProblem()` is used to create objects of class `sdcProblem`. Instances of class `sdcProblem` hold the entire information that is required to perform primary or secondary cell suppression such as assumed to be known upper and lower cell bounds or upper-, lower- or sliding protection levels that are required to fulfill when solving the secondary cell suppression problem. All this information can be modified using function `setInfo()`.

`primarySuppression()` is applied to objects of class `sdcProblem`. By setting function parameters users can choose and apply a pre-defined primary suppression rule. Using `setInfo()` one can easily implement a custom primary suppression rule, too.

Function `protectTable()` is used to protect primary sensitive table cells in objects of class `sdcProblem`. A successful run of function `protectTable()` results in an object of class `safeObj`. Using `getInfo()` one can extract information from objects of such class, most importantly of course a data set containing all table cells along with the suppression pattern.

More detailed information on all the possibilities is available in the help-files, additional information is given in the corresponding sections of this vignette that deal with specific functions. The first step however to get started is to load the package, which can easily be done as shown below:

```
> library(sdcTable)
```

3 A simple example

We now walk through the steps that are required to protect tabular data using `sdcTable`. In the first example we are going to protect table cells given a three-dimensional tabular structure with some sub-totals.

We will start by discussing input data sets in sections (3.1) and (3.2). Then we continue by discussing how to define and describe dimensional variables in (3.3) which is a crucial step in the entire procedure. Once the hierarchies are defined it is necessary to create suitable objects (3.4) that can be used to identify and suppress primary sensitive cells. This is shown in section (3.5). Finally we discuss how to protect cells that are primary sensitive in section (3.6).

Throughout it is also shown how to set and extract information from the objects we are working with using functions `getInfo()` and `setInfo()`.

3.1 Starting from microdata

In this example we suppose we have collected data from 1000 individuals. A subset of the available data is shown below:

```
> print(head(microData), row.names=FALSE)
```

```
 V1 V2 V3 numVal2 numVal1
  A  w  d   49.93   71.72
  C  m  f   48.44   55.96
 Ba  m  a   43.20   64.69
 Bc  w  d   31.38   30.72
 Bc  w  a   49.46   54.93
 Ba  m  d   42.02   22.23
```

We note that the information we have obtained for any individual corresponds to exactly one row in the input data.frame. That is supposed to be available in `R`.

The micro data consist of 5 variables. The first 3 variables ('V1', 'V2' and 'V3') are categorical variables that will later define the table that needs to be protected. Variables 'numVal1' and 'numVal2' correspond to arbitrary variables containing some kind of information measured for each individual.

To create the tabular structure that is required to protect any table cells within the table it is of course of interest to have a look at possible values or characteristics of the categorical variables that define the table.

- Variable 'V1': this variable has a total of 6 codes without subtotals which are listed below:

  ```
  [1] "A"  "Ba" "Bb" "Bc" "C"  "D"
  ```

- Variable 'V2': this variable has a total of 2 codes without subtotals which are listed below:

  ```
  [1] "m" "w"
  ```

- Variable 'V3': this variable has a total of 6 codes without subtotals which are listed below:

  ```
  [1] "a" "b" "c" "d" "e" "f"
  ```

The step on how to define level-hierarchies that have to include all possible (sub)totals is explained in section (3.3).

3.2 Using aggregated data

Using `sdcTable` it is also possible to start with a 'complete' dataset. This means that the input dataset already contains rows with all possible level-combinations that can occur. This also includes combinations with (sub)totals. In this case it is required that the input data contain a column holding cell counts. Using the example data already discussed in (3.1), the complete dataset could be specified as shown below:

```
> print(tail(completeData))
```

```
     V1  V2 V3 Freq  numVal1  numVal2
163   C Tot  f   24  7523.30  7305.85
164   D Tot  f   31  7723.40  7698.75
165 Tot   m  f  107 24033.37 24325.32
166   B   m  f   55 12797.11 13372.83
167 Tot   w  f   85 25082.25 25459.33
168   B   w  f   50 12600.68 12861.13
```

Even though we only show a small subset of the data it is immediately clear that in object `completeData` (sub)totals are listed. These combinations can be calculated from the microdata by summation over several codes in one or more dimensional variables. As in (3.1) it is of interest which codes were specified for each dimensional variable. This information is given below:

- Variable 'V1': this variable has a total of 8 codes including all possible subtotals which are listed below:

```
[1] "Tot" "A"   "B"   "Ba"  "Bb"  "Bc"  "C"   "D"
```

- Variable 'V2': this variable has a total of 3 codes including all possible subtotals which are listed below:

```
[1] "Tot" "m"   "w"
```

- Variable 'V3': this variable has a total of 7 codes including all possible subtotals which are listed below:

```
[1] "Tot" "a"   "b"   "c"   "d"   "e"   "f"
```

We also note that in `completeData` a variable 'Freq' is available which gives information on the corresponding cell counts. This means that for example a total of 50 individuals contribute to the table cell where variable 'V1' equals 'B', variable 'V2' is 'w' and variable 'V3' is equal to 'f'.

