WP. 10
ENGLISH ONLY

UNITED NATIONS ECONOMIC COMMISSION FOR EUROPE (UNECE)

CONFERENCE OF EUROPEAN STATISTICIANS

EUROPEAN COMMISSION

STATISTICAL OFFICE OF THE EUROPEAN UNION (EUROSTAT)

Joint UNECE/Eurostat work session on statistical data confidentiality
(Tarragona, Spain, 26-28 October 2011)

Topic (ii): Software research and development

A computational framework to protect tabular data – R-package sdcTable

Prepared by Bernhard Meindl, Statistics Austria

A computational framework to protect tabular data - R-package sdcTable

Bernhard Meindl*

* Department of Methodology, Statistics Austria, Vienna, Austria,
bernhard.meindl@statistik.gv.at

Abstract. In this contribution we give an overview about recent developments done in R-package `sdcTable`. `sdcTable` is free and open source software that is available on the R comprehensive archive network `http://cran.r-project.org`. It provides methods to solve the secondary cell suppression problem for multidimensional and hierarchical tables.

1 Introduction

Over the last few months a lot of work has gone into `sdcTable` resulting almost in a complete rewrite of the software. In fact, the redesign of the resulted from serveral weaknesses of the original package. To overcome existing limitations the starting point was to set up and define clear design goals for the new implementation of `sdcTable`.

It must be noted that at the time of writing, the new package has not yet been uploaded and published on CRAN. However, at the time of the Joint UNECE/Eurostat conference in Tarragona, the package will already be published. Also, the new package will come bundled with a package-vignette including a complete real-life example that will help users getting started to protect data using `sdcTable`.

2 Design goals

The following design goals have been kept in mind while developing the new version of package `sdcTable`.

Make extensive use of S4-classes: One of the main concepts during the development process that was used is data encapsulation. This means that data-structures that often occur are modeled within `R` as S4-classes. S4-classes can be considered as data objects having one or more slot. Each slot can hold any data structure, primitive types such as vectors, lists or objects of any existing class that has been defined.

Each S4-class consists also of a validity function that ensures that an instance of a class cannot be created or modified if any slot violates the class-definition. Furthermore, it is possible to write additional validity checks that ensure that instances of a given class do not violate additional constraints that are checked in the custom validity function which is called whenever an instance of a class is created or modified. These automatic integrity checks automatically lead to robustification of the software help finding errors early during the implementation.

Use S4-methods whenever possible: Once all S4-classes have been defined it is the next step to write methods that can be applied to specific classes. The advantage of using S4-methods is that since these methods are always applied to objects of well-defined classes, a lot less error-checking is required. One can always safely assume that the input objects are valid. Thus, using S4 methods leads to a cleaner implementation and robustification of the underlying algorithms.

