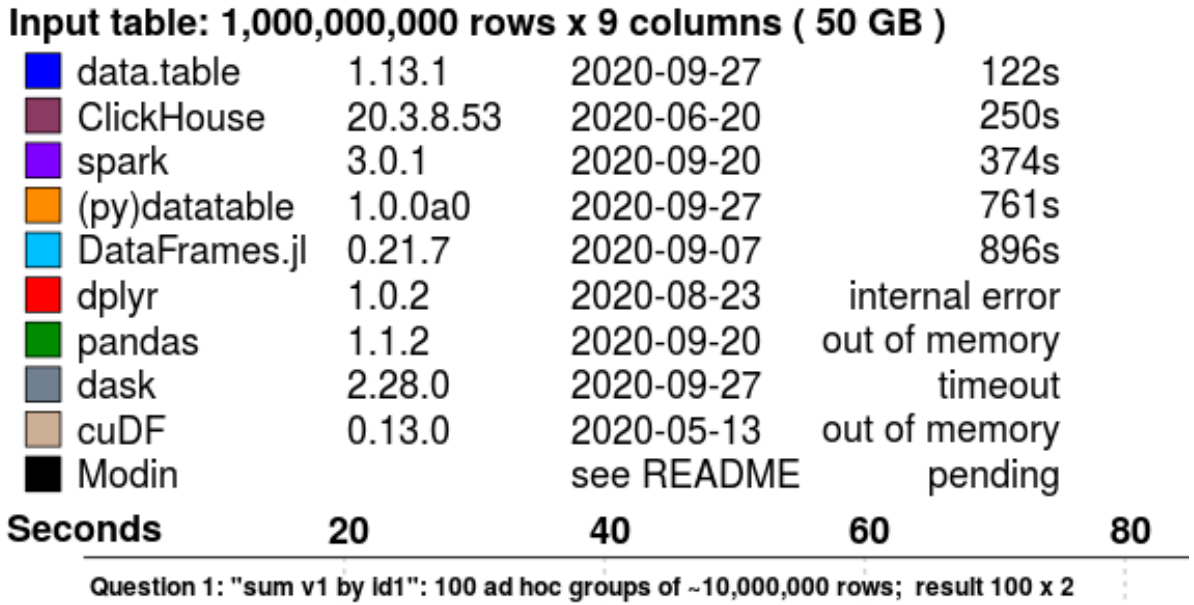
Why use data.table?

- A datatable, in context of the R package, is both a type of a data structure and a means to edit – similar to *dplyr*
- the syntax is an elegant extension of the classic *data.frame* often used for tabular data, so it can be used with any package that accepts a data frame structure
- The key aspect is its speed of execution particularly useful for complex models & exceptionally large datasets
  - It's FAST not just compared to other options in R but tools in other languages as well

How to install:

```
# Install from CRAN
install.packages('data.table')
```

https://h2oai.github.io/db-benchmark/



data.table is substantially faster across a number of aggregation tests using datasets of varying sizes.

How fast is it really?

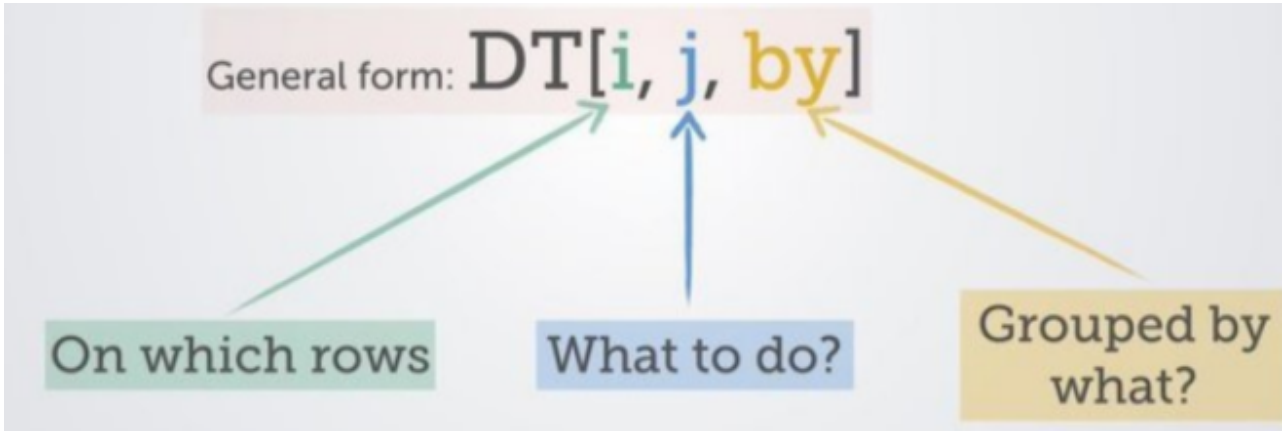
- With a dataset of 50GB, data.table takes 112 seconds to process the five tasks, compared with 2,539 seconds for *dplyr*.
- *pandas* is not able to complete the tasks at this data size due to insufficient memory in the test environment.
- On occasion a different tool beat data.table - but in those instances, data.table was second, with no other tool coming consistently in the top two for every task

```
 # Create a large .csv file
set.seed(100)
m <- data.frame(matrix(runif(10000000), nrow=1000000))
write.csv(m, 'm2.csv', row.names = F)

# Time taken by read.csv to import
system.time({m_df <- read.csv('m2.csv')})
#>    user  system elapsed
#> 39.798   1.326  43.003

# Time taken by fread to import
system.time({m_dt <- fread('m2.csv')})
#>  user  system elapsed
#> 1.735   0.097   1.877
```

