Methods library of embedded R functions at Statistics Norway

Øyvind Langsrud (Oyvind.Langsrud@ssb.no)
Statistics Norway

ABSTRACT

Statistics Norway is modernising the production processes. An important element in this work is a library of functions for statistical computations. In principle, the functions in such a methods library can be programmed in several languages. A modernised production environment demand that these functions can be reused for different statistics products, and that they are embedded within a common IT system. The embedding should be done in such a way that the users of the methods do not need to know the underlying programming language. As a proof of concept, Statistics Norway soon has established a ***methods library*** *offering a limited number of methods for macro-editing, imputation and confidentiality. This is done within an area of municipal statistics with R as the only programming language. This paper presents the details and experiences from this work. The problem of fitting real word applications to simple and strict standards is discussed and exemplified by the development of solutions to regression imputation and table suppression.*

Keywords: *Official statistics, R; Common Statistical Production Architecture, Generic Statistical Information Model, Validation and Transformation Language, Imputation, Statistical disclosure control*

JEL Classification: *C18, C88*

1 INTRODUCTION

Statistics Norway has started a process to modernise the statistical production and a long term modernisation program covering all aspects has been established. A first step towards building new IT systems has already started within a project dealing with municipality state reporting (KOSTRA). This involves a collection of 20 statistical areas about services provided by the municipalities and the accounting for these services. These statistics are based on several sources: municipality survey, files from local administrative systems and registers. Within this project a modern platform for common metadata and data storage has been built. *Validation and Transformation Language* (VTL) will be one of the main tools for working with the data (SDMX Technical Working Group, 2016). In addition a methods library has been built to offer common methods throughout the production process. These methods are made to be compatible with the *Common Statistical Production Architecture* (CSPA) described in UNECE (2017a). The methods will be made available within the IT system in a way that is independent of the underlying programming language. Within the so-called

administrative module, technical information about the methods (parameters) and short descriptions for users are stored. The methods are supposed to be functions that take a single data set as input in addition to parameters. Output is also a single data set. In principle, the functions in the methods library can be programmed in several languages. So far the new system is limited to R as the only programming language. Within the IT system, Java interacts with R via *RServe* installed on a Linux platform. The possibility of using *Renjin*, which is a Java implementation of R, was also investigated. However, at present, many common R-packages for official statistics are not compatible with Renjin. The architects are also looking at *OpenCPU* as a possible future interface to R.

The experiences obtained within the KOSTRA project will be very useful for planning the future expansion of the methods library. Below we describe some details and discuss some experiences from the work of establishing a methods library in R. Other and less R oriented details are presented in Foss et al. (2017). Section 2 describes the programming standards used for the embedded R functions. Sections 3 and 4 contain some details about the implemented methods for regression imputation and table suppression. Section 5 remarks that the R functions can be useful in several ways. Finally Section 6 ends with conclusions and discussions.

2. STANDARD FOR R PROGRAMMING

2.1 Strict standard for the library functions

It is important that the R functions used within the IT system follow a standard. From an R-programmers viewpoint the standards were set to be rather strict, but this makes building an interface to the methods easy. As earlier documented in Foss et al. (2017) the standard says:

- The first input parameter is a data set of type *data.frame*.
- The other parameters must be vectors (normally of length one) of type *character*, *numeric*, *integer* or *logical*.
- A special variant of the character type is variable name, referring to variables in the input data set.
- One such variable name parameter represents a unique identifier of observations.
- Another special variant of the character type is *list*, which means that a list of allowed input elements is pre-specified.
- When input is numeric or integer, the minimum and/or maximum can be specified.
- Output is a single data set of type data frame.

According to the discussion below, the data frame type was chosen as the standard for input and output data sets. The system in which the

functions are embedded handles a single data set at a time. That is, only a single data frame can be input and only a single data frame can be output. In many applications within official statistics one may view this as a strange limitation. For example some methods need both population data and sample data as input. Furthermore, several methods produce output at two or more hierarchical levels. It is always possible to force several data frames into a single data frame by using replicated and/or missing values. In the actual project, this solution was applied in order to produce output in some cases. Although, in more general future systems we believe that the single data set standard is too strict.

The other input parameters are ordinary data types. We have avoided using factor as a data type since this can be complicated and confusing in external communication. For the same reason we think it may be best to avoid using factor as data type in the output data frame (stringsAsFactors=FALSE). We have distinguished between numeric and integer as data types, but the exact data type is not the point. The important distinction is whether input and output are restricted to be whole numbers or not.