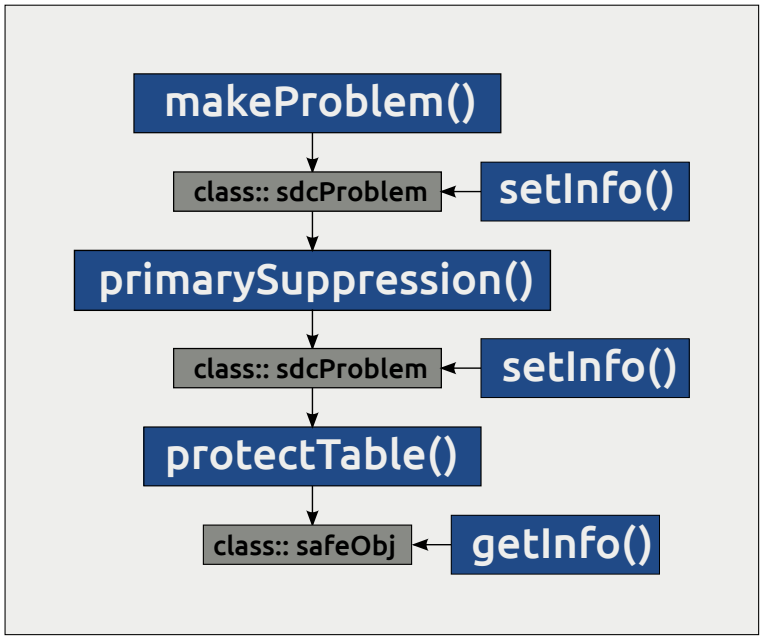

Figure 1: Overview on user-visible functions in package `sdcTable`

abstract as much detail from users as possible: While it is possible for power users to view and modify the source code to fit their personal needs, it was always an important target to abstract as much details of the implementation from end-users as possible.

A user who wants to protect tabular data should be able to get started quickly. Therefore, only a couple of functions are exported to users and can (and should) be used. The main functions of the new package `sdcTable` that are available to a user are shown in Figure (1).

Function `makeProblem()` must be used to create objects of class `sdcProblem` which forms the base of all further steps. Default parameter that have been set while creating the data object such as assumed known lower and upper bounds can be modified using function `setInfo()`. Due to the fact that the underlying code is completely based on S4 classes and methods, modification is only possible if all validation checks are passed.

`primarySuppression()` can finally be applied to objects of class `sdcProblem`. By setting function parameters, the user can choose an existing primary suppression rule and set corrsponding parameters. Currently the frequency rule, the p-percent rule and the nk-dominance rule are supported. However, using function `setInfo()` one can very easily implement custom primary suppression rules.

Function `protectTable()` can then be applied to objects of class `sdcProblem`. Depending on the function parameters that have been set, primary sensitve table cells are protected using either a cut and branch based algorithm (HITAS) that protects sub-tables in certain order, using an implementation based on the GHMITER algorithm (HYPERCUBE) or in an 'optimal' way (OPT). This is however only possible for small problem instances. A successful run of function `protectTable()` results in an object of class `safeObj`. Using `getInfo()` one can extract all kind of useful information from objects of such class, most importantly of course a data set containing all table cells along with the suppression pattern.

document everything: To help getting users (and developers) getting started with `sdcTable` a main goal was to document everything. Even though most methods will not be exported and thus will not be visible to users, all methods are properly documented in the source code. Also, `sdcTable` comes along with a package-vignette that provides a real-world example with a hints in addition to the runable code that is provided in the help files that are available for each exported and visible function.

3 An example

In this section we will walk through the process of how `sdcTable` can be used to protect a tabular structure that is based on two variables. We start from underlying micro data and show how to create an object that for which primary sensitive cells

can be identified and suppressed. Finally, the suggested way to perform secondary cell suppression and to extract information from the resulting object are discussed.

3.1 Underlying data

Suppose one has a table featuring information on age, region and the corresponding frequencies of 100 people. The aggregated data by gender and the four different regions could are supposed to be available as a data frame object `dataset` in `R` and are shown below:

```
> print(dataset, row.names = FALSE)

 gender   region Freq
 female Region A    2
   male Region A   24
 female Region B   12
   male Region B    9
 female Region C   11
   male Region C   15
 female Region D    9
   male Region D   18
```

We see that there is one cell (females in Region A) to which only 2 people contribute. The goal is now to protect this cell using `sdcTable`.

3.2 creating an instance of an `sdcProblem`:

The first step is to create a suitable object of class `sdcProblem` that can then be used to identify the primary sensitive cell according to the frequency rule and find a suitable suppression pattern to protect this sensitive table cell. To do so, we need to call function `makeProblem()`. The required input for this function is:

- **data:** in this case we will use the aggregated dataset shown in section 3.1. It is however also possible to make direct use of the micro as pre-aggregation of data is not necessary.
- **information on the structure of the variables defining the table:** in this case there are two variables (`gender` and `region`) defining the table. Both variables are hierarchical in the sense that for both variables a total (the sum of the level-codes) exists. Since the hierarchical structure of variables can be quite complex, a standardized way to recode the variables is required. Therefore, a use must provide information on all possible levels of the variables defining the table to the program. This information has to be provided in a well-defined format, which is described below for the two variables in the example.