How fast is it really?

fread () is 20 times faster in reading the data compared to the usual read.csv

user = time related to the execution of the code
system = time related to system processes such as opening and closing files
elapsed = is the difference in times since you started the stopwatch



1) DT: refers to the data table
2) i ⇔ 'where': specifies the rows that you want considered, it's a place to put the row conditions
3) j ⇔ 'select': column indexing that takes place - you might be making a new one or modifying an existing one, for example. i.e. put the conditions (to filter, to summarize) on columns here
4) by ⇔ 'group by': Doing stuff by group / categorical variable

The basic syntax (similar to SQL)

```r
#creating a dummy data table
DT <- data.table( ID = 1:50,
                  Capacity = sample(100:1000, size = 50, replace = F),
                  Code = sample(LETTERS[1:4], 50, replace = T),
                  State = rep(c("Alabama","Indiana","Texas","Nevada"), 50))

#simple data.table command
DT[Code == "C", mean(Capacity), State]
```

```
> head(DT)
   ID Capacity Code   State
1:  1      403    C Alabama
2:  2      419    C Indiana
3:  3      563    D   Texas
4:  4      552    D  Nevada
5:  5      694    D Alabama
6:  6      650    B Indiana
> #simple data.table command
> DT[Code == "C", mean(Capacity), State]
     State       V1
1: Alabama 511.3750
2: Indiana 600.6667
3:   Texas 511.3750
4:  Nevada 600.6667
>
```

- Code == "C" is used to filter the rows whose code is C
- mean(Capacity) calculates the mean capacity of the rows which have code C
- State indicates to do the above for every state separately.

Note: It's not necessary to always use all the three parts of the syntax

The basic syntax (similar to SQL)

A quick note

The data.table syntax is NOT RESTRICTED to only 3 parameters. There are other arguments that can be added to data.table syntax such as:

- with, which
- allow.cartesian
- roll, rollends
- .SD, .SDcols
- on, mult, nomatch

- You can also set keys & perform binary search algorithms
- Recode values using ‘if then else’ conditions

A few examples to test out…

```r
library(data.table)

mydata = fread("https://github.com/arunsrinivasan/satrdays-workshop/raw/master/flights_2014.csv")

# 1) selecting / keeping columns based on name - here keeping only the "origin" column
dat1 = mydata[ , .(origin)] # returns a data.table, note the preceding "." prior to (origin)

# 2) selecting columns based on position - here keeping 2nd column ="month"
dat2 = mydata[, 2]

# 3) keeping multiple columns
dat3 = mydata[, .(origin, year, month, hour)]

# 4) dropping multiple columns using the "!" sign
dat4 = mydata[, !c("origin", "year", "month"), with=FALSE]

# 5) using logical operator "%like%" to find pattern (here "dep") and keep columns
dat5 = mydata[,names(mydata) %like% "dep", with=FALSE]

# 6) subsetting rows/ filtering based on one variable
dat6 = mydata[origin == "JFK"]

# 7) subsetting rows/ filtering using '%in%' -- Filter all the flights whose origin is either 'JFK' or 'LGA'
dat7 = mydata[origin %in% c("JFK", "LGA")]

# 8) rename multiple variables with the setnames() function
setnames(mydata, c("dest","origin"), c("Destination", "origin.of.flight"))

#9) sort data using setorder() function, By default, it sorts data on ascending order
mydata01 = setorder(mydata, origin)
```

Data Transformation with data.table : : CHEAT SHEET

Basics

data.table is an extremely fast and memory efficient package for transforming data in R. It works by converting R's native data frame objects into data.tables with new and enhanced functionality. The basics of working with data.tables are:

dt[i, j, by]

Take data.table **dt**,
subset rows using **i**
and manipulate columns with **j**,
grouped according to **by**.

data.tables are also data frames – functions that work with data frames therefore also work with data.tables.

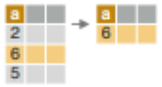
Create a data.table

data.table(a = c(1, 2), b = c("a", "b")) – create a data.table from scratch. Analogous to data.frame().

setDT(df)* or **as.data.table**(df) – convert a data frame or a list to a data.table.

Subset rows using i

dt[**1:2**,] – subset rows based on row numbers.



dt[**a > 5**,] – subset rows based on values in one or more columns.

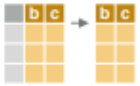
LOGICAL OPERATORS TO USE IN i

<	<=	is.na()	%in%	|	**%like%**
>	>=	!is.na()	!	&	**%between%**

Manipulate columns with j

EXTRACT

dt[, **c(2)**] – extract columns by number. Prefix column numbers with "-" to drop.



dt[, **.(b, c)**] – extract columns by name.

