

R on DST

Tips for efficient use of R on Statistics Denmark

Christian Torp-Pedersen

Kathrine Kold Sørensen
Filip Gnesin

Amalie Lykkemark Møller

19 January 2024

Contents

New additions	2
Preamble	2
Housekeeping	3
RStudio startup	3
Memory use	3
Starting a project	3
Brief introduction to Targets	3
Read data	5
Save data	6
View your data	6
Manipulating data	7
Creating new variables	7
Numeric to factor	7
Naming variables	8
Selecting a subset of variables	8
Subset selection	8
Removing variables	8
Sorting data	9
Mark groups with or without sorting	9
Conditional manipulation of variable	9
Change multiple variables	9
Append - rbind	10
Merge	10
Summary statistics	11
Numbering	12
Make multiple records	12
Specialised functions for data manipulation	12
Matching	12
Income	12
Education	13

Splitting - Time dependent analysis	13
Prescriptions to treatment periods	13
Charlson codes	13
Hypertension	13
Demographic tables	13
Survival analysis	13
Univariate and stratified analysis	13
Regression	14
Looping analyses	14
Data visualization	14
The layers of ggplot	15
Example	15

New additions

January 2024: Description of the rleid function and how to get RStudio start up a bit less slow

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. One good place to get an overview of R is [<http://r.sund.ku.dk/>], which is focused on data.frame for management. Another resource is RStudios’s cheatsheets [<https://www.rstudio.com/resources/cheatsheets/>], which covers the majority of the widely used packages. We especially recommend the sheets for data.table, ggplot2 (for visualization) and lubridate (for handling dates).

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 in the CRAN package repository are installed on the servers used by us.

Occasionally you will find that a package does not work with the latest installment of R. In such cases you can from RStudio global options select an earlier version of R.

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 consistently 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()**. The exception to that rule is the package **data.table** where many functions cannot be shown indirectly.

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.

Housekeeping

In general there are multiple ways of reaching the target of a project. We encourage to use the suggestions in the document “Good Programming Practice”.

RStudio startup

A standard start up of RStudio in Statistics Denmark is very slow. One reason is the creation of the “Packages pane”. And this is slow because CRAN has been imported in total. To get a slightly quicker start you should disable this option in Global option - Packages - Management where the tick in “Enable package pane” should be removed.

It is also wise to remove many ticks of check boxes in the Tools - Global option - General menu. The work-space should never be saved and time is spent reading old things.

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(file1,file2,file3)** (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 understand 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 your data development. Hard drive memory is found 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.

In the beginning of your program, 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 **saveRDS(R-object-name,file=“/data/filename.rds”)** to write in the “data” subfolder.

Brief introduction to Targets

A script-based approach to data management and analysis has been a common practice, often involving a series of numbered scripts that need to be executed in a specific order to reproduce the desired results.

However, this approach can lead to challenges in managing dependencies, ensuring that downstream results reflect any changes in upstream data or code, and encouraging efficient coding practices. The Targets package offers a solution to these issues by streamlining function-driven pipelines, making it easier to handle dependencies and streamline complex data workflows.

One effective way to initiate the use of the Targets package is by following these steps:

1. First loading the package with **library(targets)**
2. Then setting the working directory using **setwd()**
3. Then executing the command **use_targets()** in the R console

This sequence of actions sets up the necessary directory structure in the specified working directory. The package creates an initial script named `__targets.R`, which serves as the foundation for building the data analysis pipeline.

Below is an example illustrating the application of the Targets package:

```
# Install required packages if not already installed
if (!requireNamespace("targets", quietly = TRUE)) {
  install.packages("targets")
}

# Load the necessary packages
library(targets)

# Define functions to be used within the pipeline
tar_option_set(
  packages = c("knitr", "ggplot2") # add packages
)

# Define the target functions
create_data <- function(){
  data <- data.frame(
    x = runif(100),
    y = rnorm(100)
  )
}

# Define the pipeline
invisible(list(
  tar_target(data, create_data()),
  tar_target(plot, {
    ggplot(data, aes(x = x, y = y)) + geom_point()
  })
))

#invisible to suppress output, not necessary normally
```

The provided simple example shows a very short pipeline, comprising two steps, which can be extended endlessly to accommodate complex analyses. The pipeline is executed by running **tar_make()** in the console. One of the main time saving aspects of Targets is that when running **tar_make**, any changes to the pipeline are detected and only objects that are outdated are updated.

Context specific decision in setting up the pipeline involves considering factors such as the extent of inclusion of results and sensitivity analyses in the pipeline, determining whether to integrate functions directly within the pipeline or store them in separate scripts, and establishing the appropriate scope for each individual function. Achieving the right balance in the complexity of the functions is crucial; functions should be lengthy

enough to leverage the benefits of a function-based setup, yet concise enough to facilitate effective debugging.

