Stats II - Lab 1

Data Wrangling

Martin Arvidsson

November 4, 2024

First, who am I?

- Post doc at IAS (PhD in *Analytical Sociology*, 2022).
- Before that—MSc in *Statistics & Machine Learning*.

Research profile

- Social networks + big data + computational tools

Outline for today

Today: basic intro to R; prepare you for coming labs in the course

- This may be repetition for **some/several** of you.
- Goal: ensure **everyone** knows enough `R` to do coming labs.

Part 1: “Practical lecture”

1. Recap of R *basics* – R objects, vectors, subsetting, etc.
2. Dealing with *data sets* in R – `data.frame`
3. *Plotting* in R – `ggplot2`
4. (Bonus) “big data” in R – `data.table`

Part 2

- Work on assignment

Recap of R basics

R objects

Although you can use the **console directly** for **simple computations**, e.g.

```
2*5
```

```
## [1] 10
```

… For more advanced/nested procedures, we rely on **R objects**:

- *R objects* allow you to **store data** (and other stuff, eg *functions*)
- *R objects* are stored in the **memory**
- *R objects* are **independent** of eachother;
    - i.e modifying `x` does not affect `y`

Object assignment

We create R objects via the assign-operator (`<-`)

```
x <- 1
y <- 2
```

We can easily *manipulate* and *use* these objects, e.g.

```
z <- x * y
print(z)
```

```
## [1] 2
```

If we want to *remove* an `R` object from our memory:

```
remove(z) ; exists("z")
```

```
## [1] FALSE
```

Vectors

Vectors are the **core units** (lowest lvl) of how data is stored in `R`

```
is.vector(x)
```

```
## [1] TRUE
```

- Crucially, *vectors* can store not just one value, but many.
- To create vectors with *multiple values*, we use the *combine function* `c()`:

```
x <- c(1,2,3)
print(x)
```

```
## [1] 1 2 3
```

Maths with vectors

Just as we can multiply **single** values, we can also multiply vectors storing **multiple** values, e.g.

```
x <- c(1,2,3)
y <- c(4,5,6)
z <- x * y
print(z)
```

```
## [1]  4 10 18
```

`R` performs **element-wise** multiplication: $(x_1 \times y_1)\ (x_2 \times y_2)..$

Simple summary statistics of vectors

- When the size of vectors grow, eye-balling becomes difficult
- Instead, we typically **calculate summaries**.
- For **simple summaries** (eg means) — **base R** functions:

```
x <- c(1,2,3)
mean(x)
```

```
## [1] 2
```

What if we have **missing values**? Set `na.rm=TRUE`

```
x <- c(1,2,3,NA,NA)
mean(x,na.rm = TRUE)
```

```
## [1] 2
```

Object classes

- So far, we've only considered **integer** values, but R has several **object classes**, including:
  - **Integer**: `c(1, 2, 3)`
  - **Numeric**: `c(1.103, 2.251, 3.888)`
  - **Character**: `c("orange", "blue", "green")`
  - **Boolean**: `c(TRUE,TRUE,FALSE)`
- Object classes differ in their properties/functionality.
- E.g. as one would expect, **this does not work**:

```
x <- "orange"
y <- "blue"
z <- x + y
```

Important property of vectors

- **Homogeneity:** All elements/values must be of the **same type**
    - What happens if we try to **combine different types**?
    - R **coerces** all to the **most flexible type**
- Least to most flexible: `boolean` < `int` < `numeric` < `character`

```
# Ex: Try to create a vector with four diferent object classes
vec <- c(TRUE, 0.03335, 5, "hello")
print(vec)
```

```
## [1] "TRUE"    "0.03335" "5"       "hello"
```

```
# What type?
class(vec)
```

```
## [1] "character"
```

Subsetting – very common operation

- **Subsetting** — *extract subset* of data for *further analysis*
- In `R`, this can be done with brackets: `[]`
- **Simplest** form: **list positions** of the wanted elements:

```
# Some random vector of numbers
x <- c(1,5,3,30,7,15,8)
# Select first three values of vector x
x[c(1,2,3)]
```

```
## [1] 1 5 3
```

```
# Select the 1st and 5th item
x[c(1,5)]
```

```
## [1] 1 7
```

Conditional subsetting

A more sophisticated —and arguably more useful— kind of subsetting is **condtional subsetting**:

```
# Select values of x that are larger than 10
x[x>10]
```

```
## [1] 30 15
```

How does this work?

```
x>10
```

```
## [1] FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE
```

Every item-position gets a `TRUE` / `FALSE` value, and all `TRUE` positions are kept (here: 4 and 6)

Application: exclude missing values

- First, to test whether individual elemements in a vector are missing, we can use the `is.na()` function

```
x <- c(1,NA,3)
is.na(x)
```

```
## [1] FALSE  TRUE FALSE
```

- Thus, to filter out missing values: couple `is.na()` with `[]`

```
x <- c(1,NA,3)
x[!is.na(x)]
```

```
## [1] 1 3
```

Note: `!` is a `NOT` operator

Moving beyond 1D $\longrightarrow$ Matrices?

