DATA SCIENCE AND MACHINE LEARNING

Introduction to Data Tables

Dimitris Fouskakis

Professor in Applied Statistics,

Department of Mathematics,

School of Applied Mathematical & Physical Sciences,

National Technical University of Athens

Email: fouskakis@math.ntua.gr

Aim

- Beyond Data Frames.
- Simpler R Data manipulation operations such as subset, group, update, join etc.. Aim:
  - concise and consistent syntax irrespective of the set of operations you would like to perform to achieve your end goal.
  - performing analysis fluidly without the cognitive burden of having to map each operation to a particular function from a potentially huge set of functions available before performing the analysis.
  - automatically optimising operations internally, and very effectively, by knowing precisely the data required for each operation, leading to very fast and memory efficient code.
- Briefly, if you are interested in reducing programming and compute time tremendously, then this package is for you. The philosophy that data.table adheres to makes this possible.

Usefulness

- It works well with very large data files.
- Can behave just like a data frame.
- Offers fast subset, grouping, update, and joins.
- Makes it easy to turn an existing data frame into a data table.

Example 1

- We will use NYC-flights14 data. It contains On-Time flights data from the Bureau of Transportation Statistics for all the flights that departed from New York City airports in 2014. The data is available only for Jan-Oct'14.
- We can use data.table's fast-and-friendly file reader fread to load flights directly.
- Aside: fread accepts http and https URLs directly as well as operating system commands such as sed and awk output. See ?fread for examples.

Example 1

```
flights <- fread("flights14.csv")
flights
#        year month day dep_delay arr_delay carrier origin dest air_time distance hour
#     1: 2014     1   1        14        13      AA    JFK  LAX      359     2475    9
#     2: 2014     1   1        -3        13      AA    JFK  LAX      363     2475   11
#     3: 2014     1   1         2         9      AA    JFK  LAX      351     2475   19
#     4: 2014     1   1        -8       -26      AA    LGA  PBI      157     1035    7
#     5: 2014     1   1         2         1      AA    JFK  LAX      350     2475   13
#    ---
dim(flights)
# [1] 253316     11
```

###OR

```
flights<-fread("https://raw.githubusercontent.com/Rdatatable/data.table/master/vignettes/flights14.csv")")
dim(flights)
[1] 253316     11
```

Example 1

- data.table is an R package that provides an enhanced version of data.frames, which are the standard data structure for storing data in base R. In the previous slide, we created a data.table using fread(). We can also create one using the data.table() function. Here is an example:

Example 1

```
DT = data.table(
  ID = c("b","b","b","a","a","c"),
  a = 1:6,
  b = 7:12,
  c = 13:18
)
DT
#    ID a  b  c
# 1:  b 1  7 13
# 2:  b 2  8 14
# 3:  b 3  9 15
# 4:  a 4 10 16
# 5:  a 5 11 17
# 6:  c 6 12 18
class(DT$ID)
# [1] "character"
```

Example 1

- You can also convert existing objects to a data.table using setDT() (for data.frames and lists) and as.data.table() (for other structures); see ?setDT and ?as.data.table for more details.

Example 1

- Unlike data.frames, columns of character type are never converted to factors by default.
- Row numbers are printed with a : in order to visually separate the row number from the first column.
- When the number of rows to print exceeds the global option datatable.print.nrows (default = 100), it automatically prints only the top 5 and bottom 5 rows. If you've had a lot of experience with data.frames, you may have found yourself waiting around while larger tables print-and-page, sometimes seemingly endlessly. You can query the default number like so:

  ```
  getOption("datatable.print.nrows")
  ```

Example 1

- In contrast to a data.frame, you can do a lot more than just subsetting rows and selecting columns within the frame of a data.table, i.e., within [...]. To understand it we will have to first look at the general form of data.table syntax, as shown below:

```
DT[i, j, by]
##   R:                i                 j        by
## SQL:  where | order by   select | update  group by
```

- Users who have an SQL background might perhaps immediately relate to this syntax.
- The way to read it (out loud) is:
  - **Take DT, subset/reorder rows using i, then calculate j, grouped by by.**
- Let's begin by looking at i and j first - subsetting rows and operating on columns.