Whether or not one starts to work with micro data or already with a complete, pre-aggregated dataset the next step is always the definition of the hierarchies defining the tabular structure.

3.3 Defining hierarchies

We could see in (3.1) and (3.2) that the set of codes available in the input data for variables 'V1', 'V2' and 'V3' differ since in the case where micro data are used as input data, no codes for subtotals are included in the micro data while in the case where pre-aggregated data are used those subtotals must already be included in the input data set.

When defining the complete hierarchies, no (sub)-totals must be excluded from the description. This means that for each variable defining one dimension of the table the complete structure must of course includes all (sub)totals.

In this example the hierarchies we want to define are quite basic. We start by showing the level-codes for each variable 'V1', 'V2' and 'V3' that are included in `completeData` but not in `microData`.

- (sub)totals of variable 'V1':

```
[1] "Tot" "B"
```

- (sub)totals of variable 'V2':

```
[1] "Tot"
```

- (sub)totals of variable 'V3':

```
[1] "Tot"
```

We observe that variable 'V1' has two codes ('Tot' and 'B') that can be calculated from the codes of 'V1' available in the micro data set `microData`. For variables 'V2' and 'V3' only one total value ('Tot') exists which means the summation over all characteristics of variables 'V2' and 'V3' is the (only) total value.

To specify the complete structure of a dimensional variable one needs to create a data frame or a matrix for each of those variables. The structure of any object describing a dimensional variable be created as follows:

1. the object must consist of exactly 2 columns, both being character vectors
2. the first column specifies levels
3. the second column specifies level-codes
4. the only allowed character in the first column is '@'
5. the length of the strings of the first column defines the (numeric) level of the corresponding code
6. a top-down approach has to be taken
7. the object must contain a row for each possible level-code

While this may sound difficult, it is in fact quite easy to create such objects within R. We will now explain how to create the required objects for the dimensional variables 'V1', 'V2' and 'V3' used in the example.

defining level-structure for variable 'V1'

The hierarchy we want to describe is as follows. The overall code 'Tot' is calculated from the codes ('A', 'B', 'C' and 'D'). Additionally, code 'B' (which is the second (sub)total-code for variable 'V1' as shown in section (3.2)) can be calculated from the level-codes 'Ba', 'Bb' and 'Bc'.

Following rule 1, we have to create a data frame or matrix consisting of two columns, the first specifying levels, the second column the corresponding level codes. Since we have to follow a top-down approach, the first level code must always correspond to the grand total which is always considered as the code with a level equaling 1. Thus, we create the matrix with a single row defining the overall total as follows:

```
> dimV1 <- matrix(nrow=0, ncol=2)
> dimV1 <- rbind(dimV1, c('@','Tot'))
> print(dimV1)

     [,1] [,2]
[1,] "@"  "Tot"
```

The level code for the overall total is '@' because according to rule 4 it is the only allowed character in the first column and it consists of exactly 1 character. Also, since the overall total is defined as level 1, the number of characters of the string '@' and the level of the overall total code 'Tot' matches.

The next step is to add additional codes. As mentioned before, codes 'A', 'B', 'C' and 'D' contribute the the overall total. Therefore we know that these codes are considered as level 2 codes and must be (according to the top-down approach) listed below the overall total code. Adding these codes to object `dimV1` is shown below:

```
> mat <- matrix(nrow=4, ncol=2)
> mat[,1] <- rep('@@',4)
> mat[,2] <- LETTERS[1:4]
> dimV1 <- rbind(dimV1, mat)
> print(dimV1)

     [,1] [,2]
[1,] "@"  "Tot"
[2,] "@@" "A"
[3,] "@@" "B"
[4,] "@@" "C"
[5,] "@@" "D"
```

We know that code 'B' is a subtotal that can be calculated from codes 'Ba', 'Bb' and 'Bc'. Since 'B' is a code of level 2, the codes contributing to it must be of a lower level, in this case of level 3. We show below how to add the codes to object `dimV1`:

```
> mat <- matrix(nrow=3, ncol=2)
> mat[,1] <- rep('@@@',3)
> mat[,2] <- c('Ba','Bb','Bc')
> dimV1 <- rbind(dimV1, mat)
> print(dimV1)

     [,1]  [,2]
[1,] "@"   "Tot"
[2,] "@@"  "A"
[3,] "@@"  "B"
[4,] "@@"  "C"
[5,] "@@"  "D"
[6,] "@@@" "Ba"
[7,] "@@@" "Bb"
[8,] "@@@" "Bc"
```