- Usually — not working with **one dimension**, but **many**
- **Matrices** allow for **2D** storing (N rows $\times$ P columns)
- Specifically, matrices **combine vectors** of the same length:
    - `cbind()` to combine vectors **column-wise**
    - `rbind()` to combine vectors **row-wise**

```
x <- c(1,2,3)
y <- c(4,5,6)
z <- cbind(x,y)
print(z)
```

```
##      x y
## [1,] 1 4
## [2,] 2 5
## [3,] 3 6
```

However… matrices are not apt for *data wrangling*

- **While** matrices are **essential** for:
    - mathematical operations, optimization procedures
    - statistical computation
- They…
    - **Cannot** deal with **different object classes**:
        - *ID-variables*: names, dates
        - *Measurements*: income, education-level
    - **Lack functionality** for *data wrangling*

data.frames

Can store columns (variables) of different classes and have extensive data

wrangling functions

Creating a data.frame

- We **create** a data.frame using the `data.frame()` function
- **Input arguments** are expected to be **vectors of equal length**

```
# Three column/variable data.frame
df3 <- data.frame(x = c(1,2,3),
                  y = c("a","b","c"),
                  z = c(TRUE,FALSE,TRUE))
print(df3)
```

```
##   x y     z
## 1 1 a  TRUE
## 2 2 b FALSE
## 3 3 c  TRUE
```

Importing data.frames

- We rarely create datasets from scratch – we typically import already prepared ones.

- The **file-type** determines **how** to import data into R:

    - `read.csv()` for **.csv** files
    - `read.table()` for **.txt** files
    - `read.dta()` for stata **.dta** files

Child-IQ dataset(s)

In this presentation, we'll consider two datasets containing information about a sample of 400 children and their mothers:

- `childiq.csv` — IQ test scores of children at age 3
- `motherinfo.txt` — Age and Education of their mothers

Thus:

```
# Import datasets
childiq <- read.csv(file = "/source/to/childiq.csv")
motherinfo <- read.table(file = "/source/to/motherinfo.txt",
                         header = TRUE)
```

(This dataset will be re-used in future labs as well)

Basic `data.frame` information

- When retrieving a new dataset – good to get an **overview**.
- To obtain **basic information** about a data.frame, these commands are useful:
  - `str()`: returns **structure** of dataset (eg data types)
  - `head()` and `tail()`: returns first/last `n` rows
  - `summary()`: returns **quantile info** on all variables
  - `dim()`: returns the **dimensions** of the data.frame
  - `nrow()` and `ncol()`: returns the number of rows/columns

Inspecting **childiq**

```
str(childiq)
```

```
## 'data.frame':    400 obs. of  3 variables:
##  $ id        : int  186 18 226 127 17 297 188 126 246 144 ...
##  $ ppvt      : int  80 79 50 87 73 94 76 82 65 90 ...
##  $ test_month: int  1 2 3 3 2 2 2 3 3 2 ...
```

```
head(childiq)
```

```
##    id ppvt test_month
## 1 186   80          1
## 2  18   79          2
## 3 226   50          3
## 4 127   87          3
## 5  17   73          2
## 6 297   94          2
```

Inspecting **motherinfo**

```
str(motherinfo)
```

```
## 'data.frame':    400 obs. of  3 variables:
##  $ id      : int  143 240 301 389 261 48 128 122 69 161 ...
##  $ momage  : int  18 20 21 23 25 20 27 23 29 22 ...
##  $ educ_cat: int  2 2 3 2 1 3 4 2 2 2 ...
```

```
head(motherinfo)
```