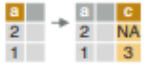
SUMMARIZE

dt[, **.(x = sum(a))**] – create a data.table with new columns based on the summarized values of rows.

Summary functions like mean(), median(), min(), max(), etc. can be used to summarize rows.

COMPUTE COLUMNS*

dt[, **c := 1 + 2**] – compute a column based on an expression.



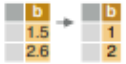
dt[**a == 1**, **c := 1 + 2**] – compute a column based on an expression but only for a subset of rows.

dt[, **`:=`(c = 1 , d = 2)**] – compute multiple columns based on separate expressions.

DELETE COLUMN

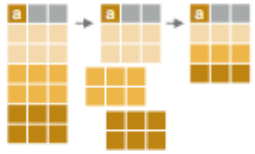
dt[, **c := NULL**] – delete a column.

CONVERT COLUMN TYPE



dt[, **b := as.integer(b)**] – convert the type of a column using as.integer(), as.numeric(), as.character(), as.Date(), etc..

Group according to by



dt[, j, **by = .(a)**] – group rows by values in specified columns.

dt[, j, **keyby = .(a)**] – group *and simultaneously sort* rows by values in specified columns.

COMMON GROUPED OPERATIONS

dt[, **.(c = sum(b))**, **by = a**] – summarize rows within groups.

dt[, **c := sum(b)**, **by = a**] – create a new column and compute rows within groups.

dt[, **.SD[1]**, **by = a**] – extract first row of groups.

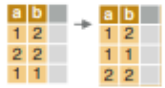
dt[, **.SD[.N]**, **by = a**] – extract last row of groups.

Chaining

dt[…][…] – perform a sequence of data.table operations by *chaining* multiple "[]".

Functions for data.tables

REORDER

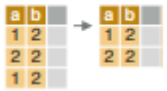


setorder(dt, a, **-b**) – reorder a data.table according to specified columns. Prefix column names with "-" for descending order.

*** SET FUNCTIONS AND :=**

data.table's functions prefixed with "set" and the operator ":=" work without "<-" to alter data without making copies in memory. E.g., the more efficient "setDT(df)" is analogous to "df <- as.data.table(df)".

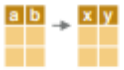
UNIQUE ROWS



unique(dt, by = c("a", "b")) – extract unique rows based on columns specified in "by". Leave out "by" to use all columns.

uniqueN(dt, by = c("a", "b")) – count the number of unique rows based on columns specified in "by".

RENAME COLUMNS



setnames(dt, c("a", "b"), c("x", "y")) – rename columns.

SET KEYS

setkey(dt, a, b) – set keys to enable fast repeated lookup in specified columns using "dt[.(value),]" or for merging without specifying merging columns using "dt_a[dt_b]".

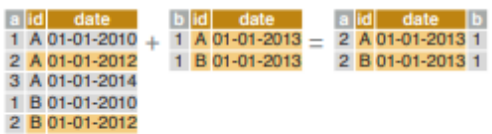
Combine data.tables

JOIN

dt_a[dt_b, **on = .(b = y)**] – join data.tables on rows with equal values.

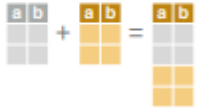
dt_a[dt_b, **on = .(b = y, c > z)**] – join data.tables on rows with equal *and unequal* values.

ROLLING JOIN



dt_a[dt_b, **on = .(id = id, date = date), roll = TRUE**] – join data.tables on matching rows in id columns but only keep the most recent preceding match with the left data.table according to date columns. "roll = -Inf" reverses direction.

BIND

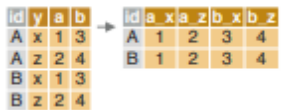


rbind(dt_a, dt_b) – combine rows of two data.tables.

cbind(dt_a, dt_b) – combine columns of two data.tables.

Reshape a data.table

RESHAPE TO WIDE FORMAT

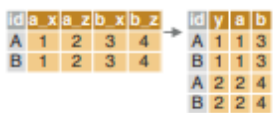


dcast(dt,
id ~ y,
value.var = c("a", "b"))

Reshape a data.table from long to wide format.

dt	A data.table.
id ~ y	Formula with a LHS: ID columns containing IDs for multiple entries. And a RHS: columns with values to spread in column headers.
value.var	Columns containing values to fill into cells.

RESHAPE TO LONG FORMAT



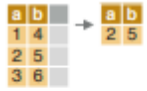
melt(dt,
id.vars = c("id"),
measure.vars = patterns("^a", "^b"),
variable.name = "y",
value.name = c("a", "b"))

Reshape a data.table from wide to long format.

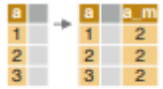
dt	A data.table.
id.vars	ID columns with IDs for multiple entries.
measure.vars	Columns containing values to fill into cells (often in pattern form).
variable.name, value.name	Names of new columns for variables and values derived from old headers.

Apply function to cols.

APPLY A FUNCTION TO MULTIPLE COLUMNS



dt[, **lapply(.SD, mean)**, **.SDcols = c("a", "b")**] – apply a function – e.g. mean(), as.character(), which.max() – to columns specified in .SDcols with lapply() and the .SD symbol. Also works with groups.