Basics

- Get all the flights with “JFK” as the origin airport in the month of June.

```
ans <- flights[origin == "JFK" & month == 6L]   #(6L denotes integer)
head(ans)
#    year month day dep_delay arr_delay carrier origin dest air_time distance hour
# 1: 2014     6   1        -9        -5      AA    JFK  LAX      324     2475    8
# 2: 2014     6   1       -10       -13      AA    JFK  LAX      329     2475   12
# 3: 2014     6   1        18        -1      AA    JFK  LAX      326     2475    7
# 4: 2014     6   1        -6       -16      AA    JFK  LAX      320     2475   10
# 5: 2014     6   1        -4       -45      AA    JFK  LAX      326     2475   18
# 6: 2014     6   1        -6       -23      AA    JFK  LAX      329     2475   14
```

Basics

- Within the frame of a data.table, columns can be referred to as if they are variables, much like in SQL. Therefore, we simply refer to dest and month as if they are variables. We do not need to add the prefix flights$ each time. Nevertheless, using flights$dest and flights$month would work just fine.
- The row indices that satisfy the condition origin == "JFK" & month == 6L are computed, and since there is nothing else left to do, all columns from flights at rows corresponding to those row indices are simply returned as a data.table.
- A comma after the condition in i is not required. But flights[dest == "JFK" & month == 6L,] would work just fine. In data.frames, however, the comma is necessary.

Basics

- Get the first two rows from flights.

```
ans <- flights[1:2]
ans
#    year month day dep_delay arr_delay carrier origin dest air_time
distance hour
# 1: 2014     1   1        14        13      AA    JFK  LAX      359
2475    9
# 2: 2014     1   1        -3        13      AA    JFK  LAX      363
2475   11
```

- In this case, there is no condition. The row indices are already provided in i. We therefore return a data.table with all columns from flights at rows for those row indices.

Basics

- Sort flights first by column origin in ascending order, and then by dest in descending order:
- We can use the R function order() to accomplish this.

```
ans <- flights[order(origin, -dest)]
head(ans)
#    year month day dep_delay arr_delay carrier origin dest air_time distance hour
# 1: 2014     1   5         6        49      EV    EWR  XNA      195     1131    8
# 2: 2014     1   6         7        13      EV    EWR  XNA      190     1131    8
# 3: 2014     1   7        -6       -13      EV    EWR  XNA      179     1131    8
# 4: 2014     1   8        -7       -12      EV    EWR  XNA      184     1131    8
# 5: 2014     1   9        16         7      EV    EWR  XNA      181     1131    8
# 6: 2014     1  13        66        66      EV    EWR  XNA      188     1131    9
```

Basics

- Select arr_delay column, but return it as a vector.

```
ans <- flights[, arr_delay]
head(ans)
# [1]  13  13   9 -26   1   0
```

- Since columns can be referred to as if they are variables within the frame of data.tables, we directly refer to the variable we want to subset. Since we want all the rows, we simply skip i.
- It returns all the rows for the column arr_delay.

Basics

- Select arr_delay column, but return as a data.table instead.

```
ans <- flights[, list(arr_delay)]
head(ans)
#    arr_delay
# 1:        13
# 2:        13
# 3:         9
# 4:       -26
# 5:         1
# 6:         0
```

Basics

- Select both arr_delay and dep_delay columns.

```
ans <- flights[, .(arr_delay, dep_delay)]
head(ans)
#    arr_delay dep_delay
# 1:        13        14
# 2:        13        -3
# 3:         9         2
# 4:       -26        -8
# 5:         1         2
# 6:         0         4

## alternatively
# ans <- flights[, list(arr_delay, dep_delay)]
```