- **additional information:** the function can also deal for example with data for which sampling weights are available and need to be specified. The documentation of `makeProblem()` gives a lot of information and examples about all parameters that can be set.

To specify information about dimensional variable `gender` one has to provide a data frame that contains exactly two columns. The first column specifies the levels while corresponding codes that have to be provided in the second column of a dataframe. It is important to note that the complete hierarchy needs to be specified. This means that rows for all possible characteristics for a given dimensional variable need to be listed, even codes that are not available in the input data set because the complete data structure is internally built from scratch using this information.

Levels are listed using strings consting only of `'@'`s. The number of characters is then used as the (numeric) level of the corresponding level-code. The most simply hierarchy consists of several codes that add up to exactly one total. This is the case for the dimensional variables `gender` and `region` of this example. One would have to provide a data frame for dimensional variable `gender` as listed below:

```
> print(levelInfo.gender, row.names = FALSE)

 levels  codes
      @  Total
     @@   male
     @@ female
```

The first row specifies the Total of this hierarchy that is simply calculated by summation over the two characteristics `male` and `female` that are both found in the input-dataset. The information has to be provided using a top-down approach starting from the total that is defined as level 1 (`@` has one character). Below the corresponding level are listed. Both characteristics are of level 2, thus two characters (`@@`) are required to model this information.

In a similar way the required information about the dimensional variable `region` is specified. The total is again specified in the first row and the 4 characteristics that contribute to the total are all of level 2, thus requiring 2 characters in the code specifying the level in the first column as one can see below.

```
> print(levelInfo.region, row.names = FALSE)

 levels    codes
      @    Total
     @@ Region A
```

```
@@ Region B
@@ Region C
@@ Region D
```

However, level structures can be quite complex and sub-levels are inserted in place. For example if we had some sub-regions which together form `Region B` one would have to insert rows below this region with levels consisting of three characters (`@@@`) and the corresponding codes. However, in this case these characteristics must be available in the input data set instead of the information for `Region B` because all values for this region could be calculated from the codes of the lower levels.

The next step is to call `makeProblem()` with apropriate parameters. In this simple example we have to provide the data as shown in (3.1), a list whose elements contain data-frames holding level-specifications which names of the corresponding variables within the input data set. Furthermore it is required to specify the indices of the dimensional variables within the input dataset and providing information to the software if the input data are microdata are already pre-aggregated. Since we are dealing with the second case, we have to set parameter `isMicroData` to `FALSE`.

```
> # a list containing information on levels
> dimList <- list(levelInfo.gender, levelInfo.region)
> names(dimList) <- c('gender', 'region')
> prob <- makeProblem(
+         data=dataset,
+         dimList=dimList,
+         dimVarInd=1:2,   # indices with dimensional variables
+         freqVarInd=3,    # index with cell counts
+         isMicroData=FALSE
+ )
```

The result object `prob` after calling `makeProblem)` is of class `sdcProblem` and will be used as input for `primarySuppression()` and `protectTable()`.

3.3 Primary cell suppression

The identification of primary sensitive table cells is usually achieved by applying function `primarySuppression()` to objects of class `sdcProblem` such as object `prob` that was created in (3.2). In this simple example we show how to use the frequency-rule that will identify and mark as primary sensitive all cells that have counts below a given threshold. We note that it is easily possible to implement a custom primary suppression rule using function `setInfo()` which is of course documented in the package itself.

Marking all table cells with frequencies below a threshold of 4 is done as it is shown below. Specifying `type=='freq'` selects to apply the frequency rule and passing parameter `maxN` changes the default value for this rule. The function call to mark cells with counts below 4 as sensitive is given by:

```
> prob <- primarySuppression(prob, type = "freq", maxN = 4)
```

This result again in an object of class `sdcProblem`. Having identified primary sensitive cells it is required to protect these cells by eventually finding additional suppressions that are needed to adequately protect sensitive table cells.