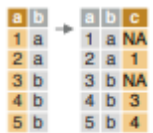
cols <- c("a")
dt[, **paste0(cols, "_m") := lapply(.SD, mean)**, **.SDcols = cols**] – apply a function to specified columns and assign the result with suffixed variable names to the original data.

Sequential rows

ROW IDS

dt[, **c := 1:.N**, **by = b**] – within groups, compute a column with sequential row IDs.

LAG & LEAD



dt[, **c := shift(a, 1)**, **by = b**] – within groups, duplicate a column with rows *lagged* by specified amount.

dt[, **c := shift(a, 1, type = "lead")**, **by = b**] – within groups, duplicate a column with rows *leading* by specified amount.

read & write files

IMPORT

fread("file.csv") – read data from a flat file such as .csv or .tsv into R.

fread("file.csv", select = c("a", "b")) – read specified columns from a flat file into R.

EXPORT

fwrite(dt, "file.csv") – write data to a flat file from R.

resources

- https://www.machinelearningplus.com/data-manipulation/datatable-in-r-complete-guide/
- https://rdatatable.gitlab.io/data.table/
- https://www.listendata.com/2016/10/r-data-table.html

Matlab package for R

Presentation by Lucas Jeay-Bizot

Class CS 510

Preliminary steps

- Please go to *https://github.com/lucasjeaybizot/PresentationCS510_MatlabPackage_R*
- You will find there a git repository with information on the matlab package for R

What is it ?

- This package was designed by P. Roebuck in June 2014
- This package simulates a set of useful MATLAB functions in R
- Package website: http://cran.r-project.org/package=matlab

How to install it ?

- Run the command *install.packages("matlab")* in Rstudio
  - Alternatively you can run the *install_package.R* script available in the github repository
- Before using it in a session, call *library("matlab")*

What does it do ? - 1

- It allows you to use a large set of matlab functions in R (full list available here: (https://cran.r-project.org/web/packages/matlab/matlab.pdf)
- For instance the eye() function in matlab can now be run in R

In R

```
> eye(9)
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
 [1,]    1    0    0    0    0    0    0    0    0
 [2,]    0    1    0    0    0    0    0    0    0
 [3,]    0    0    1    0    0    0    0    0    0
 [4,]    0    0    0    1    0    0    0    0    0
 [5,]    0    0    0    0    1    0    0    0    0
 [6,]    0    0    0    0    0    1    0    0    0
 [7,]    0    0    0    0    0    0    1    0    0
 [8,]    0    0    0    0    0    0    0    1    0
 [9,]    0    0    0    0    0    0    0    0    1
```

In MATLAB

```
>> eye(9)

ans =

     1     0     0     0     0     0     0     0     0
     0     1     0     0     0     0     0     0     0
     0     0     1     0     0     0     0     0     0
     0     0     0     1     0     0     0     0     0
     0     0     0     0     1     0     0     0     0
     0     0     0     0     0     1     0     0     0
     0     0     0     0     0     0     1     0     0
     0     0     0     0     0     0     0     1     0
     0     0     0     0     0     0     0     0     1
```

What does it do ? - 2

- Another example is the matlab function linspace()

In R

```
> linspace(1,11,7)
[1]  1.000000  2.666667  4.333333  6.000000  7.666667  9.333333 11.000000
```

In MATLAB

```
>> linspace(1,11,7)

ans =

    1.0000    2.6667    4.3333    6.0000    7.6667    9.3333   11.0000
```

- Or the function primes()

In R

```
> primes(33)
 [1]  2  3  5  7 11 13 17 19 23 29 31
```

In MATLAB

```
>> primes(33)

ans =

     2     3     5     7    11    13    17    19    23    29    31
```

How to translate MATLAB to R ?

- Some key differences:
  - Assigning variables
    - In MATLAB: “=“
    - In R: “<-”
  - Commenting code
    - In Matlab: “%”
    - In R: “#”
- Otherwise, you can just write identically each function included in the package in both languages

Where can I find out more about it ?

- You can visit the github repository:

  https://github.com/lucasjeaybizot/PresentationCS510_MatlabPackage_R

- Open *examples.R* to find a list of the packages' functions and accompanying examples

- Open *exercise.R* to practice using the repository a little

Exercise.R walkthrough solution

Convert the code below from MATLAB to R using the matlab package in R:

```
A = magic(6);
A = reshape(A,2,18);
B = numel(A);
C = factor(B);
D = isprime(C);
```

The line by line code in R:

```
library("matlab")

A <- magic(6)
A <- reshape(A,2,18)
B <- numel(A)
C <- factors(B)
D <- isprime(C)
```

Similar packages

- R.matlab: permits to read and write mat files from within R as well as call MATLAB
- Reticulate: R interface to python

Thank You!

How is “survival” package used?

By

Isaac Nwi-Mozu

Introduction

Survival analysis is a statistical method used to describe/predict the occurrences and timing of events.

The basic functions of survival analysis are the same in all fields, but the name usually depends on the field of study. E.g.

Engineering: reliability analysis

Sociology : event history analysis

Economics: duration analysis

Biology/medicine: survival analysis.