Basics

- Select both arr_delay and dep_delay columns and rename them to delay_arr and delay_dep.
- Since .() is just an alias for list(), we can name columns as we would while creating a list.

```
ans <- flights[, .(delay_arr = arr_delay, delay_dep = dep_delay)]
head(ans)
#    delay_arr delay_dep
# 1:        13        14
# 2:        13        -3
# 3:         9         2
# 4:       -26        -8
# 5:         1         2
# 6:         0         4
```

Basics

- How many trips have had total delay < 0?

```
ans <- flights[, sum( (arr_delay + dep_delay) < 0 )]
ans
# [1] 141814
```

Basics

- Calculate the average arrival and departure delay for all flights with “JFK” as the origin airport in the month of June.

```
ans <- flights[origin == "JFK" & month == 6L,
               .(m_arr = mean(arr_delay), m_dep =
                 mean(dep_delay))]
ans
#       m_arr    m_dep
# 1: 5.839349 9.807884
```

Basics

- We first subset in i to find matching row indices where origin airport equals "JFK", and month equals 6L. We do not subset the entire data.table corresponding to those rows yet.
- Now, we look at j and find that it uses only two columns. And what we have to do is to compute their mean. Therefore we subset just those columns corresponding to the matching rows, and compute their mean().
- Because the three main components of the query (i, j and by) are together inside [...], data.table can see all three and optimise the query altogether before evaluation, not each separately. We are able to therefore avoid the entire subset (i.e., subsetting the columns besides arr_delay and dep_delay), for both speed and memory efficiency.

Basics

- How many trips have been made in 2014 from “JFK” airport in the month of June?

```
ans <- flights[origin == "JFK" & month == 6L, length(dest)]
ans
# [1] 8422
```

- The function length() requires an input argument. We just needed to compute the number of rows in the subset. We could have used any other column as input argument to length() really. This approach is reminiscent of SELECT COUNT(dest) FROM flights WHERE origin = 'JFK' AND month = 6 in SQL.
- This type of operation occurs quite frequently, especially while grouping, to the point where data.table provides a special symbol .N for it.

Basics

- .N is a special built-in variable that holds the number of observations in the current group. It is particularly useful when combined with by as we'll see later. In the absence of group by operations, it simply returns the number of rows in the subset.
- So we can now accomplish the same task by using .N as follows:

```
ans <- flights[origin == "JFK" & month == 6L, .N]
ans
# [1] 8422
```

Basics

- Once again, we subset in i to get the row indices where origin airport equals “JFK”, and month equals 6.
- We see that j uses only .N and no other columns. Therefore the entire subset is not materialised. We simply return the number of rows in the subset (which is just the length of row indices).
- Note that we did not wrap .N with list() or .(). Therefore a vector is returned.
- We could have accomplished the same operation by doing nrow(flights[origin == "JFK" & month == 6L]). However, it would have to subset the entire data.table first corresponding to the row indices in i and then return the rows using nrow(), which is unnecessary and inefficient.

Basics

- Select both arr_delay and dep_delay columns the data.frame way.

```
ans <- flights[, c("arr_delay", "dep_delay")]
head(ans)
#    arr_delay dep_delay
# 1:        13        14
# 2:        13        -3
# 3:         9         2
# 4:       -26        -8
# 5:         1         2
# 6:         0         4
```

Basics

- Select columns named in a variable using the .. prefix.

```
select_cols = c("arr_delay", "dep_delay")
flights[ , ..select_cols]
#         arr_delay dep_delay
#      1:        13        14
#      2:        13        -3
#      3:         9         2
#      4:       -26        -8
#      5:         1         2
#     ---
```

- .. signals to data.table to look for the select_cols variable "up-one-level", i.e., in the global environment in this case.

Basics

- Select columns named in a variable using with = FALSE

```
flights[ , select_cols, with = FALSE]
#         arr_delay dep_delay
#      1:        13        14
#      2:        13        -3
#      3:         9         2
#      4:       -26        -8
#      5:         1         2
#     ---
```

- Setting with = FALSE disables the ability to refer to columns as if they are variables, thereby restoring the "data.frame mode".