Now object `dimV1` contains all possible codes along with their levels. However, it not valid because the top-down approach is violated. This means that codes

that contribute to a (sub)total must be listed directly below it. If we would not change the order of object `dimV1`, `sdcTable` would assume that code 'D' can be calculated by summation over codes 'Ba', 'Bb' and 'Bc'. For this reason it is necessary to move this 'block' up so that it is directly below code 'B'. The required code and the resulting correct object describing the structure of variable 'V1' is printed below:

```
> dimV1 <- dimV1[c(1:3,6:8, 4:5),]
> #dimV1 <- as.data.frame(dimV1, stringsAsFactors=FALSE)
> print(dimV1, row.names=FALSE)
```

```
     [,1]  [,2]
[1,] "@"   "Tot"
[2,] "@@"  "A"
[3,] "@@"  "B"
[4,] "@@@" "Ba"
[5,] "@@@" "Bb"
[6,] "@@@" "Bc"
[7,] "@@"  "C"
[8,] "@@"  "D"
```

Using this information, `sdcTable` internally calculates all kinds of information on dimensional variables. So for example it is able to deal with codes that can be (temporarily) removed from the structure because it can be considered as a 'duplicate'. This is however not the case for this basic dimensional variable that has a total of 8 codes of which 6 are required to calculate information for the 2 (sub)totals.

defining level-structure for variable 'V2'

The creation of a suitable object that describes the hierarchical structure of variable 'V2' is easy. We are only dealing with one overall Total ('Tot') that is the sum of all codes listed in (3.1) for this variable.

The code how to specify an object that describes the structure of dimensional variable 'V2' is given below:

```
> dimV2 <- matrix(nrow=3, ncol=2)
> dimV2[,1] <- c('@','@@','@@')
> dimV2[,2] <- c('Tot','m','w')
> #dimV2 <- mat
> #dimV2 <- as.data.frame(mat, stringsAsFactors=FALSE)
> print(dimV2, row.names=FALSE)
```

```
     [,1] [,2]
[1,] "@"  "Tot"
[2,] "@@" "m"
[3,] "@@" "w"
```

We see that the overall total ('Tot') is again listed in the first row as level 1 ('@' has one character) while all contributing codes ('m' and 'w') are listed below the total and are of level 2 ('@@' has two characters).

defining level-structure for variable 'V3'

The creation of a suitable object that describes the hierarchical structure of variable 'V3' is easy. We are only dealing with one overall Total ('Tot') that is the sum of all codes listed in (3.1) for variable 'V3'.

The required code to generate an object specifying the hierarchical structure of variable 'V3' is given below:

```
> dimV3 <- matrix(nrow=7, ncol=2)
> dimV3[,1] <- c('@',rep('@@',6))
> dimV3[,2] <- c('Tot',letters[1:6])
> #dimV3 <- as.data.frame(mat, stringsAsFactors=FALSE)
> print(dimV3, row.names=FALSE)

     [,1] [,2]
[1,] "@"  "Tot"
[2,] "@@" "a"
[3,] "@@" "b"
[4,] "@@" "c"
[5,] "@@" "d"
[6,] "@@" "e"
[7,] "@@" "f"
```

It is required to create an object defining the complete structure and hierarchies for each dimensional variable. Once this step has been done, the multidimensional tabular structure that is required to apply any statistical disclosure methods can be created using `makeProblem()`.

3.4 Creating objects of class `sdcProblem` for further processing

We now show how to create objects of class `sdcProblem` which can further be used to identify, suppress and protect sensitive table cells.

It was discussed in (3.1) and (3.2) how micro data and pre-aggregated data can be used as data-input objects. We will now explain how to create instances of class `sdcProblem` from both `microData` and `completeData` and describe the required and optional parameters of function `makeProblem`.

We start building a suitable object of class `sdcProblem` starting with the micro data set `microData` discussed in section (3.1).

```
> dimInfo <- list(V1=dimV1, V2=dimV2, V3=dimV3)
> prob.microDat <- makeProblem(
+         data=microData,
+         dimList=dimList,
+         dimVarInd=1:3,
+         freqVarInd=NULL,
+         numVarInd=4:5,
+         weightInd=NULL,
+         sampWeightInd=NULL)
```

First we have to combine the objects describing the hierarchical variables 'V1', 'V2' and 'V3' into a list called `dimList`. Each list element is one of the objects created in section (3.3). The names of the list-elements must correspond to the variable name that the corresponding list-element refers to. In this case, the first list-element - 'dimV1' - should describe variable 'V1' in the input data set `microData` when calling `makeProblem()` while the second list element - 'dimV2' - defines the hierarchy of variable 'V2' and 'dimV3' - the third list element - describes the structure of variable 'V3'.