Area and Application of Survival Analysis

- In medicine : comparison of survival times of different treatments in some fatal diseases
- In engineering: time until an electrical component fails
- In Economics: time to the collapse of a new business / promotion times for employees

Why survival analysis differ from other statistical method?

1. Censored information

2. Response variable is always time.

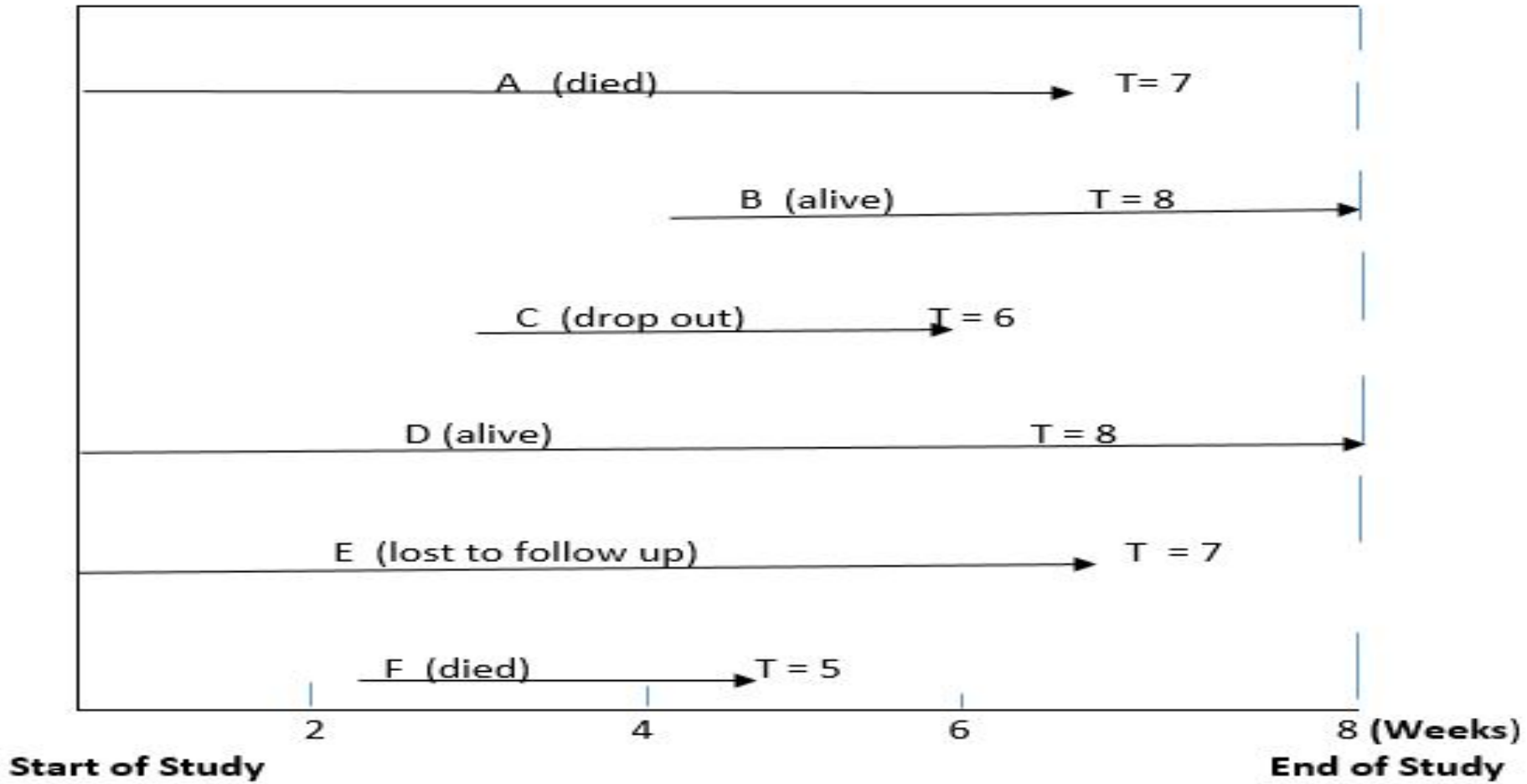
3. Staggered entries (in medical research). That is all individuals in the study do not have the same entrance time.

4. The assumption of normality is not hold in survival analysis, as survival data are generally skewed.

Reason for censoring

- A subject does not experience the event before the study ends i.e. limiting duration of study period.
- A person is lost to follow-up during the study period
- A person withdraws from the study
- Limiting duration of study period.

Survival study illustration



Package for survival analysis (model)

The package is **“survival”.**

The package is not an R basic package and so must be install for the first time only.

Installing the survival package for the first time.

```
install.packages(“survival”)
```

Loading the survival package to be used

```
library(survival)
```

```
> data.frame(foltime,event,age,teletherapy,telebrachy)
   foltime event age teletherapy telebrachy
1      189     0  45           1          0
2      148     1  66           1          0
3      100     0  52           1          0
4      221     0  67           1          0
5      237     0  74           1          0
6      365     0  78           1          0
7      138     0  49           1          0
8      365     0  53           1          0
9      150     0  57           1          0
10     120     0  47           1          0
```

Three approaches of survival analysis and the fitting is based on the type of approach used.