Basics

- We can also deselect columns using - or !. For example:

```
# returns all columns except arr_delay and dep_delay
ans <- flights[, !c("arr_delay", "dep_delay")]
# or
ans <- flights[, -c("arr_delay", "dep_delay")]
```

Basics

- Select by specifying start and end column names.

```
# returns year, month and day
ans <- flights[, year:day]
# returns day, month and year
ans <- flights[, day:year]
# returns all columns except year, month and day
ans <- flights[, -(year:day)]
ans <- flights[, !(year:day)]
```

Aggregations

- How can we get the number of trips corresponding to each origin airport?

```
ans <- flights[, .(.N), by = .(origin)]
ans
#    origin     N
# 1:    JFK 81483
# 2:    LGA 84433
# 3:    EWR 87400

## or equivalently using a character vector in 'by'
# ans <- flights[, .(.N), by = "origin"]
```

Aggregations

- We know .N is a special variable that holds the number of rows in the current group. Grouping by origin obtains the number of rows, .N, for each group.
- By doing head(flights) you can see that the origin airports occur in the order "JFK", "LGA" and "EWR". The original order of grouping variables is preserved in the result. This is important to keep in mind!
- Since we did not provide a name for the column returned in j, it was named N automatically by recognising the special symbol .N.
- by also accepts a character vector of column names. This is particularly useful for coding programmatically, e.g., designing a function with the grouping columns as a (character vector) function argument.

Aggregations

- When there's only one column or expression to refer to in j and by, we can drop the .() notation. This is purely for convenience. We could instead do:

```
ans <- flights[, .N, by = origin]
ans
#    origin     N
# 1:    JFK 81483
# 2:    LGA 84433
# 3:    EWR 87400
```

Aggregations

- How can we calculate the number of trips for each origin airport for carrier code "AA"?

```
ans <- flights[carrier == "AA", .N, by = origin]
ans
#    origin     N
# 1:    JFK 11923
# 2:    LGA 11730
# 3:    EWR  2649
```

- We first obtain the row indices for the expression carrier == "AA" from i.
- Using those row indices, we obtain the number of rows while grouped by origin. Once again no columns are actually materialised here, because the j-expression does not require any columns to be actually subsetted and is therefore fast and memory efficient.

Aggregations

- How can we get the total number of trips for each origin, dest pair for carrier code "AA"?

```
ans <- flights[carrier == "AA", .N, by = .(origin, dest)]
head(ans)
#    origin dest    N
# 1:    JFK  LAX 3387
# 2:    LGA  PBI  245
# 3:    EWR  LAX   62
# 4:    JFK  MIA 1876
# 5:    JFK  SEA  298
# 6:    EWR  MIA  848
## or equivalently using a character vector in 'by'
# ans <- flights[carrier == "AA", .N, by = c("origin", "dest")]
```

- by accepts multiple columns. We just provide all the columns by which to group by. Note the use of .() again in by – again, this is just shorthand for list(), and list() can be used here as well. Again, we'll stick with .() here.

Aggregations

- How can we get the average arrival and departure delay for each orig, dest pair for each month for carrier code "AA"?

```
ans <- flights[carrier == "AA",
       .(mean(arr_delay), mean(dep_delay)),
       by = .(origin, dest, month)]
ans
#      origin dest month        V1        V2
#   1:    JFK  LAX     1  6.590361 14.2289157
#   2:    LGA  PBI     1 -7.758621  0.3103448
#   3:    EWR  LAX     1  1.366667  7.5000000
#   4:    JFK  MIA     1 15.720670 18.7430168
#   5:    JFK  SEA     1 14.357143 30.7500000
#  ---
```

Aggregations

- Since we did not provide column names for the expressions in j, they were automatically generated as V1 and V2.
- Once again, note that the input order of grouping columns is preserved in the result.