Since all the data are passed to the function via a data frame, parameters referring to variables in the data frame are needed. The standard is to pass these as variable names, but variable number could also be used. In fact, we made an implementation where both names and numbers can be used.

2.2 A general R programming standard

We have developed an internal R programming guide. The starting point was Google's R style guide. We collect the R code in packages and the functions are documented using *roxygen2* which means that documentation and code are side-by-side on the source files.

An important part of the guidance is to implement computations in functions that only do computations. Plotting and reading and writing of data are done separately or in functions that make a call to a computing function.

Another part of the guidance is to make the code as portable as possible. This is in accordance with the CRAN repository policy which says: "*Package authors should make all reasonable efforts to provide cross-platform portable code*". I practice one can say that the code is as portable as possible if the code is compatible with Renjin (a java implementation of R). This is an important reason for choosing data frame as the input and output standard for library functions. Internally in the functions, when a general data set is needed (instead of matrix), we also use data frame as standard. Operations in R involving data frames can, however, be rather slow and memory consuming compared to alternative implementations. In particular, the packages *data.*

table and *dplyr* offer efficient alternatives to data frame. Such packages can be used if this in practise makes very important improvements. Note that in many cases there are good reasons to make use of other packages not compatible with Renjin.

2.3 Function signature and input checking

We can say that a documented function consists of three parts: the head, the body and the function documentation. In general the function signature needed to integrate the function into another system cannot be captured from the function head alone. As an example, the head of an ordinary R function is `var(x, y=NULL, na.rm=FALSE, use)`. We can only read variable names and default values from the head. In the function documentation we can find out that the last parameter is optional and must be one of the strings "everything", "all.obs", "complete.obs", "na.or.complete", or "pairwise. complete.obs". The head of a related function is `cor(x, y=NULL, use="everything", method= c("pearson","kendall","spearman"))`. In this case one may think that the default of the last parameter is a vector of length three. But the documentation says that the parameter is one of "pearson" (default), "kendall", or "spearman". This function is an example where the function *match.arg* is used inside the body. Then the specification in the head has another meaning and allowed input is specified together with the default. In this case an appropriate error message is produced when the input is not valid. One could make use of this functionality in the methods library functions, but match.arg is limited to the character type.

To do input checking in general we need another approach and the aim is twofold. First, the input should be checked according to the requirements. Second, it should be easy to capture the requirements from the source file in the sense that it is should be both human readable and machine readable. We have discussed two possibilities. One way is to make an extended variant of match.arg so that everything is visible in the function head, both variable type and allowed input. We chose an easier a more straightforward variant. A function named *CheckInput* was made to be used at the beginning of the function body. Below are three example lines.

- `CheckInput(var1, type="character", alt=c("pearson","kendall","spearman"))`
- `CheckInput(var2, type="integer", min=0, max=99, okSeveral=TRUE)`
- `CheckInput(var3, type="varNrName", data=data)`

The first line corresponds to the parameter "method" in the function "var" mentioned above. The default value ("pearson") cannot be seen in this line, but this appears in the function head. The second line requires a vector of nonnegative whole number(s) below 100. The parameter that is checked in the third line refers to a variable in the data frame, "data". A special type is constructed that allows both variable names and variable numbers. Alternative variable types are "varNr" and "varName". Sending the whole data frame to the function when only the variable names are needed may look superfluous. But note that even if the syntax is pass-by-value, the data frame will not be copied in memory.

The function CheckInput is constructed to produce standardised error messages. In addition, by looking at the function head together with the lines with CheckInput, we can see a detailed signature of the function. It is also possible to make programs that read this information automatically. As simple example of extracting this information is `lapply(as.list(parse(text=s)),as.list)`, where "s" is a character vector whose elements are lines with CheckInput.

A drawback is that, since CheckInput is in the body, the content will not be visible in the function documentation. In practise the information will be repeated in the documentation text. So far, at Statistics Norway, the information is also repeated manually in the administrative module. This module contains necessary information about the methods needed to create a user interface in Norwegian. Thus, the English function and parameter names are translated to Norwegian. Short user documentation is also given in Norwegian in this module. More comprehensive user documentation is available on an internal wiki page.

