R on DST

Preamble

This document is made with the purpose of aiding use of R on servers in Statistics Denmark. We will not exclude that it can be helpful elsewhere. The particular characteristic of this environment is the very large size of datasets. For this reason data management is in general suggested to use functions from the package "data.table" which is specifically designed to handle very large datasets. Most introductions to R refer to "data.frame" and "data.table" is a data.frame with some add-ons. There are plenty of pdf and introductions to data.table on the web. This document does not replace such reading.

The structure of this document attempts to follow the typical path of a project: Include data, manage data and finally perform calculations and produce graphics.

Selected chores make use of functions in the package "heaven" which is made specifically for use in Statistics Denmark. This package is available on github and can be installed with

```
devtools::install_github("tagteam/heaven").
```

Statistics Denmark is a closed environment and all packages available are pre-installed, therefore only library statements are necessary. All packages on CRAN installed on the servers used by us.

In general there are multiple ways of reaching the target of a project. The suggestions in this document are not rules to be followed, but reflect experience after many years of working in the particular environment. Suggestions for improvement are always welcome.

Most R programs start with a list of **library()** statements to provide connection with functions in selected packages. Another possibility is to provide both the package name and the function name for a call. In this document we will try consitently to use the latter convention to show which packages needs to be included in a program. For standard use library statements should be the rule. The function thisFunction() in thisPackage is in this document shown as **thisPackage::thisFunction()**

Housekeeping rules

It is important to structure the data of a project in such a way that a logic is maintained - and also a logic which can be identified by others. In the common circumstance that help is required by others this logic is tested. It should also be noted that a scientific project can later be criticized and require a need to document how a result was obtained. In this situation a logical structure of the project is necessary when the programs need to be rerun years later.

For a beginner it is tempting to produce multiple programs as a project is developed mixing data management with analysis which can easily result in a situation where the project contains a large block of a mixture of outdated and current programs - where the content of each program needs to be rerun in a specific order to obtain the results of the project. Therefore, users are strongly encouraged to follow certain rules.

- Create a directory for each logical part of a project - in general a directory for each paper. We recommend that this directory is created using **heaven::createProject("path to project")**
- Separate data management from calculations. Often it is useful to have two files for a project, one for data management and the other containing those statement that produce the tables and graphics for a paper.

- Keep the number of program very low and format them to follow the same logic as the paper you are writing. This should not prevent you from having multiple versions that are numbered such that the final version is easily identified. An alternative is to use git within Statistics Denmark to keep track of versions.
- Comment VERY generously. You will be surprised how short memory is regarding variable names etc.
- During development of a program many little tests are performed. Remove these statements when testing has been done. If you want to keep testing versions of a program use a separate directory such as the "sandbox".
- When you make major changes, then keep the old version either numbered or in a selected directory. If you results suddenly change this can be the only way of determining whether the changes reflect correcting a problem or creating a problem.
- Keep only moderate sized analysis datasets as files. Huge datasets can be created and clog the resources of our environment. Old datasets can be recreated by rerunning programs
- Keep the final analysis dataset until the paper is published. Data on Statistics Denmark are periodically updated which can result in changes.
- All datasets should be deleted when the paper has been published, but you need to make sure that the programs can be found even years later.
- It is wise to request export of programs so you can make sure these are not changed or deleted by mistake. To enable this without risk it is critical that programs never include statements that involve "microdata". An example of such a problem can be a logical statement that involves a person identifier variable.
- We encourage that the total output to produce a paper is programmed. Most will benefit from "R markdown" and geeks may choose EMACS-ORG. The advantage comes as a project develops. It is very common that the tables and figures for a project needs to be generated many times pending on various decisions regarding data. Having a program that generates everything is in such cases a great advantage. These output programs should not include very time consuming calculations. Therefore it is often advantageous to have one program generate the data as R objects and then have another R markdown program produce the output.
- Keep programming compact! It is common in programs that similar chores are repeated for slightly changing conditions. It is tempting for the beginner to copy-paste programming sections several times and make slight changes in each block. This makes programs difficult to read and difficult to maintain. Whenever such chores are needed the repeated blocks should be replaced by functions and/or loops.
- Make programs where not everything has to be rerun after changes. For many projects a single final dataset is used for many calculations. This can be generated by consecutive addition of more and more variables from other datasets. In this case if just one of the steps require a change everything has to be rerun. It is wiser to create the components independently and then have a final large merge step in the end. In this case only the component needing change and the final merge needs to be rerun.