Aggregations

- data.table retaining the original order of groups is intentional and by design. There are cases when preserving the original order is essential. But at times we would like to automatically sort by the variables in our grouping.
- So how can we directly order by all the grouping variables?

```
ans <- flights[carrier == "AA",
       .(mean(arr_delay), mean(dep_delay)),
       keyby = .(origin, dest, month)]
ans
#      origin dest month       V1        V2
#   1:    EWR  DFW     1 6.427673 10.0125786
#   2:    EWR  DFW     2 10.536765 11.3455882
#   3:    EWR  DFW     3 12.865031  8.0797546
#   4:    EWR  DFW     4 17.792683 12.9207317
#   5:    EWR  DFW     5 18.487805 18.6829268
#  ---
```

Aggregations

- All we did was to change by to keyby. This automatically orders the result by the grouping variables in increasing order. In fact, due to the internal implementation of by first requiring a sort before recovering the original table's order, keyby is typically faster than by because it doesn't require this second step.

Chaining

- Let's reconsider the task of getting the total number of trips for each origin, dest pair for carrier "AA".

```
ans <- flights[carrier == "AA", .N, by = .(origin, dest)]
```

- How can we order ans using the columns origin in ascending order, and dest in descending order?
- We can store the intermediate result in a variable, and then use order(origin, -dest) on that variable. It seems fairly straightforward.

```
ans <- ans[order(origin, -dest)]
head(ans)
#    origin dest    N
# 1:    EWR  PHX  121
# 2:    EWR  MIA  848
# 3:    EWR  LAX   62
# 4:    EWR  DFW 1618
# 5:    JFK  STT  229
# 6:    JFK  SJU  690
```

Chaining

- But this requires having to assign the intermediate result and then overwriting that result. We can do one better and avoid this intermediate assignment to a temporary variable altogether by chaining expressions.

```
ans <- flights[carrier == "AA", .N, by = .(origin, dest)][order(origin, -dest)]
head(ans, 10)
#    origin dest    N
# 1:    EWR  PHX  121
# 2:    EWR  MIA  848
# 3:    EWR  LAX   62
# 4:    EWR  DFW 1618
# 5:    JFK  STT  229
# 6:    JFK  SJU  690
# ...
```

- We can tack expressions one after another, forming a chain of operations, i.e., DT[...][...][...].

Expressions in by

- We would like to find out how many flights started late but arrived early (or on time), started and arrived late etc…

```
ans <- flights[, .N, .(dep_delay>0, arr_delay>0)]
ans
#    dep_delay arr_delay      N
# 1:      TRUE      TRUE  72836
# 2:     FALSE      TRUE  34583
# 3:     FALSE     FALSE 119304
# 4:      TRUE     FALSE  26593
```

Expressions in by

- The last row corresponds to dep_delay > 0 = TRUE and arr_delay > 0 = FALSE. We can see that 26593 flights started late but arrived early (or on time).
- Note that we did not provide any names to by-expression. Therefore, names have been automatically assigned in the result. As with j, you can name these expressions as you would elements of any list, e.g. DT[, .N, .(dep_delayed = dep_delay>0, arr_delayed = arr_delay>0)].

Multiple columns in j - .SD

- It is of course not practical to have to type mean(myCol) for every column one by one. What if you had 100 columns to average mean()?
- How can we do this efficiently, concisely? Suppose we can refer to the data subset for each group as a variable while grouping, then we can loop through all the columns of that variable using the already- or soon-to-be-familiar base function lapply(). No new names to learn specific to data.table.
- Special symbol .SD: data.table provides a special symbol, called .SD. It stands for Subset of Data. It by itself is a data.table that holds the data for the current group defined using by.