1. **Non -parametric method.** Usually graphical description.

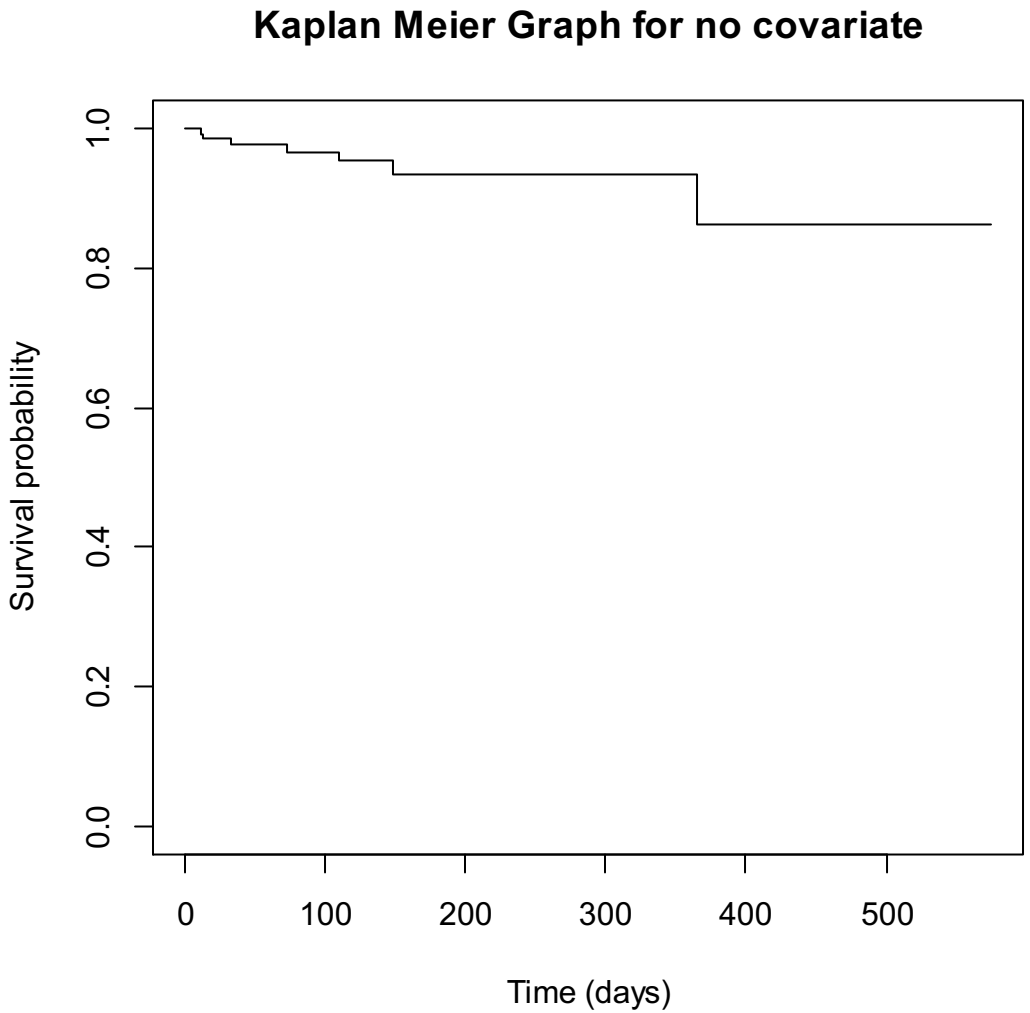
The syntax is:

survfit(Surv(time, event) ~1, data=dataname)

- Surv: transform the data into survival object
- Survfit: produce the non-parametric method (Kaplan Meier)

```
> kp.fit=survfit(Surv(foltime, event)~1,data=cancer)
> summary(kp.fit)
Call: survfit(formula = Surv(foltime, event) ~ 1, data = cancer)

 time n.risk n.event survival std.err lower 95% CI upper 95% CI
   11    132       1    0.992 0.00755        0.978        1.000
   13    131       1    0.985 0.01063        0.964        1.000
   33    117       1    0.976 0.01347        0.950        1.000
   73    100       1    0.967 0.01650        0.935        1.000
  109     75       1    0.954 0.02071        0.914        0.995
  148     51       1    0.935 0.02748        0.883        0.991
  365     26       2    0.863 0.05506        0.762        0.978
```

> plot(kp.fit,xlab="Time (days)",ylab="Survival probability", conf.int=F, main="Kaplan Meier Graph for no covariate")



2. Semi-parametric method (Cox Proportional Hazard model)

The syntax is:

coxph(Surv(time, event)~ x1 +x2, data= dataname)