Memory use

R keeps all data in RAM (random access memory) and the program is prone to "memory leakage" which implies that large chunks of memory are blocked from all users, but do not contain usable data. It is common to include very large datasets during data management and once these are not needed anymore they should be removed with **rm("names of data separated with comma")**

It is important to realize that this command does not free the RAM you have used, so when chunks of data have been "removed" you can clear the memory use with the garbage collector: **gc()**

The Rstudio program can under the pane "Environment" show the use of RAM memory. The garbage collector also reports on memory use and the operating system can provide lists of users along with their memory use. This is provided with the **Task Manager**. This feature is the one to use if you cannot undertand why there is not enough room for you or the server appears slow.

Hard drive memory should also be used sparingly, although less so than RAM. What should not be done is to keep long lists of files that represent the history of you data development. Hard drive memory is interrogated with the windows explorer.

Starting a project

If you have used the **create_project()** function to define the project there are suggestions for R scripts that encourage a starting comment defining the author and the purpose of the program.

At start you should use the function **setwd("path to data")** to the directory for your data - or to the directory for your project. This makes reference to datasets in your project easier. If you choose the directory of the project you can use relative paths to read and write in subfolders of your project such as **writeRDS(R-object-name,file="/data/filename.rds")** to write in the "data" subfolder.

Read data

Data in Statistics Denmark is generally provided as SAS datasets - and a few data are provided in ASCII format which includes the very useful "csv" format.

If you need to read an entire table from SAS you can use the **haven::read_sas()** function. The disadvantage of this is that it can take a long time to read a large dataset and the use of memory will be extensive. On the other hand this function can handle nearly any complicated SAS dataset which is not always the case with the import_SAS function described below.

To include data from SAS with limited time and resources we have developed the **heaven::import_SAS()** function that can accept a range of SAS commands to read only a part of the data. The saving of time can be dramatic. The help page for this function provides the many available options - a few are described here:

- obs=n - Read only the first n records. This is very useful during start to ensure that reading is correct while not waiting for the entire dataset to be read.
- keep=c("var1","var2",...) - limits the variables that are read
- date.vars=c("var1","var2",...) - converts SAS date vars to proper R dates. Note that in principle a SAS date is an integer representing the number of days since 01-JAN-1960 and in R it is the number of days since 01-JAN-1970.
- character.vars=c(....) - forces variables to be character
- where="SAS logical statement" - Reads only the records corresponding to the logic. Note that the statement needs to be correct SAS.
- filter=object - if object is a data.table with a single column containing person identifier, then only records from thos people are kept. This is useful if you at the start of a project can define the population and then ensure that you only read records for thos persons from other datasets. *set.hook="SAS statements" - this allows inclusion of such SAS statements that can appear in parenthesis after names of datasets in a SAS data statement. Keep statements should not be included here as they result in errors because the function attempts to define formats for all data except those in the dedicated keep statement.

Since the time to read large SAS datasets very often requires patience, it is usually wise to limit the number of times a SAS dataset is read. Often it is wiser to grab all the data you need in one pass and afterwards use R to select further data for various purposes. typically you may need medications and diagnoses to define very separate structures in the project - but try to collect all in one pass anyhow.

When you need to import ASCII data the most efficient function is **data.table::fread()**. This function can usually identify the type of data from the extension to the data file. The help page provides a long list of options for reading.

For most of the programming suggested here data need to be **data.table** rather than **data.frame**. When a data.table function does not work, it is likely that the format was **data.frame** and the most convenient way to convert is the function **data.table::setDT(object_name)**. This function can also be used just to make sure an object is **data.table**. For the rare situation that you need to convert the other way to data.frame the function **data.table::setDF()** does the job.

If an object has been saved as ".rdata" the way to read it is **load("path to data")**. The name of the object is also read and will appear in the environment.