By using CheckInput the checking is limited to other parameters than the data set. Checking the input data frame according to the requirements are more advanced and we have, so far, not set a standard for this. One possibility is to make use of the package *checkmate* (Lang, 2017), but such a standardisation will be a part of future discussions.

2.4 Function internal data

As an example, suppose that we have a function that performs ordinary linear regression (`y~x`). Within the function it is appropriate to write the code using the names *x* and *y* even if other names are used in the input data. It is also natural that the output data set use standard names such as "fitted", "resid" and "rstudent". When *y* is a matrix with an arbitrary number of columns, the original names may be needed in output. Internally within the function one would then use a matrix named *y* with column names from input. Similarly

x could be a matrix. This means that sometimes input names are visible in output and sometimes not.

Extracting *x* and *y* from the input data using the input parameters are straightforward. However, we have decided to use a standardised function, *GetData*, for this. In the first place the programmers can assume that this function is defined as

```
GetData <- function(data, ...) {
 a <- unlist(list(...))
 b <- data[, a, drop=FALSE]
 colnames(b) <- names(a)
 b }
```

This function returns a data frame with only the required columns and with new column names. For example, `GetData(data, x=xVar, y=yVar)`, returns a data frame with variable names "x" and "y" instead of the original names represented by *xVar* and *yVar*. In practise, a more advanced version of GetData is used. With vector input (e.g. `y=c(yVar1, yVar2)`), the returned data frame contain a matrix embedded in one variable.

To discuss another problem we can assume that the original data frame contain the variables idVar, xVar, yVar and yearVar. Suppose that we want to run regression with *y* taken from year 2017 and *x* from year 2016. In this case one possibility is to create a new data set beforehand, possibly within a wrapper function. How to solve this problem is part of a bigger discussion of how to create an interface between R and the other system.

The chosen solution for us was to extend the functionality of GetData. In this case one could make this call: `GetData(data, id = list(id="idVar"), x=list("xVar", yearVar=2016), y=list("yVar", yearVar=2017))`. The ordinary character input is replaced by list output with the variable name as the first element. This element is unnamed except in the case of the identifier variable. Other elements are used to specify rules for extracting the data. The output data is created in a way that uses the identifier variable when matching. A function that makes use of GetData can similarly be called by using lists instead of variable names. The programmer of the function does not need to thick about this. When implementing a function in practise, one may or may not make use of this functionality. The system at Statistics Norway makes use of the advanced functionality in some cases.

2.5 Wrapping functions to fit the standards

Assume that a researcher has developed a method that is to be programmed as a methods library function. In such cases the standard for the methods library can be viewed as being too strict. The best approach is then to make a good function without considering the standards. This function can be made very general and several ideas can be included. Such a function may also be shared with others (within a published package). To fit the standards, this function can be wrapped into one or several library functions. It is much easier to make a restricted function from a general function than it is to go the other way around.

Instead of programming a method from scratch, code already available in an open source package may be re-used. The programming job is then to wrap available function(s) into a library function. Many packages have been programmed in an object oriented way which is somewhat different from the single function approach of the methods library. Then, several functions need to be called. Typically, an initial object is first created from the input data and thereafter other functions are called.

3 REGRESSION IMPUTATION

The starting point for this work was to re-implement in R the imputing methodology that has been used in practice for several years (Thorud et al., 2011). The two most important methods are ordinary linear regression and the ratio model (weighted regression without a constant term). Within this approach, the observations are categorised into three groups:

- *Representative*: These observations are used for imputation modelling.
- *Non-representative*: These observations are not used for imputation modelling, but the values are kept.
- *Missing or wrong*: These observations are to be imputed.

Obviously, missing observations are in the last group. Beyond that, the categorisation of the observations is normally based on externally studentized residuals. This means that two limits are needed. Observations with studentized residuals below *limitModel* are considered as representative. Similarly, observations with studentized residuals above *limitImpute* are considered as wrong. However, extreme values influence the calculation of the studentized residuals and therefore we decided to introduce a third limit. In an iterative process, observations with studentized residuals above *limitIterate* are successively thrown out of the model.