- coxph: fits the CPHM
- x's : name(s) of your covariates (predictors)

cox.fit=coxph(Surv(foltime, event) ~ age +teletherapy+telebrachy, data=cancer,method="breslow")

summary(cox.fit)

```
Call:
coxph(formula = Surv(foltime, event) ~ age + teletherapy + telebrachy,
    data = cancer, method = "breslow")

  n= 136, number of events= 8

               coef exp(coef) se(coef)     z Pr(>|z|)
age         0.05962   1.06143  0.02720 2.192   0.0284 *
teletherapy 0.44668   1.56311  0.92302 0.484   0.6284
telebrachy  0.03533   1.03596  0.82639 0.043   0.9659
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

            exp(coef) exp(-coef) lower .95 upper .95
age             1.061     0.9421    1.0063     1.120
teletherapy     1.563     0.6398    0.2560     9.543
telebrachy      1.036     0.9653    0.2051     5.233

Concordance= 0.694  (se = 0.075 )
Likelihood ratio test= 5.33  on 3 df,   p=0.1
Wald test            = 4.85  on 3 df,   p=0.2
Score (logrank) test = 5.27  on 3 df,   p=0.2
```

3. Parametric model

Common survival parametric models are: exponential, Weibull, log-logistic, lognormal

The syntax is:

survreg(Surv(time,event)~x, dataname, dist="dist name")

- survreg: fitting survival parametric model
- dist: name of the parametric distribution used.

exp.fit=survreg(Surv(foltime,event) ~ age, data= cancer, dist="exp")

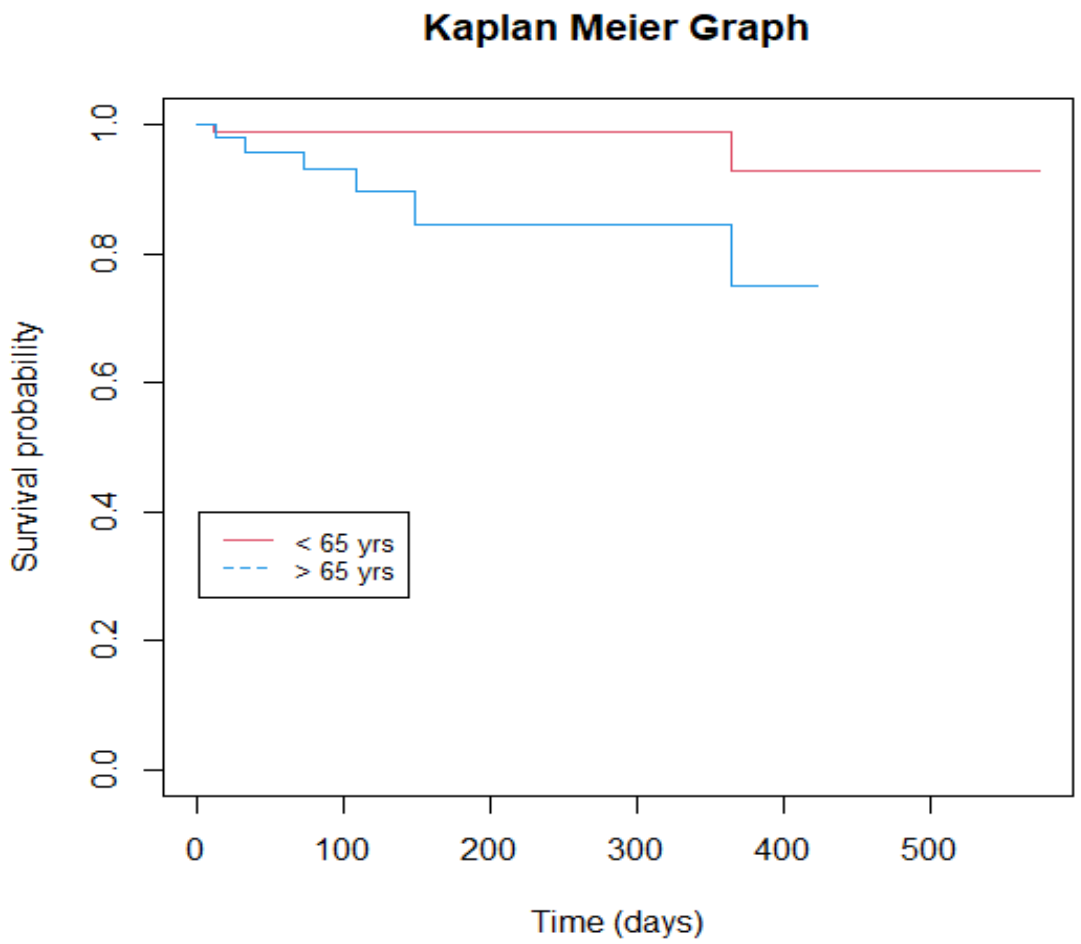
summary(exp.fit)

```
Call:
survreg(formula = Surv(foltime, event) ~ age, data = cancer,
    dist = "exp")
             Value Std. Error     z       p
(Intercept) 11.9092     1.9032  6.26 3.9e-10
age         -0.0612     0.0265 -2.31   0.021

Scale fixed at 1

Exponential distribution
Loglik(model)= -68.7   Loglik(intercept only)= -71.5
        Chisq= 5.71 on 1 degrees of freedom, p= 0.017
Number of Newton-Raphson Iterations: 7
n= 136
```