Learning Targets in R is most effective when approached through hands-on practice, but it can be beneficial to consider the following recommendations:

- Organize all functions in a dedicated folder titled “functions” within the working directory and use the command **sapply(list.files("~/functions"), source)** in the `__targets.R` script to source all the functions pre-pipeline.
- Create a separate script for each function and name it identically to the function it represents.
- Consider saving the results in a distinct output folder within the functions preparing the output, utilizing functions like **ggplot2::ggsave** or **writexl::write_xlsx**.
- Retain the ability to debug using the browser functionality by inserting the **browser()** statement within the relevant function and executing a **tar_make(callr_function=NULL)** command.
- Remove objects using **tar_delete()** as needed, and consider running a **tar_prune()** subsequently to remove any unnecessary objects.
- Utilize the functionality of **tar_load_everything()** or **tar_load(starts_with())** to load objects in bulk, avoiding the need to list each object explicitly.
- **tar_visnetwork()** can be used to visualize the dependencies and status of the objects in the pipeline.

The above being a very brief introduction to targets, we encourage further reading including:

- The {targets} R package user manual: <https://books.ropensci.org/targets/>
- Thomas Gerds is course director of the course “Targeted Register Analysis” which includes some nice exercises in R using targets, available at: <https://github.com/tagteam/registerTargets/tree/main>

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 the object is a data.table with a single column containing person identifier, then only records from those 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 those 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 (text files) 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 datasets are saved as “rds” files, single R objects. They are read and added to the environment with **MyData <- readRDS(“path to file”)**.

Save data

The most efficient way to store datasets is as single object using **saveRDS(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 **saveRDS(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 Excel 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 Excel. Note that the **fwrite** function senses the details of output format from the file name you provide.

View your data

As data is 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 sheet. 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 than 100
- **object[1:30]** - shows the first 30 lines
- **object[1:10,.(var1,var2,var2)]** - the first 10 lines and only the selected variables
- **View(object[1:10,.(var1,var2,var2)])** - can be used as a more conscious alternative to clicking on the object

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,keyby=var1]** which will tabulate the number of records by **var1**. Tabulation by multiple variables can be done by replacing **keyby=var1** with character vector: **keyby=c(“var1”,“var2”...)** or list **keyby=.(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,by]**. This is most easily learned by remembering that records or rows can be subset using “i”, columns can be selected, manipulated, or calculated using “j”, and finally grouped using “by”. So the general rule is **DT[where,what,grouping]**.

`Data.table` attempts 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

The creation of new variables will be made using ‘j’: **DT[,j]**. A simple new variable exemplified 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 smallest non-missing of several other variables. This is accomplished with **DT[,max:=pmax(var1,var2,var3,na.rm=TRUE)]**. Beware 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 the variable type. Commonly used variable types include **numeric, integer, factor, and character**.

- The age example above is a “date difference” and if you require it to be 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 the variable 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(boolean.var)** or by multiplying the boolean variable with one **DT[,numeric.var:=boolean.var*1]**.
- 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 convenient 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"))]`

- Importantly, converting a numeric variable containing only 0 and 1 to a factor will result in a factor with levels 1 and 2, where 0 was converted to 1 and 1 converted to 2.

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 "subset of data.table" and refers to a group of columns in the dataset.

Subset selection

If a dataset has been sorted by an individual identifier and another variable such as a date, it is common that you want to select just 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 replaced 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 at [<https://stringr.tidyverse.org/articles/regular-expressions.html>]. Further CHATGBT is very good are making regular expressions.

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 comorbidities which each are 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 can 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.

Mark groups with or without sorting

It is common that a variable is required which changes value when treatment, status or other individual characteristics change. Such grouping can be tricky since "by" in `data.table` also may sort the data - and unwanted results are returned if the characteristic studied may fluctuate between values such that one value appear several times. The `rleid` function is very useful in such cases.

```
library(data.table)
# Create a sample data.table
data <- data.table(
  person = c(1, 1, 1, 2, 2, 1, 1, 2, 2),
  time = c(1, 2, 3, 1, 2, 4, 5, 3, 4),
  value = c(10, 10, 10, 5, 10, 5, 30, 5, 10)
)
setkeyv(data, c("person", "time"))
data[,rl:=rleid(value),by="person"]
data[]
```

```
##      person time value rl
## 1:         1     1    10  1
## 2:         1     2    10  1
## 3:         1     3    10  1
## 4:         1     4     5  2
## 5:         1     5    30  3
## 6:         2     1     5  1
## 7:         2     2    10  2
## 8:         2     3     5  3
## 9:         2     4    10  4
```