Many data are saved as "rds" files, single R objects. They are read with **mydata <- readRDS("path to file")**.

Write data

The most efficient way to store datasets is as single object using **writeRDS(name_of_object,file="PathToFile.rds")**. If your working directory is the project directory and you want the object saved in the subdirectory "data" then use **writeRDS(name_of_object,file="/data/PathToFile.rds")**.

If you need to output text files (ASCII) then use **data.table::fwrite(name_of_object,file="pathToFile.csv")**. This is useful if You want to use excell and similar to produce tables. Most table are generated in programs as data.frame or data.table and after converting to a csv-object they can be read with excell.

Show data

As data are imported to the project it is necessary to check that what You have desired has actually been accomplished. You can click on object in the environment pane to show the datasets as a spread sheat. This is discouraged since large datasets take a long time to load. Very useful functions are:

- names(object) - display the names of a data.table (or data.frame)
- str(object) - provides a list of variables and the first values
- head(object) - provides the first 5 lines of the data
- object - if a data.table then the first 5 and last 5 are shown - and all if there are less han 100
- object[1:30] - shows the first 30 lines
- object[1:10,.(var1,var2,var2)] - the first 15 lines and only the selected variables

It is generally useful to create small tables to check that your data are what you expect them to be and a very useful procedure is

- **object[,.N,by="var1"]** which will tabulate the number of records by var1. Tabulation by multiple variables can be done by replacing by=var1 with character vector: **by=c("var1","var2"...)**.

Manipulating data

This is described using data.table. In this description **DT** represents your data.table object. The general format for manipulating is **DT[i,j,..]**. This is most easily learned by remembering that "i" are the records and "j" what should be done - and ".." further conditions. So the general rule is **DT[where,what,and_so_on]**

Data.table attemps to limit use of RAM by not making copies of data. If you write the statement **DT2 <- DT** you will not get a copy of the data but DT2 and DT will address the same data.table. If you really need a copy to manipulate the solution is **DT2 <- copy(DT)**. The avoidance of copying also makes it important to realize when you need to use "<-" to change the data. For the examples below to create a new variable no "<-" is present. Only the new vector with an additional variable is created without copying the rest of the data. If the whole dataset is changed by some command the "<-" is absolutely necessary.

Creating new variables

A simple new variable examplified by age is created with **DT[,age:=(date-birthdate)/365.25]**. This assumes that both date and birthdate are R dates.

A common target is to make a variable that is the largest or smallst non-missing of several other variables. This is accomplished with **DT[,max:=pmax(var1,var2,var3,na.rm=TRUE)]**. Beward of the "p" in "pmax" this p dictates that the comparison is made on a record level.

Several variables can be created in one step: **DT[,':='(var1=var3+var4,var5=var2*5)]**

When You create variables You need to make decisions regarding type:

- The age example above is a "date difference" and if You require it to ba a numeric you need to enclose the right hand side of the equation in the function **as.numeric()**
- Numerics are "real" numbers with high but limited precision which you can realise by showing that **round(0.5)** is zero rather than one. Therefore it can be wise to use **integer** for numbers that are integers.

- Integers are provided by putting "L" after the number, 25L is the integer 25. Similar to numeric, **as.integer()** converts to an integer.
- A logical statement produces a boolean, either TRUE or FALSE. It is a common convention to use the numerics 0 and 1 for FALSE and TRUE which can be achieved either by the function **as.numeric** or by multiplying the boolean variable with one.
- Variables that represent classes are **factors** in R. If we assume that sex has been coded as 0/1 you can generate a factor with relevant names with **DT[,sex:=factor(levels=c(0,1),labels=c("female","male"))]**. For practical programming it is common to change multiple variables to factors. A shortcut for this is the function **Publish::lazyFactorCoding(name_of_data)**. This function creates programming lines in the console similar to the sex example for all variables with a selected maximum number of levels. This block of lines can then be copied to the program and make it easy to generate multiple factors.
- It is tempting to change all class variables to factors, but be careful. Many regression functions require the outcome variable to be numeric.