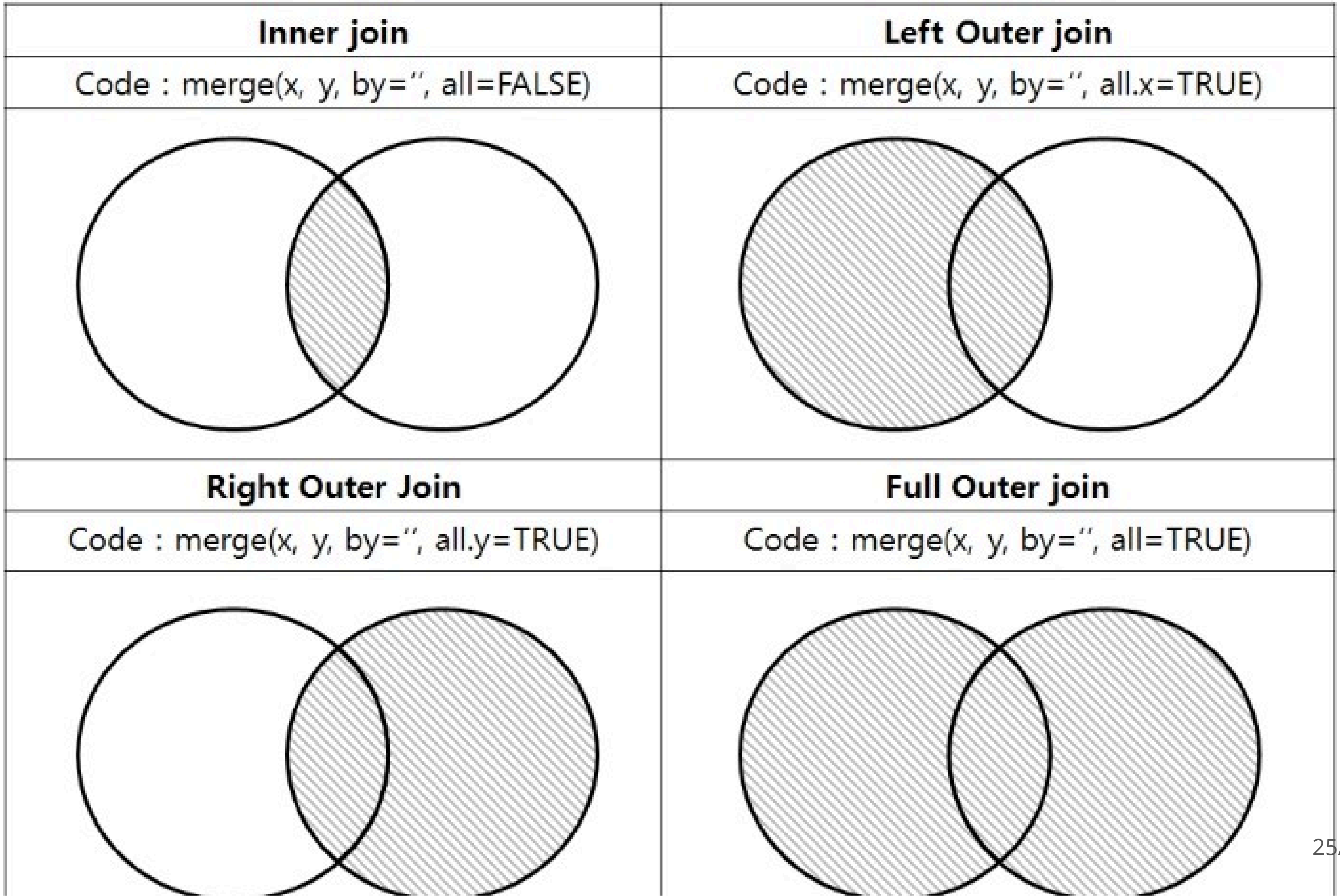
```
##    id momage educ_cat
## 1 143     18        2
## 2 240     20        2
## 3 301     21        3
## 4 389     23        2
## 5 261     25        1
## 6  48     20        3
```

Combining data.frames

- Often —as we do here— we have **multiple linked datasets** that we want to **combine**
- The 3 most common ways of combining `data.frames` are:
  - Paste *cols*: `cbind()` – assumes that **rows** are **aligned**
  - Paste *rows*: `rbind()` – assumes that **columns** are **aligned**
  - Horizontal **merge** based on **keys** — `merge()`

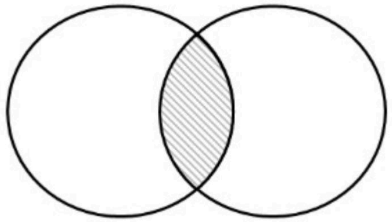
In this presentation, we focus on **horizontal merging** with **keys**

Different kinds of `merges`



Merging `childiq` & `mothersinfo`

Suppose we want to perform an **inner-join** of `childiq` & `mothersinfo`:



```
cm <- merge(x = childiq, y = motherinfo,
                     by='id', all=FALSE)
head(cm,n=3)
```

```
##   id ppvt test_month momage educ_cat
## 1  1  120          2     21        2
## 2  2   89          3     17        1
## 3  3   78          1     19        2
```

The `by` argument specifies the **shared key column** that uniquely identifies rows in both datasets.