3.4 Secondary cell suppression

The secondary cell suppression problem can be solved using `protectTable()`. By default each primary sensitive cell is considered as protected if it is not not possible to exactly recalculate its cell value. This can of course be changed using function `setInfo()` and changing required upper- lower- or sliding protection levels. For now we keep the defaults and want to find additional suppressions using `protectTable()` with the optimal algorithm since we the problem is that simple. All parameter that can be passed to the function are again documented in the package-manual.

```
> result <- protectTable(prob, method = "OPT", verbose = FALSE)
```

After a successfull run the object `result` is of class `safeObj`. It is then easy to extract the required information from such an object. One could use `getInfo()` to extract the final data set that contains all level-codes specified in the hierarchy definitions along with cell counts and the anoymization states for each table cells.

```
> getInfo(result, type = "finalData")
```

```
   gender   region Freq sdcStatus
1  female Region A    2         u
2    male Region A   24         x
3  female Region B   12         x
4    male Region B    9         x
5  female Region C   11         s
6    male Region C   15         s
7  female Region D    9         s
8    male Region D   18         s
9   Total    Total  100         s
10   male    Total   66         s
11 female    Total   34         s
12  Total Region A   26         s
```

```
13  Total Region B   21         s
14  Total Region C   26         s
15  Total Region D   27         s
```

In this case we see that we are dealing with 1 (`sdcStatus=='u'`) for which a total of 3 additional cells needed to be suppressed (`sdcStatus=='x'`). Therefore, a total of 11 (`sdcStatus=='s'`) may be published for this simple table.

4 Conclusion and Outline

The design principles that were adhered to during the redesign of `sdcTable` make it easier to expand, change, modify and adjust the software.

Due to the changes in the underlying code-base, it will also be easier to include new algorithms to protect sensitive information in tabular data such as rounding or perturbation algorithms. The clean package design and definition of well-defined classes for data-structures that are used throughout the software also make it simpler to test or include different solvers that are required when trying to solve the secondary cell suppression problem using (mixed) linear programming techniques. Automatic validation checks lead to a more robust implementation.

The next step in the evolution of `sdcTable` will be extensive testing and performance optimization. But since the package design is modular, it is easy to measure, test and probably to improve the performance on several key-steps of the algorithms. Due to the existence of `R`-packages such as `Rcpp` it is also possible to include C/C++ code in `R` which can of course be exploited to improve performance even further.

Another important step in the future development will be the improvement of the functions and possibilities of the package that are available to users by taking into account the feedback of people actually working with the software. However, there are no plans yet to provide a graphical user interface for the software as feedback has already shown that calling the software using batch-jobs using the command line interface is the preferred way for most subject matter specialists working with the software.

References

Fischetti, M. and Salazar, J. J. (1999) Models and Algorithms for the 2-Dimensional Cell Suppression Problem in Statistical Disclosure Control. *Mathematical Programming*, **84**, 283–312.

Fischetti, M. and Salazar, J. J. (2001) "Solving the Cell Suppression Problem on Tabular Data with Linear Constraints", *Manage. Sci.*, **47/7**, 1008–1027.

de Wolf, P. (2002) "HiTaS: A Heuristic Approach to Cell Suppression in Hierarchical Tables", *Inference Control in Statistical Databases, From Theory to Practice*, *ISBN 3-540-43614-6*, 74–82.

Repsilber, D. (2002) "Sicherung persönlicher Angaben in Tabellendaten", *Statistische Analysen und Studien Nordrhein-Westfalen*.

R Development Core Team (2007). R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria. *ISBN 3-900051-07-0*.

Meindl, B. (2009). sdcTable: Statistical disclosure control methods for tabular data. published online.
<http://cran.r-project.org/web/packages/sdcTable/index.html>.

Eddelbuettel, D. and Francois, R. (2011). Rcpp: Seamless R and C++ Integration. published online.
<http://cran.r-project.org/web/packages/Rcpp/index.html>.