Conditional manipulation of variable

The standard format for a conditioned change of a variable is `DT[var==value,var:=new_value]` Another option which occasionally is more convenient is to use `fifelse()` which needs three (or four) parameters: first a logical statement, 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 be repeated on many variables, an 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 achieved 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 table 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 opposed to by rows, can be achieved with **cbind()**. This function simply combines the first row of each data.table with the first row of other data.tables - and requires that they have the same size.

if two data.tables have been provided the same key with **setkey** or **setkeyv** then **DT1[DT2]** will generate an **right merge**, that is a data.table with those records where DT2 is intact and updated with information from DT1:

```
library(data.table)
DT1 <- data.table(a=1:5,b=11:15)
DT2 <- data.table(a=c(2,4,6),c=c(22,24,26))
setkey(DT1,"a")
setkey(DT2,"a")
DT1[DT2]
```

```
##    a  b  c
## 1: 2 12 22
## 2: 4 14 24
## 3: 6 NA 26
```

Alternatively, you can use **newDT <- DT1[DT2,on=.(idvar=idvar)]** if you are not using the **setkey** function. A left merge, one where DT1 is kept intact and updated with variables from DT2 could be made by switching the order.

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. In this case you need to use the **merge** function with **all=TRUE**:

```
merge(DT1,DT2,by="a",all=TRUE)
```

```
##    a  b  c
## 1: 1 11 NA
## 2: 2 12 22
## 3: 3 13 NA
## 4: 4 14 24
## 5: 5 15 NA
## 6: 6 NA 26
```

An **inner merge** is one where only records where both datasets contribute are in the result:

```
merge(DT1,DT2,by="a") # or you could have written all=FALSE
```

```
##    a  b  c
## 1: 2 12 22
## 2: 4 14 24
```

Using the **merge** function You can select **left** and **right** merge with **all.x=TRUE** and **all.y=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,...))**

In some cases one might be interested in merging two datasets based on an approximate match rather than exact match. This is useful when a match is defined by a time interval, for example when defining re-admissions after disease onset. For this, a **rolling join** can be used.

```
library(data.table)
disease<-data.table(disease.date=sample(seq(as.Date('2015/01/01'), as.Date('2020/01/01'), by="day"), 6),
  pnr=seq(1:6),
  sex=rep(1:2,3))

hospital.adm<-data.table(admission.date=sample(seq(as.Date('2015/01/01'), as.Date('2021/01/01'), by="day"), 30),
  pnr=sample(c(1:6),30, replace=TRUE))

#rolling join:
#prepare roll
disease[, rolldate := disease.date]
hospital.adm[, rolldate := admission.date]

setkey(disease,pnr,rolldate) #the variables which define the match
setkey(hospital.adm,pnr,rolldate) #the variables which define the match

# add the first hospital admission occurring after the disease
hospital.adm[disease,roll=-Inf]

##      admission.date pnr   rolldate disease.date sex
## 1:      2017-10-08   1 2017-06-15   2017-06-15   1
## 2:      2016-04-24   2 2015-06-22   2015-06-22   2
## 3:      2018-07-12   3 2018-04-13   2018-04-13   1
## 4:      2020-05-22   4 2019-07-08   2019-07-08   2
## 5:      2018-10-25   5 2018-08-23   2018-08-23   1
## 6:              <NA>   6 2019-10-09   2019-10-09   2

# add the first hospital admission occurring after the disease date but within 1 year of disease
hospital.adm[disease,roll=-364.25]

##      admission.date pnr   rolldate disease.date sex
## 1:      2017-10-08   1 2017-06-15   2017-06-15   1
## 2:      2016-04-24   2 2015-06-22   2015-06-22   2
## 3:      2018-07-12   3 2018-04-13   2018-04-13   1
## 4:      2020-05-22   4 2019-07-08   2019-07-08   2
## 5:      2018-10-25   5 2018-08-23   2018-08-23   1
## 6:              <NA>   6 2019-10-09   2019-10-09   2
```

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),number=length(var1), max=max(var1)), by="grouping_variable"]**. This statement will calculate the mean and the max for 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[] # Display the result

##      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 it finds the “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: incidence 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::incidenceMatch`. 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 or 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 condition changes and then a new record starts at the time of the new condition. Since many conditions can change, and both age and calendar 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 calendar time periods.

There are additionally three extra splitting programs **splitTwo**, **splitFromTo** and **splitSeq**. These are wrappers to the similar lexis programs and serve the purpose of providing splitting without specifying an “event” variable. They have been specifically included for use with LTMLE.