Did we loose any observations?

- Inner-joins only keep cases that exist in both datasets.
- Did we loose any cases in our merge?

```
nrow(childiq)
```

```
## [1] 400
```

```
nrow(motherinfo)
```

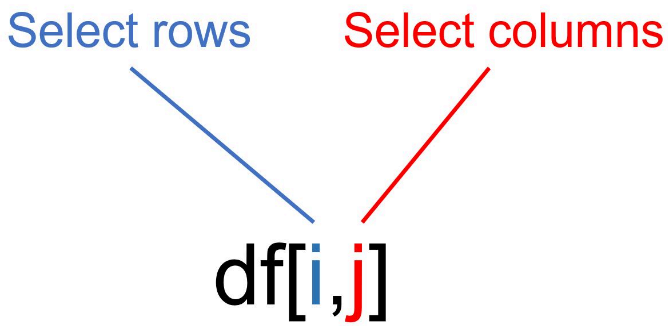
```
## [1] 400
```

```
nrow(cm)
```

```
## [1] 400
```

Subsetting `data.frames`

- Just as for vectors, we can use brackets `[]` to select elements of a `data.frame`
- In contrast to vectors, `data.frame's` are **2D**



- `data.frame` elements can be selected with:
  - Indices
  - Condtional statements
  - Names (only for columns)

Subsetting columns (`cm`)

There are 4 standard ways of selecting a **single-column**:

```
# Select the column "age"
cm[,2]
cm[,"age"]
cm$age        # Works since data.frames also are list-objects
cm[["age"]]  # Works since data.frames also are list-objects
```

Two ways of selecting **multiple columns**:

```
# Select the columns "age" and "education"
cm[,c(1,2)]
cm[,c("age","education")]
```

Subsetting rows (`cm`)

Example of **index**-subsetting:

```
# Select row 5-6 of the cm data
cm[c(5:6),]
```

```
##   id ppvt test_month momage educ_cat
## 5  5  115          1     26        4
## 6  6   97          3     20        1
```

Example of **conditional** subsetting:

```
# Select all cases where the mother is of age 29
cm[cm$momage == 29,]
```

```
##      id ppvt test_month momage educ_cat
## 12   12   64          2     29        4
## 28   28  102          3     29        2
## 69   69  107          1     29        2
## 88   88  104          2     29        4
## 295 295  107          1     29        3
## 312 312  108          1     29        2
```

Again, how it works.

Create a subset (20 rows) and check.

```r
# Subset of 20 rows of cm
cm_20 <- cm[1:20,]
# Testing which elements of momage are equal to 29
cm_20$momage == 29
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```r
# This TRUE/FALSE vector is the inserted to select rows
cm_20[cm_20$momage == 29,]
```

```
##    id ppvt test_month momage educ_cat
## 12 12   64          2     29        4
```

Variable summaries

Now that we've got a grip on **row/variable selection**, we can consider more fine grained summaries of `data.frame` **variables**:

- **Continuous variables**:
  - Univariate: `mean()`, `median()`,`sd()`
  - Bivariate: `cor()`, `cov()`
- **Categorical variables**:
  - Univariate/Bivariate: `table()` – *contingency table*

Univariate variable summaries

Continuous example

```
# Mean & Median of IQ-score
mean(cm$ppvt) ; median(cm$ppvt)
```

```
## [1] 86.9325
```

```
## [1] 90
```

Categorical example

```
# Number of mothers in each education category
table(x = cm$educ_cat)
```

```
## x
##   1   2   3   4
##  85 212  76  27
```

Bivariate variable summaries

Continuous example

```
# Correlation between mom's age and childs iq-score
cor(x = cm$momage, y=cm$ppvt)
```

```
## [1] 0.1105672
```

Categorical example

```
# Association between childs education categ and month
table(x = cm$educ_cat, y = cm$test_month)
```

```
##    y
## x    1  2  3
##   1 30 27 28
##   2 75 71 66
##   3 37 17 22
##   4  9 11  7
```

Conditional variable summaries

We can also compute summaries of **subsetted data**:

1. Subset subpopulation of interest
2. Select column
3. Compute summary statistic

For example:

```
# Compute mean(ppvt) for mothers with an education == 1
mean(cm[cm$educ_cat==1,]$ppvt)
```

```
## [1] 78.44706
```