Multiple columns in j - .SD

- Let's use the data.table DT from before to get a glimpse of what .SD looks like.

```
DT
#    ID a  b  c
# 1:  b 1  7 13
# 2:  b 2  8 14
# 3:  b 3  9 15
# 4:  a 4 10 16
# 5:  a 5 11 17
# 6:  c 6 12 18
```

Multiple columns in j - .SD

```
DT[, print(.SD), by = ID]
#    a b  c
# 1: 1 7 13
# 2: 2 8 14
# 3: 3 9 15
#    a  b  c
# 1: 4 10 16
# 2: 5 11 17
#    a  b  c
# 1: 6 12 18
# Empty data.table (0 rows) of 1 col: ID
```

Multiple columns in j - .SD

- .SD contains all the columns except the grouping columns by default.
- It is also generated by preserving the original order - data corresponding to ID = "b", then ID = "a", and then ID = "c".
- To compute on (multiple) columns, we can then simply use the base R function lapply().

```
DT[, lapply(.SD, mean), by = ID]
#    ID   a    b    c
# 1:  b 2.0  8.0 14.0
# 2:  a 4.5 10.5 16.5
# 3:  c 6.0 12.0 18.0
```

Multiple columns in j - .SD

- .SD holds the rows corresponding to columns a, b and c for that group. We compute the mean() on each of these columns using the already-familiar base function lapply().
- Each group returns a list of three elements containing the mean value which will become the columns of the resulting data.table.
- Since lapply() returns a list, so there is no need to wrap it with an additional .().

Multiple columns in j - .SD

- How can we specify just the columns we would like to compute the mean() on?
- Using the argument .SDcols. It accepts either column names or column indices. For example, .SDcols = c("arr_delay", "dep_delay") ensures that .SD contains only these two columns for each group.
- You can also provide the columns to remove instead of columns to keep using - or ! sign as well as select consecutive columns as colA:colB and deselect consecutive columns as !(colA:colB) or -(colA:colB).
- Now let us try to use .SD along with .SDcols to get the mean() of arr_delay and dep_delay columns grouped by origin, dest and month.

Multiple columns in j - .SD

```
flights[carrier == "AA",                    ## Only on trips with carrier "AA"
        lapply(.SD, mean),                  ## compute the mean
        by = .(origin, dest, month),          ## for every 'origin,dest,month'
        .SDcols = c("arr_delay", "dep_delay")] ## for just those specified in .SDcols
#      origin dest month  arr_delay  dep_delay
#   1:    JFK  LAX     1   6.590361 14.2289157
#   2:    LGA  PBI     1  -7.758621  0.3103448
#   3:    EWR  LAX     1   1.366667  7.5000000
#   4:    JFK  MIA     1  15.720670 18.7430168
#   5:    JFK  SEA     1  14.357143 30.7500000
#  ---
# 196:    LGA  MIA    10  -6.251799 -1.4208633
# 197:    JFK  MIA    10  -1.880184  6.6774194
# 198:    EWR  PHX    10  -3.032258 -4.2903226
# 199:    JFK  MCO    10 -10.048387 -1.6129032
# 200:    JFK  DCA    10  16.483871 15.5161290
```

Subset .SD for each group

- How can we return the first two rows for each month?

```
ans <- flights[, head(.SD, 2), by = month]
head(ans)
#    month year day dep_delay arr_delay carrier origin dest air_time distance hour
# 1:     1 2014   1        14        13      AA    JFK  LAX      359     2475    9
# 2:     1 2014   1        -3        13      AA    JFK  LAX      363     2475   11
# 3:     2 2014   1        -1         1      AA    JFK  LAX      358     2475    8
# 4:     2 2014   1        -5         3      AA    JFK  LAX      358     2475   11
# 5:     3 2014   1       -11        36      AA    JFK  LAX      375     2475    8
# 6:     3 2014   1        -3        14      AA    JFK  LAX      368     2475   11
```

Subset .SD for each group

- .SD is a data.table that holds all the rows for that group. We simply subset the first two rows as we have seen here already.
- For each group, head(.SD, 2) returns the first two rows as a data.table, which is also a list, so we do not have to wrap it with .().