More details.

```
> age2=ifelse(age>65,1,0)
> age2
  [1] 0 1 0 1 1 1 0 0 0 0 0 0 1 0 0 0 1 0 1 1 0 1 0 1 0 0 0 0 0
 [30] 0 1 0 1 0 1 1 0 0 0 0 1 1 1 1 1 1 0 0 0 0 1 0 1 0 0 1 0 0
 [59] 1 0 1 0 1 1 1 1 0 0 1 0 0 0 1 1 0 1 0 0 1 1 0 0 1 0 0 1 0
 [88] 0 0 1 0 1 0 0 0 1 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1 0 0
[117] 0 0 0 1 1 0 1 0 0 1 0 0 0 0 1 0 0 1 1 1
```

```
cancer.data=data.frame(cancer,age2)
attach(cancer.data)

kp=survfit(Surv(foltime,event)~ as.factor(age2), data =cancer)
plot(kp.trt,xlab="Time (days)",ylab="Survival probability",
col=c(2,4), conf.int=F,main="Kaplan Meier Graph")

legend(1, 0.4, legend=c("< 65 yrs", "> 65 yrs"), col=c(2,4), lty
=1:2, cex=0.8)
```



How do we compare two curves

Most common test is a **log-rank test**.

The syntax is:

survdiff(Surv(time, status)~x, data= dataname)

survdiff: test the survival difference of the group.

X : must be categorical covariate

Example:

```
> survdiff(Surv(time,event)~as.factor(age2), data= cancer)
Call:
survdiff(formula = Surv(time, event) ~ as.factor(age2), data = cancer)

                    N Observed Expected (O-E)^2/E (O-E)^2/V
as.factor(age2)=0 84        2      4.5      1.39      3.88
as.factor(age2)=1 52        6      3.5      1.78      3.88

 Chisq= 3.9  on 1 degrees of freedom, p= 0.05
```

Question?

Thank you

Modelr package

Garrett Stemmler

Package `modelr'

- Functions for modelling that help you seamlessly integrate modelling into a pipeline of data manipulation and visualization.
- Good package for wanting to interact with models and for partitioning and sampling data.

Useful functions

- add_predictions
- add_predictors
- add_residuals
- data_grid
- resample
- resample_bootstrap
- resample_partition
- resample_permutation

data_grid function

- data_grid: generate a data grid
- Helps to visualize a model, and it is very useful to be able to generate an evenly spaced grid of points from data.

```
data_grid(mtcars, wt = seq_range(wt, 10), cyl, vs)
#> # A tibble: 60 x 3
#>       wt   cyl    vs
#>    <dbl> <dbl> <dbl>
#>  1  1.51     4     0
#>  2  1.51     4     1
#>  3  1.51     6     0
#>  4  1.51     6     1
#>  5  1.51     8     0
#>  6  1.51     8     1
#>  7  1.95     4     0
#>  8  1.95     4     1
#>  9  1.95     6     0
#> 10  1.95     6     1
#> # … with 50 more rows
```

Add functions

- add_predictions: add predictions to a data frame
- add_predictors: add predictors to a formula
- add_residuals: add residuals to a data frame

```
mod <- lm(y ~ x, data = df)
df %>% add_predictions(mod)
#> # A tibble: 100 x 3
#>          x     y  pred
#>      <dbl> <dbl> <dbl>
#>  1 0.00740 3.90   3.08
#>  2 0.0201  2.86   3.15
#>  3 0.0280  2.93   3.19
#>  4 0.0281  3.16   3.19
#>  5 0.0312  3.19   3.21
#>  6 0.0342  3.72   3.23
#>  7 0.0514  0.984  3.32
#>  8 0.0586  5.98   3.36
#>  9 0.0637  2.96   3.39
#> 10 0.0652  3.54   3.40
#> # … with 90 more rows
df %>% add_residuals(mod)
#> # A tibble: 100 x 3
#>          x     y   resid
#>      <dbl> <dbl>   <dbl>
#>  1 0.00740 3.90   0.822
#>  2 0.0201  2.86  -0.290
#>  3 0.0280  2.93  -0.256
#>  4 0.0281  3.16  -0.0312
#>  5 0.0312  3.19  -0.0223
#>  6 0.0342  3.72   0.496
#>  7 0.0514  0.984 -2.34
#>  8 0.0586  5.98   2.62
#>  9 0.0637  2.96  -0.428
#> 10 0.0652  3.54   0.146
#> # … with 90 more rows
```

Resample Function

- The resample class stores a “reference” to the original dataset and a vector of row indices.
- resample_bootstrap: generates a bootstrap replicate
- resample_partition: generate an exclusive partitioning of a data frame
- resample_permutation: create a resampled permutation of a data frame

```
# a subsample of the first ten rows in the data frame
rs <- resample(mtcars, 1:10)
as.data.frame(rs)
#>                    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
#> Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
#> Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
#> Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
#> Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
#> Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
#> Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
#> Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
#> Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
#> Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
#> Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
as.integer(rs)
#>  [1]  1  2  3  4  5  6  7  8  9 10

# generate a 30% testing partition and a 70% training partition
ex <- resample_partition(mtcars, c(test = 0.3, train = 0.7))
lapply(ex, dim)
#> $test
#> [1]  9 11
#>
#> $train
#> [1] 23 11
```