Conditional variable summaries (several categories)

- How can we compute the mean age **for all** `educ_categs`?
  - We could copy code several times — but, not efficient!
  - Instead, we can use `R`'s `aggregate()` function.

```
# Mean age by education-level
aggregate(x = cm$momage,
          by = list(cm$educ_cat),
          FUN = mean)
```

```
##   Group.1        x
## 1       1 21.58824
## 2       2 22.69811
## 3       3 23.32895
## 4       4 25.77778
```

Conditional variable summaries (several variables)

- How can we compute **conditional means** of **several variables**?
  - We could copy code many times — not efficient!
  - Again to the rescue: the `aggregate()` function, but formula-style!

```
# Mean ppvt & mean momage by education-level
aggregate(. ~ educ_cat,
          data = cm[,c("educ_cat","ppvt","momage")],
          FUN = mean)
```

```
##   educ_cat     ppvt   momage
## 1        1 78.44706 21.58824
## 2        2 88.70283 22.69811
## 3        3 87.78947 23.32895
## 4        4 97.33333 25.77778
```

Basic data cleaning

Two important steps of **data cleaning** are:

- Identifying **missing values**:
  - in *any* column — `complete.cases()`
  - in a *specific* column — `is.na(column)`
- Identifying **duplicated rows** — `duplicated()`

Basic data cleaning (2)

```
# See if there are any missing values for the column "ppvt"
cm[is.na(cm$ppvt),]
```

```
## [1] id         ppvt       test_month momage     educ_cat
## <0 rows> (or 0-length row.names)
```

```
# complete.cases() returns TRUE for rows with no missing values
cm[!complete.cases(cm),]
```

```
## [1] id         ppvt       test_month momage     educ_cat
## <0 rows> (or 0-length row.names)
```

```
# duplicated() returns TRUE for duplicated rows
cm[duplicated(cm),]
```

```
## [1] id         ppvt       test_month momage     educ_cat
## <0 rows> (or 0-length row.names)
```

Creating new columns

To create a new column in a `data.frame`, we:

1. Select a non-existing column
2. Assign a vector —of the same length as the df— to it

For example, a common transformation: `log_age` $= log(age)$

```
# Creating a new variable "log_age" as a function of "age"
cm$log_momage <- log(cm$momage)
head(cm,n=3)
```

```
##   id ppvt test_month momage educ_cat log_momage
## 1  1  120          2     21        2   3.044522
## 2  2   89          3     17        1   2.833213
## 3  3   78          1     19        2   2.944439
```

Creating new columns (2)

- Another common case of variable creation is when we want **discretize** a particular variable, i.e. *transform a continious variable into a categorical one*.
- In `R`, `ifelse()` is great for this purpose!

```
# Creating a new variable "ppvt_binary" that binarizes "ppvt"
cm$ppvt_binary <- ifelse(test = cm$ppvt>100,
                         yes = 1,
                         no = 0)
head(cm,n=3)
```

```
##   id ppvt test_month momage educ_cat log_momage ppvt_binary
## 1  1  120          2     21        2   3.044522           1
## 2  2   89          3     17        1   2.833213           0
## 3  3   78          1     19        2   2.944439           0
```

Renaming columns

- To change column names in R, we can use `colnames()`
- Suppose, for example, that we don't like the `"_"` part of `log_age`:

```
# Change the name "log_momage" to "logage"
# Note: "ncol" returns the number of columns
# Thus: changes the name of the next-last column
colnames(cm)[ncol(cm)-1] <- "logmomage"
head(cm,n=3)
```

```
##   id ppvt test_month momage educ_cat logmomage ppvt_binary
## 1  1  120          2     21        2  3.044522           1
## 2  2   89          3     17        1  2.833213           0
## 3  3   78          1     19        2  2.944439           0
```

Removing columns

- To **remove a column** from a `data.frame`, we set it to `NULL`, e.g.
- Suppose we changed our minds and want to **remove** the **logged** version of `momage`:

```
# Delete column "logmomage"
cm$logmomage <- NULL
head(cm,n=3)
```

```
##   id ppvt test_month momage educ_cat ppvt_binary
## 1  1  120          2     21        2           1
## 2  2   89          3     17        1           0
## 3  3   78          1     19        2           0
```

Exporting a `data.frame`

Just as we can *import* different file-types (e.g. .csv, .txt) we can also **export** our `data.frames` to different file-types:

- `write.table()` for `.txt`
- `write.csv()` for `.csv`

```
# Export cm to .csv file
write.csv(x = cm, file = ".../export/here/cm.csv", row.names = FALSE)
```

To avoid exporting a column of *row names*, set `row.names=FALSE`

ggplot2

General framework for plots in R

Basic steps to making a `ggplot`

1. Call main function `ggplot()` and specify **data** (`df`) to plot

```
ggplot(df,...)
```

2. Specify **which variables** we want on **which axis** (`x=x,y=y`)

```
ggplot(df, aes(x=x, y=y))
```

3. Specify which **kind of plot** we want: **scatterplot**

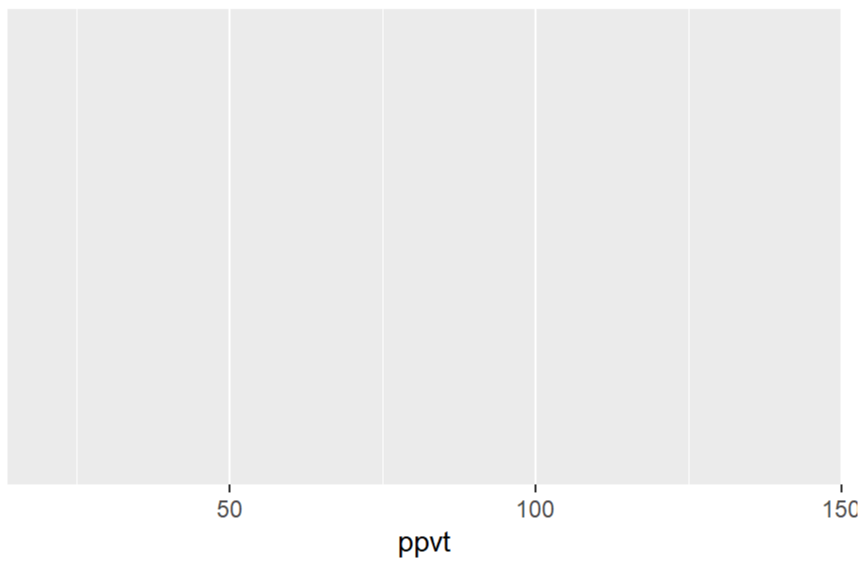
```
ggplot(df, aes(x=x, y=y)) + geom_point()
```

- `aes()` is a function that **maps data** to `geom`-objects.
- Here, we have a `geom_point()` for a scatterplot
- We **add more properties** to the graph with `+`

Step 1 & 2: Select data and variable(s)

- Dataset: `cm`
- x-axis: `ppvt`

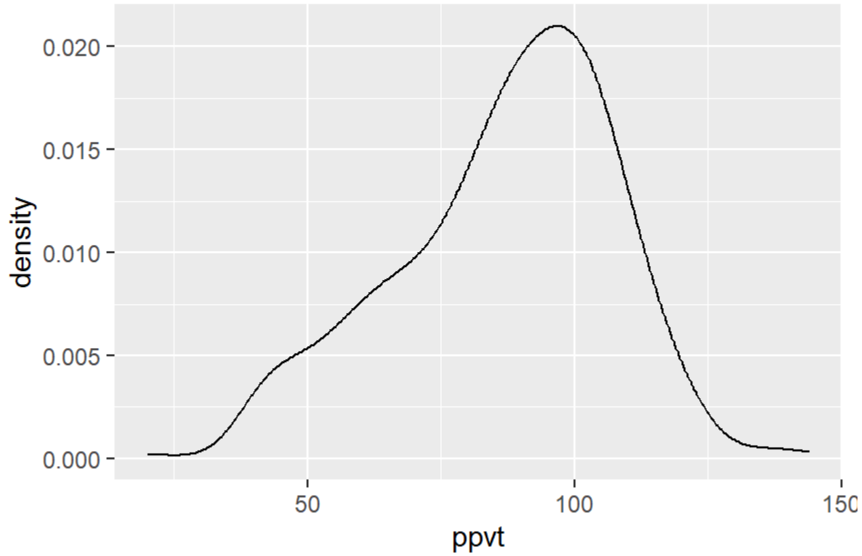
```
ggplot(cm, aes(x=ppvt))
```



Step 3: Select plot-type

- Plot-type: **density plot** $\rightarrow$ `geom_density`

```
ggplot(cm, aes(x=ppvt)) + geom_density()
```