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 **prodlm::prodlm**. 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 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 named **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 choice for Cox regression and the **glm** function is the choice 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::plotConfidence** 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 functions:

- **lapply(x,y)** is a strong looping function that requires two arguments, a **list(x)** and a **function(y)**. lapply will consecutively apply the function to each element of the list and produce a list object containing the results of the function evaluation.
- **rbindlist** - 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
```

Data visualization

Visualizing your data in an intuitive and aesthetically pleasing manner is both useful in the process of getting to know your data and in the publication of your results. Always attempt that you conclusions of any study

is supported by graphics, and take the time necessary to make these graphics of high quality.

The package ggplot2 is a powerful tool for plotting basically anything you could think of and it allows you to modify all aspects of your plot which makes it much more flexible than the plot function in base R.

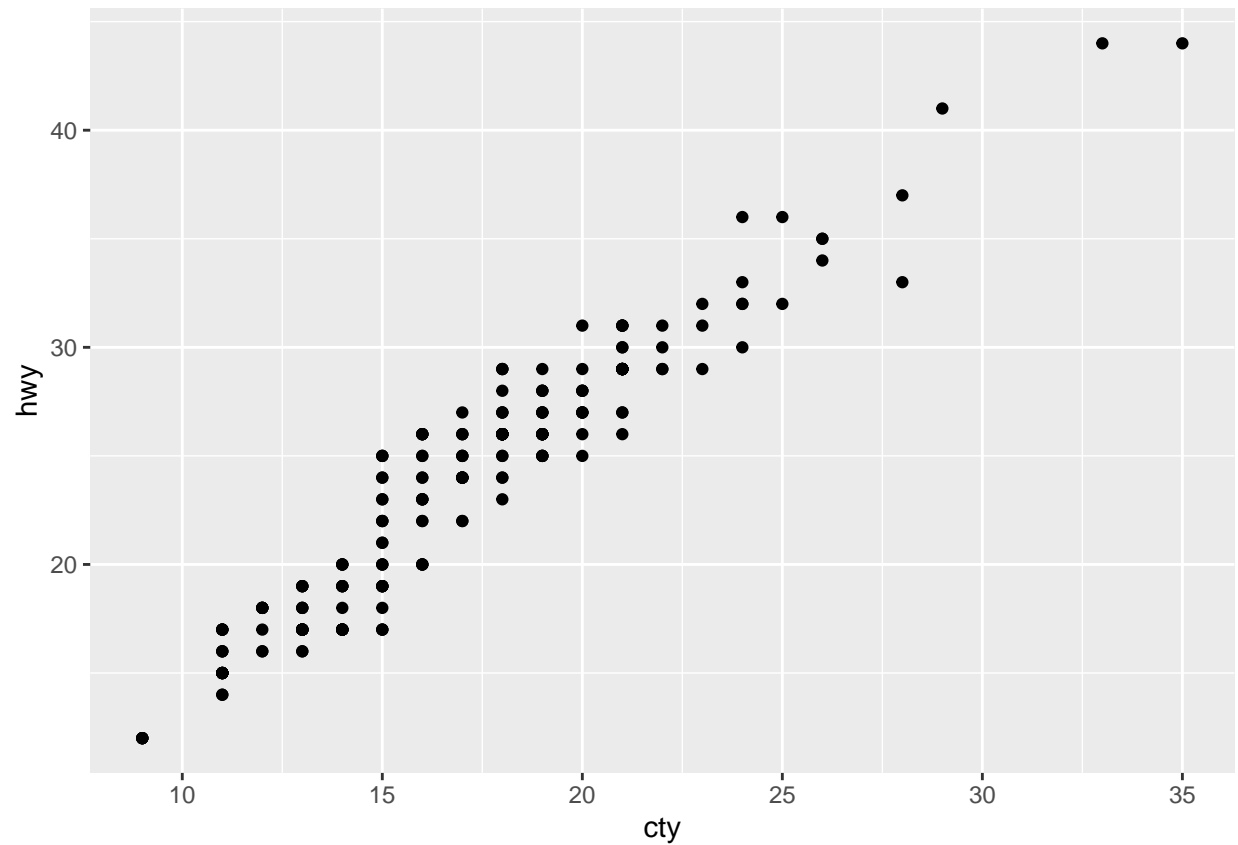
The layers of ggplot

A ggplot consists of different layers that can be combined to a graph. The different layers will add an extra layer of information to the plot. You can (in most cases) add several layers of the same type, e.g. two different geometries can be added and will create one plot with two types of graphs inside. In the following the different layers and their content is described briefly and a (simple) example is shown.

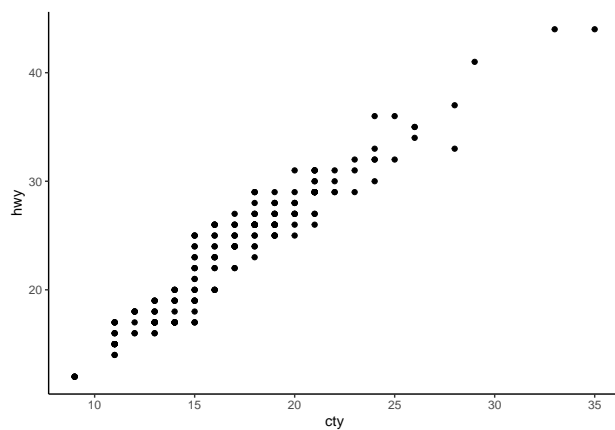