Numeric to factor

In the example above with sex the numeric variable already represented levels. When the variable is a continuous numeric, the convinient way of creating a factor is the **cut()** function. A very simplistic example is **DT[,newvar:=cut(oldvar,3)]**. This creates three levels with equal range. A more useful example is creating of a variable which represents quartiles of the old variable: **DT[,quart:=cut(oldvar,breaks=c(quantile(oldvar,probs=seq(0,1,by = 0.25))),labels=c("Q1","Q2","Q3","Q4"))]**

Naming variables

the function **setNames(DT,..)** can be used to name all variables by providing for ".." a character vector with the new names: c("new1","new2"...) - or you can change selected variables by providing two character vectors.

Many imported datasets have a confusing attitude to "case" with mixtures of upper case and lower case names. Changing all variables to lower case can be achieved with **names(DT) <- tolower(names(DT))**

Selecting a subset of variables

The most efficient way to select a subset of variables is **DT2 <- DT[,.(var1,var5,var8)]**. In some circumstances, such as within a function another version is: **DT2 <- DT[,.SD,.SDcols=c(var1,var5,var8)]**. The variable **.SD** is "sub data.table" and refers to a group of columns in the dataset.

Subset selection

If a dataset has been sorted by individual identifier and another variable such as a date, it is common that You want to select jest the first occurrence for each individual. This is done with **DT2 <- DT[,.SD[1],by="individual_identifier"]**. If You instead want the last occurrence **[1]** is replace by **[.N]** and if You for some reason want the second occurrence the replacement is **[2]**.

If You want to select a subset based on a logical statement, and efficient way is **DT2 <- DT[age<100]**.

In case you want to search for all records that start with a selected string you can use: **DT2 <- DT[grepl("^A10",var)]** that finds all records where "var" starts with "A10". This is a very simple example of a "regular expression" which is an extremely versatile tool to search for complicated relation. An overview of regular expressions can be found here: https://stringr.tidyverse.org/articles/regular-expressions.html

It is common to require a number of conditions to be defined each by a range of variable content. An example is to search for multiple comorbidites which each is defined by a range of diagnosis codes. A special function **heaven::findCondition** have been developed for this purpose.

To use this function You first need to create a "named list" of (start of/end of) codes for each condition You wish to find. An example of such a list is **heaven::charlsonCodes**. You can also define another exclusion list. This can be useful if You e.g. want to find "all cancer except non melanoma skin cancer". You an then have "cancer='C' " in the names list of inclusions and in the list of exclusions you have an element "cancer" with the few condition you wish to exclude.

With the one or two named list generated the **heaven::findCondition** will find all occurrences of you selected condition. The examples on the help page also shows how you can manipulate the result further.

Removing variables

A single variable is removed with **DT[,var1:=NULL]** and a group of variables is removed with **DT[,c("var1","var3"):=NULL]**. Note that no "<-" is necessary, just the vector with the particular variable is removed.

Sorting data

The functions **setkey(DT,"var")** and **setkeyv(DT,c("var1","var2"))** sorts the data in ascending order and creates a key for fast use of the sorted data.

If you need sorting an variable order the functions to use are **setorder()** and **setorderv()** with this example **setorderv(DT, c("A", "B"), c(1, -1))** sorting ascending by A and descending by B.

Conditional change

The standard format for a conditioned change of a variable is **DT[var==value,var:=new_value]** Another option which occasionally is more convinient is to use **fifelse()** which needs three (or four) parameters: first a logical statment, then the value if TRUE then the value if FALSE and finally the optional value to return for NA: **DT[,newvar:=fifelse(oldvar>x,2,1,NA)]**

Change multiple variables

Many chores need to repeated on many variables and example being changing a numeric value of e.g. 999 to NA. This can be achieved by first creating a character vector of the variables you want changed and then creating a loop to make the changes:

```
vars <- c("var1","var2"...)
for (x in vars) DT[get(x)==999,(x):=NA]
```

Note two peculiarities here. If we had written x== instead of get(x)== data.table would not know whether x was an object or representing the loop variable. get() solves that problem by enforcing evaluation of x. Similarly, if we had written x:=NA we would repeatedly create a variable named x - the parenthesis forces the evaluation in this case.