The remaining parameters are quite self-explanatory and shorty described:

- **data**: the data set that should be used, in this case `microData`
- **dimList**: a named list containing information on the structure of dimensional variables as described just above
- **dimVarInd**: the column indices of dimensional described in `dimList`.
- **freqVarInd**: if not `NULL` an index specifying the column that contains information on cell counts
- **numVarInd**: if not `NULL` an index specifying the columns holding other numerical variables
- **weightInd**: if not `NULL` an index specifying the column that contains info on weights that should be used in the secondary cell suppression problem instead of cell counts
- **sampWeightInd**: if not `NULL` an index specifying the column holding sampling weights for each person | group

Building an object of class `sdcProblem` using the complete, pre-aggregated data `completeData` as discussed in section (3.2) is very similar as it is shown below:

```
> ### problem from complete data ###
> dimInfo <- list(V1=dimV1, V2=dimV2, V3=dimV3)
> prob.completeDat <- makeProblem(
+         data=completeData,
```

```
+         dimList=dimList,
+         dimVarInd=1:3,
+         freqVarInd=4,
+         numVarInd=5:6,
+         weightInd=NULL,
+         sampWeightInd=NULL)
> #print(table(prob.completeDat@problemInstance@Freq))
```

The only difference is that in this case we define parameter 'freqVarInd' that specifies a column within the input data set `completeData` containing information on cell counts. Also the indices of argument 'numVarInd' are different to the first example.

In any case, both procedures return an object of class `sdcProblem` as can easily be checked:

```
> all(c(class(prob.microDat), class(prob.completeDat))=='sdcProblem')
```

```
[1] TRUE
```

We now can check if the cell counts of both objects are equal. Function `getInfo()` can be used to extract information from objects of class `sdcProblem`. Specifying argument 'type' as 'freq', `getInfo()` returns cell counts which of course are equal independent if micro-data or pre-aggregated data have been used as input to create the complete tabular structure.

```
> counts1 <- getInfo(prob.completeDat, type='freq')
> counts2 <- getInfo(prob.microDat, type='freq')
> all(counts1==counts2)
```

```
[1] TRUE
```

Once the problem has been set up and an instance of class `sdcProblem` is available, it is possible to identify and suppress sensitive table cells as we demonstrate in the next section using object `prob.completeDat`.

3.5 Primary suppression

Identifying and suppressing primary sensitive cells is usually done by applying function `primarySuppression()`.

Having a look at the cell counts in table `prob.completeDat` shows that a total of

```
> length(which(counts1 <= 10) )
```

```
[1] 15
```

cells have less than 10 individuals contributing to it. We think that these cells should be considered as primary sensitive and we want to have them protected. When creating an object of class `sdcProblem` all cells are assigned an anonymization state. The possible codes are listed below:

- **'u'**: cell is primary suppressed and needs to be protected
- **'x'**: cell has been secondary suppressed
- **'s'**: cell can be published
- **'z'**: cell must not be suppressed

The goal is now to change the anonymization status of all cells having less than 10 individuals contributing to it from the default value of 's' to 'u'. The easiest way is to use function `primarySuppression()` directly:

```
> prob.completeDat <- primarySuppression(prob.completeDat, type='freq', maxN=10)
```

Argument 'type' specifies the primary suppression rule we want to apply. In this case we want to use the frequency threshold rule that allows to suppress all table cells having cell counts less or equal than the threshold specified using argument 'maxN'. `primarySuppression()` also allows to apply the nk-dominance rule or the p-percent rule directly, in case micro data have been used as input data. For all possible parameters and their explanation the interested reader may consult the manual or the help-page of `primarySuppression()`.

After performing the suppression, we can have a look at the distribution of the anonymization states:

```
> print(table(getInfo(prob.completeDat, type='sdcStatus')))

  s   u
153  15
```

One can see that the 15 cells having counts less or equal than 10 have been identified and marked as primary suppressed. However, we should note that it is very easy to implement custom suppression rules by manually changing the anonymization state of cells using functions `setInfo()` or `changeCellStatus()`. Information on how to use these functions is of course provided in the manual and the corresponding help-pages.

To protect these cells by solving the secondary cell suppression problem one can go on to use function `protectTable()` as explained in the next section.