Instead of implementing the specific requested methods directly, a general function, named *LmImpute*, was made. In addition to the data set

and the three limit parameters, *model* is an important input parameter to this function. This parameter is a general formula that can involve several dependent variables and several independent variables. LmImpute also supports transformations and weights. Furthermore, LmImpute can calculate total estimates with standard errors within requested groups of observations. It was decided to call *lm* within LmImpute, but this choice was not obvious. The support for multiple responses when using lm is limited and sometimes wrong results are produced.

By making LmImpute very general it could be the working horse within several library functions. The function *ImputeRegression* performs imputations within several strata and a limited number of pre-specified models can be chosen. Another function, *ImputeRegression2*, performs imputation in two rounds. First a primary dependent variable is used and thereafter a secondary variable is used for cases where the primary is missing. The standard error estimates take into account variability from both imputation models. In this function, observations imputed in the first round are forced to be non-representative (see above) in the second round. The function *ImputeRegressionMulti* performs simultaneous imputation of several dependent variables. Direct imputation of the most resent non-missing historical value is performed by the function *ImputeHistory*. Then, standard error estimates are based on the naive model where the difference between the current and the historical value is assumed to be pure error. A trick is used to utilize the function LmImpute also in this case. Finally, we also have a function called *OutlierRegression* which is similar to ImputeRegression, but imputation is not performed. This function is only used to find outliers using a limit for studentized residuals.

4. TABLE SUPPRESSION

The request was to make an R function for the methods library that could do table suppression according to a frequency rule. The starting point of this work was the package *sdcTable* (Meindl, 2017). This is an example of a package programmed in an object oriented way. Then, in order to satisfy the standards for the methods library, several functions need to be combined into a single function.

One of the input parameters needed by sdcTable is much more complicated than our standard input types. The parameter *dimList* contains necessary information about level-hierarchy coded in a certain way. For our problems, this information is hidden in the data and it is possible to create dimList automatically. Sorting of unique row is one element of such automatic methodology. As an example, suppose that a data frame, *m*, contains the three

hierarchically related categorical variables, county, municipal, and school district. Then `unique(m[order(m[,1],m[,2],m[,3]),])` produces a table that can be used as a basis for the level-hierarchy coding. In a general application we have several dimensions and several hierarchies. Another point is that, within the actual project, municipals are grouped in two ways (county is one of them). When doing table suppression this means that two so-called linked tables are needed. This makes the input needed even more complicated. When working with this, new ideas came up and it turned out that, for our needs, generic automation was possible.

A single function for the methods library could then be made. Specialised functions for each problem were not needed. Nevertheless, the interface between R and the other system was challenging. The multidimensional nature of the table suppression problems makes it more complicated than the other functions that were integrated. The decision about how to organise input and output changed a few times. A request was that input and output data should be organised in a wide (unstacked) way so that a standard of having municipals as rows could be met. The possibility of having input and output organised this way was also programmed and additional problems were automated.

All in all, the resulting function named *ProtectTable* contains additional functionality on the top of sdcTable. It can be view as a limited wrapper of the function *protecTable* in sdcTable or an easy interface to some of the functionality in sdcTable. Using variable numbers (instead of names) an example of a call to ProtectTable is: `ProtectTable(data, dimVar=1:4, freqVar=5, maxN=3, method="OPT")`. The fifth column contains the frequencies. The four preceding columns will be interpreted by the automatic methodology to find hierarchical relationships. The parameter *maxN* defines the suppression rule and *method* is the algorithm to be used.

It is clear that the new functionality can be useful for others outside Statistics Norway. The programmed code was therefore released as a package named *easySdcTable* (Langsrud, 2017). In reality, two packages were released. The basic functions of the new methods were included in the package *SSBtools* which is compatible with Renjin (see above). Only the actual wrapping to sdcTable was included in easySdcTable. This way, the functions for the easy user interface and more specific functions are separated. In particular, SSBtools contains the function *FindDimLists* which generates the level-hierarchies that can be derived from a data frame with the relevant variables. The package SSBtools also contain functions not needed by easySdcTable. A policy is to publish several functions in SSBtools for possible re-use by others. Note that

the package easySdcTable has been extended with a graphical user interface based on *shiny*. This will not be included in the IT system embedding the methods library. It is meant for other types of usage as described below.

Quite recently, the possibility of calling routines in the non-R program, tau-*Argus*, has been included in sdcTable. In the future several methods (such as *"OPT"* above) may also be removed from sdcTable. The function ProtectTable needs to be changed accordingly. This means the valid strings for the parameter *method* will change. In fact, the possibility of calling tau-Argus has already been implemented in easySdcTable. In particular, the method *"TauArgusOPT"* directs ProtectTable to use sdcTable to do optimal cell suppression by calling tau-Argus.