Append - rbind

Combining several data.tables to create one long table can be achived with **rbindlist()** or **rbind()**. If the only input is a vector of data.table objects then names and order of columns need to match perfectly. If

one tables contains extra variables these columns can be set to NA for the other tables with the option **fill=TRUE**. The option **idcol="file"** adds a column with the name of the object that contributed to each record.

Merge

Combining data.tables by columns as apposed by rows can be achived with **cbind()**. This function simply the first row of each data.table with the first row of other data.tables.

if two data.tables have been provided the same key with **setkey** or **setkeyv** then **DT1[DT2]** will generate and **inner merge**, that is a data.table with those records where both data.tables have contributed. An **outer merge** is one where all records from both data.tables are present and NAs are placed where tables do not contribute with data. It is also common that You want one of the data.tables to define which individuals are in the final results and this is achieed by **left and right merge**. With the **merge** function these variants of merge are achived with **all=TRUE** for outer merge, **all.x=TRUE** for lesf merge and **all.y=TRUE** for right merge. Not adding any of these options results in an inner merge. As an example of a left merge: **newDT <- merge(DT1,DT2,by="idvar",all.x=TRUE)**

If a whole series of data.tables need to be merged, the simple merge function cannot do it, but it can be achieved with the **Reduce()** function. This function requires two parameters - a function and a list. Reduce will perform the function on the first two members of the list, then the result of this is put in the function along with the third member of the list etc. Thus, a left merge where the first member of a list of data.tables defines which individuals are in the final dataset can be made with: **DT <- Reduce(function(x,y){merge(x,y,all.x=TRUE,by="idvar")},list(DT1,DT2,DT3,...))**

Summary statistics

It is common that You need to create datasets that are summary statistics of other datasets. Data.table has a strong interface for this: **DT2 <- DT[,.(mean=mean(var1),nunmber=length(var1), max=max(var1)), by="grouping_variable"]**. This statement will calculate the mean and the maxfor each level of the grouping variable. In this statement the "." just before the parenthesis is short for "list".

Numbering

Numbering records by subgroup of another variable is done with **DT[,number:=1:.N, by=group_variable]**

Numbering groups, that is numbering all levels of one variable, is done with **DT[group_number:=1:.GRP, by=group_variable]**

Make multiple records

For some counting purposes it may be a good idea to create multiple records for levels of a selected variable:

```
library(data.table)
DT <- data.table(ID=c(1:3),number=c(1,2,3))
DT[]
```

```
##    ID number
## 1:  1      1
## 2:  2      2
## 3:  3      3
```

```
DT[,rownum:=1:.N] # variable indicating row
DT2 <- DT[,.(ID=ID,newvar=seq(1,number)),by="rownum"] # makes the extra records
DT2[] # Display the result
```

```
##    rownum ID newvar
## 1:      1  1      1
```

```
## 2:      2  2      1
## 3:      2  2      2
## 4:      3  3      1
## 5:      3  3      2
## 6:      3  3      3
```

Specialised functions for data manipulation

Matching

Simple matching can efficiently be performed with the **matchit::matchit** function. By default is finds "nearest neighbor" which also implies that the nearest neighbor may be far away. You therefore need to tabulate to check the effect of matching.

For survival type problems you will often need to find "risk sets". This implies that you are not only satisfied with finding individuals with e.g. same age and sex, but the controls you find need to be part of the risk set - they have not yet obtained the outcome of interest and they are not dead or have passed end of follow-up.

Risk-set matching comes in two flavors: incidense density matching and exposure density matching. The former is for a case-control type analysis where you match cases at the time of outcome with controls that have not (yet) reached the outcome. The function for this is **heaven::incidenseMatch**. The other flavor is a cohort type study where you wish to match individuals at the time of an exposure with controls that have not (yet) been exposed - and also have not reached the outcome, death of end of follow-up. The function for this is **heaven::exposureMatch**. ## Income The function **heaven::averageIncome** is helpful for defining average income over a range of years prior to a selected date

Education

Education codes on Statistics Denmark are 4-digit numbers. To convert these into accepted international classes you can merge with the data.table **heaven::edu_code** which has several groupings including ISCED codes.