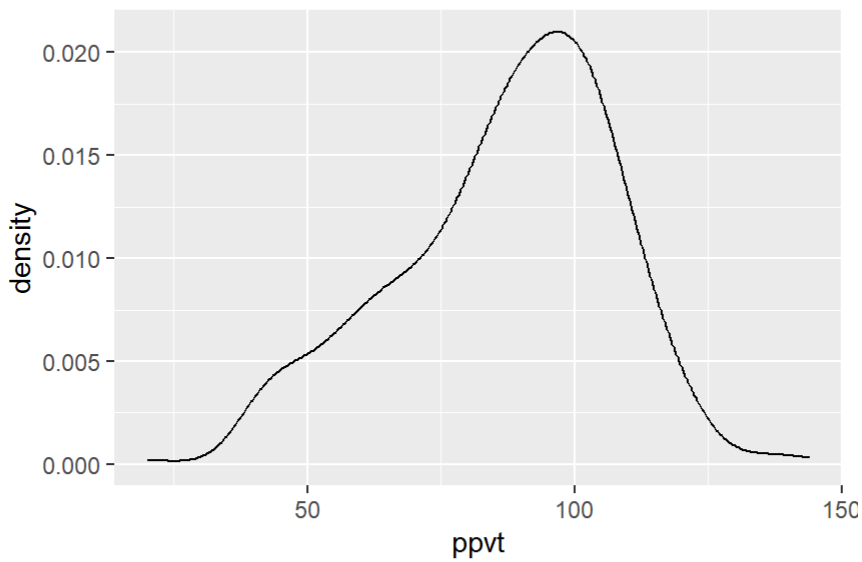


Step 4: Condition on other variable

- Suppose we're also interested in how`ppvt` differ **by** `educ_cat`
- Specify argument `fill` or `color`

```
ggplot(cm, aes(x=ppvt,fill=educ_cat)) + geom_density()
```

```
## Warning: The following aesthetics were dropped during statistical transformation: fill.
## ℹ This can happen when ggplot fails to infer the correct grouping structure in
##   the data.
## ℹ Did you forget to specify a `group` aesthetic or to convert a numerical
##   variable into a factor?
```



Problem

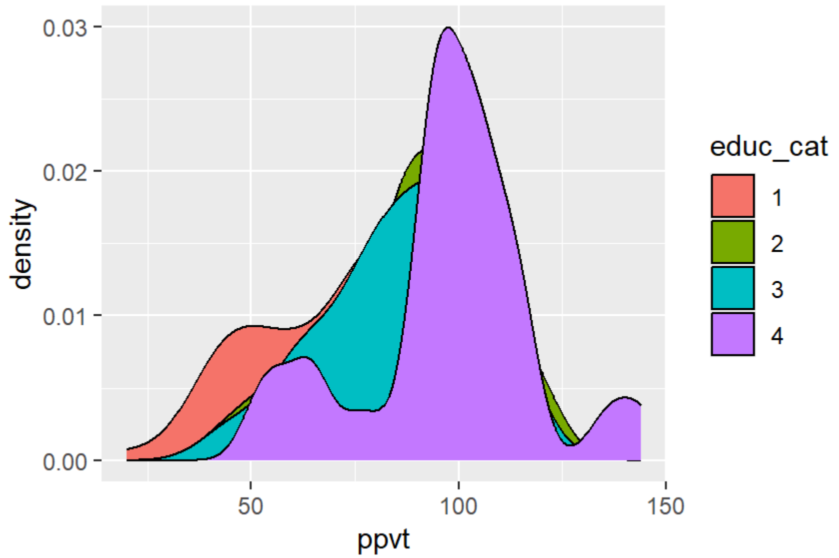
- `ggplot` thinks `educ_cat` is a **numeric** variable
- How do we address this? Change its format!
- Specifically, make it a `Factor` variable!
  - `factor()` instantiates a **categorical structure** to vectors and also **allow an order** to be specified.
  - If the order is not specified, it defaults to alphabetical order

```
cm$educ_cat <- factor(cm$educ_cat, levels = c(1,2,3,4))
```

Let’s try again!