5. SPIN-OFF USAGE

R functions programmed for the methods library are meant to be used within the general IT system. However, useful functions may be useful in several ways.

The code may, of course, be used directly by R users. Therefore, when developing an R package, it can be a good idea to make not only the methods library functions visible (export). In the case of regression imputation described above, the function LmImpute has been re-used in other projects.

Often SAS users would like the functionality to be available within SAS. It is possible to call R form SAS within *PROC IML*. This can sometimes be a practical solution.

Another possibility is to make a separate graphical user interface. In the case of table suppression, a graphical user interface has been developed using the package *shiny*. The interface can be used to produce suppressed tables in practise. It can also be used to generate strings with R function calls that can be copied for later usage.

One possibility is to call R in batch mode. The user may not call R directly. Instead a script file (windows or linux) that makes call to R could be run.

6. CONCLUSIONS AND DISCUSSIONS

Using R as the programming language within the methods library has been successful. Requested methods have been implemented according to demanded standards. Within the KOSTRA project some methods for macro-editing, imputation and confidentiality have been implemented in the methods library. The project has involved new types of collaborations between statistical methodologists and IT experts. This has been encouraging, but also sometimes frustrating. The standards were very strict and it seems

that the complexity of the statistical methods that are needed in a statistical production process has been underestimated. Better knowledge when planning the system would have prevented some of these problems. Having IT oriented statistical methodologists and statistical methodology oriented IT experts (knowing R) will be very advantageous. So far the methods library is at an early stage and, according to the modernisation program, the methods library will play a key role in the future production system of Statistics Norway. A lot of collaboration between statistical methodologists and IT will be needed. A common ground of knowledge will result in better solutions. One step towards this is the development of a statistical information model within Statistics Norway based on the *Generic Statistical Information Model* (GSIM) described in UNECE (2017b). This will create a common vocabulary and support the communication between different types of subject areas and disciplines.

Within the KOSTRA project R has only been used as a programming language for implementing computational functions. In a modernised system R may also be used in other ways, for example to produce interactive graphics using shiny. Other countries are also planning similar systems as Norway. Common standards, sharing and reuse are important key words. We look forward to exchanging experiences and to discuss solutions.

References

1. **Foss, A. H., Langsrud, Ø. and Seierstad, A.**, 2017, "Methods Library as part of the modernization of the statistical production in Norway", Work Session on Statistical Data Editing, The Hague, Netherlands, 24-26 April 2017, https://statswiki.unece.org/download/attachments/125436234/Paper_hague.doc?version=1&modificationDate=1487336416148&api=v2
2. **Lang, M.**, 2017. "checkmate: Fast Argument Checks for Defensive R Programming." The R Journal, 9(1), pp. 437–445. https://journal.r-project.org/archive/2017/RJ-2017-028/index.html.
3. **Langsrud, Ø.**, 2017, "easySdcTable: Easy Interface to the Statistical Disclosure Control Package", R package on CRAN, https://cran.r-project.org/package=easySdcTable
4. **Meindl, B.**, 2017, "sdcTable: Methods for Statistical Disclosure Control in Tabular Data", R package on CRAN, https://cran.r-project.org/package=sdcTable
5. **SDMX Technical Working Group**, 2016. "VTL – version 1.1", https://sdmx.org/wp-content/uploads/VTL-1-1-review-User-Manual-20161017-final.pdf, https://github.com/statisticsnorway/java-vtl
6. **Thorud, A. B, Fløysvik, T, Abrahamsen, D., Tønseth, H., Berge, G., Foss, A. H. and Hagemo. J. O. J.**, 2011, "System for beregning av nasjonale tall i KOSTRA II" Notater2011/50, Statistics Norway (Norwegian only), https://www.ssb.no/offentlig-sektor/artikler-og-publikasjoner/_attachment/99930?_ts=13d2044d230
7. **UNECE**, 2017a, "Common Statistical Production Architecture", https://statswiki.unece.org/display/CSPA/Common+Statistical+Production+Architecture
8. **UNECE**, 2017b, "GSIM Version 1.1", https://statswiki.unece.org/display/gsim/Generic+Statistical+Information+Model