Splitting - Time dependent analysis

In analyses of time dependent phenomena variables need to be updated to new values as time passes. This is practically handled by splitting. This implies that the record of an individual ends when a conditions changes and then a new record starts at the time of the new condition. Since many conditions can change and both age and calender time with certainty changes the splitting of an individual can result in truly many records for each individual. Three functions have been developed to carry out this task: The **heaven::lexisTwo** function can split each record once for each condition - and is typically used for comorbidities where the status of the comorbidity variable is zero prior to the date of the comorbidity and "1" after. The **heaven::lexisFromTo** function can handle intervals and is typically used for medication where each treatment period is an interval. Finally **heaven::lexisSeq** can split by vectors and is typically used for a vector of age levels or a vector defining calender time periods.

Prescriptions to treatment periods

Medication on Statistics Denmark is provided as prescriptions that are picked up at Danish pharmacies. Provided is date of prescription, number and strength of medication units (pills). The function **heaven::medicinMacro** can transform this list of prescriptions into a list of treatment periods providing start, end and intensity of treatment. To enable this the user has to provide information on the use of each strength of each medication and other data. This information is provided as named lists.

Charlson codes

The function **heaven::charlsonIndex** can provide Charlson Index and individual codes using diagnosis codes and a variable defining the date at which the index should be calculated.

Hypertension

The function **heaven::hypertensionMedication** can define the presence of hypertension using medication used for treating hypertension. The function can provide either the presence of hypertension at a specific date or the date at which hypertension appears.

Demographic tables

A key for any publication is presentation of the population, most often a range of characteristics subgrouped by exposure or outcome of interest. Such tables can be made in a single step with functions from the **table1** package or with the **Publish::univariateTable** function. The **Publish::summary(univariateTable)** function can be abbreviated with **Publish::sutable**

Survival analysis

Univariate and stratified analysis

Cumulative incidence and Kaplan meier analysis can be obtained with a range of function, with our favorite being **prodlim::prodlim**. For practical use of this (and other functions) it is important to understand some general rules of function creation: * The immediate product of this function and regression function is an "object" with is a list of all sorts of varied information that the programmer considers useful for use. It is for very special purpose that this object needs to be interrogated. * Just producing the object without assigning it to a new object will in general result in execution of the **summary.prodlim** function which tabulates a summary useful for many purposes. * Most such objects also have a plot function which in this case following a convention is names **plot.prodlim**. Each of these standard functions have their own options for adjusting the output. If you need to adjust the plot.prodlim function it is not helpful to interrogate the **help(prodlim)** page, but rather the **help(plot.prodlim)** page.

Regression

The **survival::coxph** function is the choise for Cox regression and the **glm** function is the choise for logistic regression and Poisson regression.

The simple summary function is not useful for publication ready output from these function, but the **Publish::regressionTable** function provides useful tables. A simplistic forest plot can be made with **Publish::plot(regressionTable)** and more flexible tabulation beside the plot with **Publish::plotConficense** function. Advanced tables and plots can be created within the rich environment of **ggplot2**.

It is common not only to present an overall analysis, but also a range of subgroups by selected variables. Tables and plots of such subgroups can be made with the **Publish::subgroupAnalysis** function.

Looping analyses

It is common to present the results of multiple outcomes with and without multiple subgroups in a single table. In such situations it is much faster and much more transparent to create the multiple results in a loop rather than as separate programming blocks. To understand an efficient way of producing such analyses we need to introduce two function: * **lapply** is a strong looping function that requires two arguments, a **list** and a **function**. The function will consecutively apply the function to each element of the list and produce a list object containing the results of the function evaluation. * The final list elements of the lapply function

are for practical use data.tables (or data.frames) - and these can be aggregated into a single data.table object with **data.table::rbindList**

Example:

```
library(data.table)
DT <- data.table(ID=c(1:3),number=c(1,2,3),number2=c(8,9,10))
rbindlist(lapply(DT[,number],function(x){ # the function being defined right here
  data.table(PTID=DT[ID==x,ID],output=DT[ID==x,number2]*x) # Illustrative nonsense calculation
  # The last statement of a function is what is output
}))
```

```
##    PTID output
## 1:    1      8
## 2:    2     18
## 3:    3     30
```