3.6 Secondary cell suppression using `sdcTable`

`sdcTable` provides the algorithms to protect primary sensitive table cells within objects of class `sdcProblem`. The algorithms that may be selected are:

- **OPT**: protect the complete hierarchical, multidimensional table at once. This algorithm is however only suitable for small problem instances.
- **HITAS**: solving the secondary cell suppression problem by applying a cut and branch algorithm to subtables that are protected in specific order
- **HYPERCUBE**: solving the problem using a heuristic that is based on finding geometric hypercubes that are required to protect primary sensitve table cells

We show how to protect the data using the three possible variants and its default parameters. For an extensive discussion on the possible parameters have a look at the manual or help page for function `protectTable()`.

```
> resHITAS <- protectTable(prob.completeDat, method="HITAS")
> resOPT <- protectTable(prob.completeDat, method="OPT")
> resHYPER <- protectTable(prob.completeDat, method="HYPERCUBE")
```

Having a look at the Output objects we can observe that the number of secondary suppressions required to protect the 15 primary sensitive cells (by default against exact re-calculation given sliding protection levels of 1 for each primary sensitive cell) differs.

Using the 'OPT'-algorithm, a total of 23 cells have been marked as secondary suppressions. When using 'HITAS'-algorithm, it was required to additionally suppress 25 cells. A total of 29 cells was selected and marked as secondary suppressions when the 'HYPERCUBE' algorithm was used.

One now easily get information from the resulting output objects that are instances of class `safeObj` by using function `getInfo()` or applying the `summary`-method. For the former we show how to extract the final data set which can be achieved as follows:

```
> finalData <- getInfo(resOPT, type='finalData')
> print(head(finalData))
```

```
  V1 V2 V3 Freq numVal1 numVal2 sdcStatus
1  A  m  a   12 3919.84 4090.31         s
2 Ba  m  a   18 4707.53 5101.81         s
3 Bb  m  a   18 4199.31 4186.62         x
4 Bc  m  a   10 3890.27 4084.40         u
5  C  m  a   11 3341.39 2988.84         s
6  D  m  a   13 3975.03 3873.34         s
```

As we can see above the final result data set contains all columns specified in the input data set along with another column 'sdcStatus' that specifies the anonymization state for each table cell.

For the latter we show how to apply the summary method. This can be done by applying the following code:

```
> summary(resOPT)

#########################################################
### Summary of the result object of class 'safeObj' ###
#########################################################
--> The input data have been protected using algorithm OPT.
--> The algorithm ran for 45 seconds.
--> To protect 15 primary sensitive cells, 23 cells need to be additionally suppressed.
--> A total of 130 cells may be published.

###################################
### Structure of protected Data ###
###################################
'data.frame':        168 obs. of  7 variables:
 $ V1       : Factor w/ 8 levels "A","B","Ba","Bb",..: 1 3 4 5 6 7 1 3 4 5 ...
 $ V2       : Factor w/ 3 levels "m","Tot","w": 1 1 1 1 1 1 3 3 3 3 ...
 $ V3       : Factor w/ 7 levels "a","b","c","d",..: 1 1 1 1 1 1 1 1 1 1 ...
 $ Freq     : num  12 18 18 10 11 13 19 12 12 18 ...
 $ numVal1  : num  3920 4708 4199 3890 3341 ...
 $ numVal2  : num  4090 5102 4187 4084 2989 ...
 $ sdcStatus: chr  "s" "s" "x" "u" ...
NULL
```

We see that the summary provides all kind of useful information such as the algorithm that has been used to protect primary sensitive cells, the time it has been taken to solve the problem, the number of primary sensitive and secondary suppressed cells as well as the number of cells that may be published. Also, a excerpt of the final data set is shown.

I would also like to mention that an iterative algorithm is available in function `protectLinkedTables()` that allows to protect two tables that have common table cells. The function takes two objects of class `sdcProblem` as input and a list defining the common cells in both tables. Details on how to construct this a list-element are given in the manual and help-page of `protectLinkedTables()`.

4 Remarks

A lot of work has gone into the rewrite of `sdcTable` using S4-classes and methods in order to robustify the code and in order to make it easier in future to add new algorithms such as rounding- or cell-perturbation methods and features.

I would really like to hear any kind of feedback and will be more than happy to work in patches you submit or ideas any one might have which would make it easier to work `sdcTable`. Also, the next step in the evolution of the package will be performance optimization, evaluation for possibilities of parallel computing and so on. I would really like to hear any kind of feedback on package users on

these kind of things. Thus, for any remarks, please do not hesitate to contact me using my e-mail adress `bernhard.meindl@statistik.gv.at`.