- **Data:** The data input for your plot can be in different formats, either as a full dataset, where each row is an observation or as summarized data, e.g. counts or percentages for different groups.
- **Mapping of data:** The mapping of data is defined using the function `aes()`. It is used to describe which part of your data input that is used for what e.g. which variable is your x and y axis.
- **Geometries:** The geometry is used to define how the data should be interpreted, e.g. as point, lines or bars.
- **Theme:** Themes can be used to modify anything that is unrelated to your data including labels, fonts, text size, background colors etc.

Example

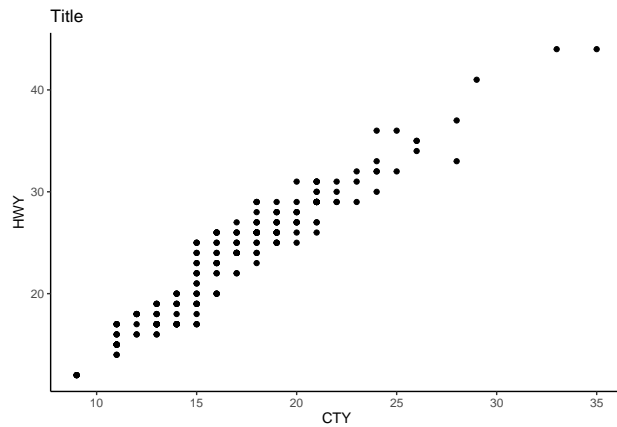
```
library(data.table)
library(ggplot2)
setDT(dt <- mpg) #toy dataset described at
#[https://ggplot2.tidyverse.org/reference/mpg.html]
ggplot(data=dt, mapping=aes(x=cty, y=hwy))+
  geom_point() #most simple scatterplot
```



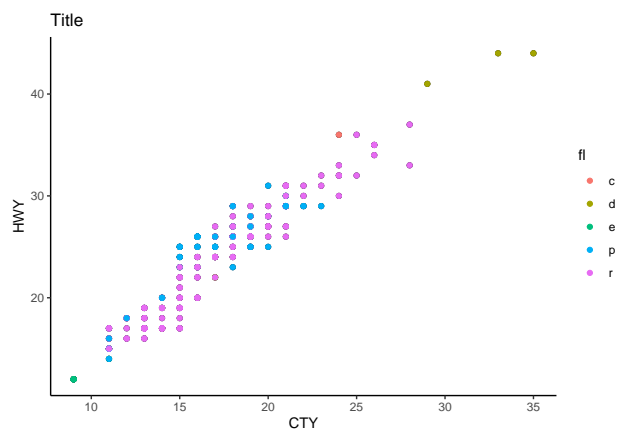
```
ggplot(data=dt,mapping=aes(x=cty,y=hwy))+
  geom_point()+ #most simple scatterplot
  theme_classic() #adding theme
```



```
ggplot(data=dt,mapping=aes(x=cty,y=hwy))+
  geom_point()+ #most simple scatterplot
  theme_classic()+ #adding theme
  labs(title="Title",
        x="CTY",
        y="HWY") #adding plot titles
```

```
ggplot(data=dt,mapping=aes(x=cty,y=hwy))+
  geom_point()+ #most simple scatterplot
  theme_classic()+ #adding theme
  labs(title="Title",
        x="CTY",
        y="HWY")+ #adding plot titles
  geom_point(mapping=aes(color=fl)) #adding grouping
```



#(could also be included in the previous geom_point call)

The possibilities are endless and we can suggest using the help page, [<https://www.r-graph-gallery.com/index.html>], as inspiration or our very own demonstration at [https://heart.dk/wp-content/uploads/2020/04/introduction_to_ggplot.html].