- Now with `educ_cat` as a factor variable

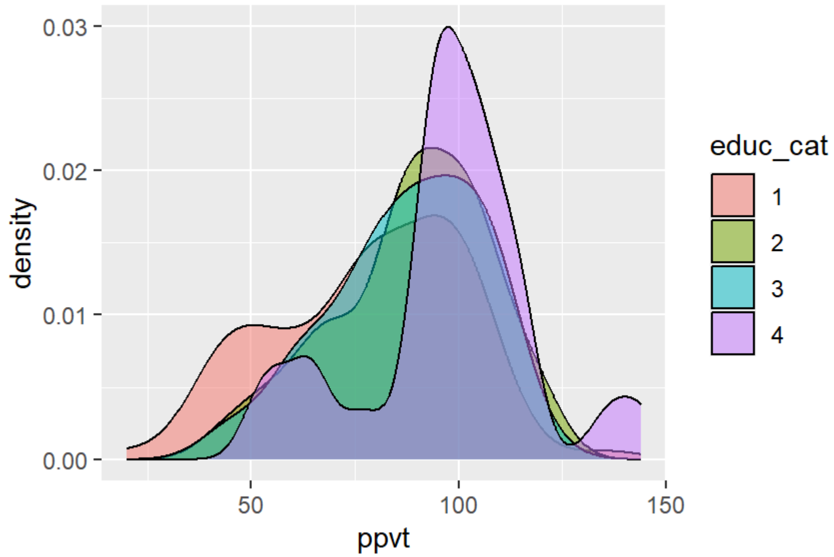
```
ggplot(cm, aes(x=ppvt,fill=educ_cat)) + geom_density()
```



There’s a bit too much overlap. Hard to see some categories…

Increase transparency via `alpha`

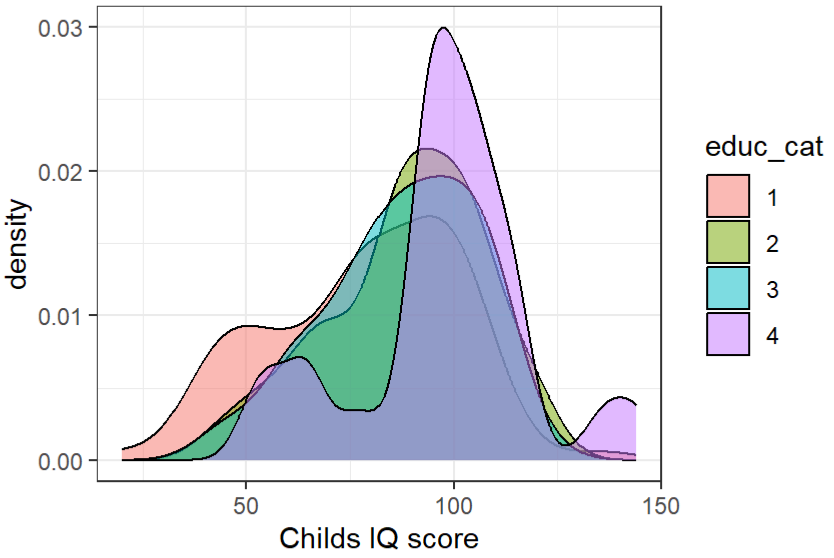
```
ggplot(cm, aes(x=ppvt,fill=educ_cat)) + geom_density(alpha=0.5)
```



Note: in relation to other categories, distribution for `educ_cat==4` cleary skews right

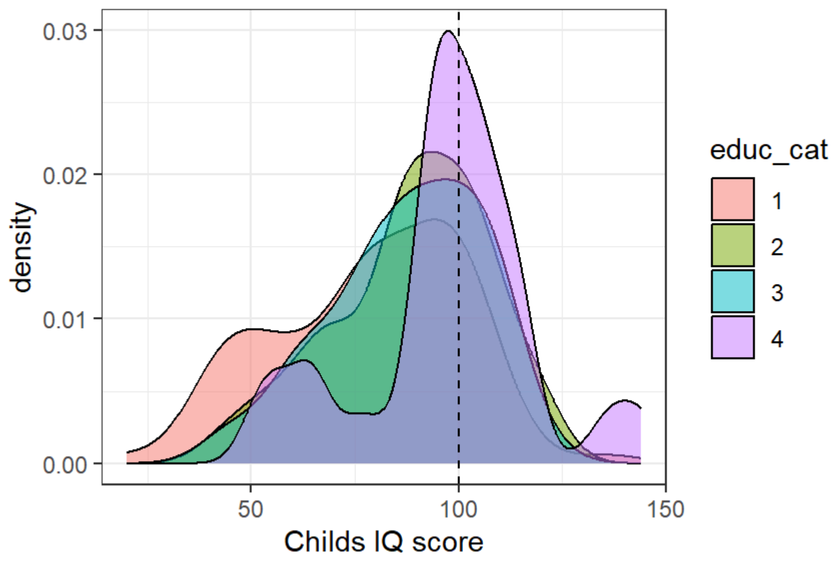
More: Labels and theme

```
ggplot(cm, aes(x=ppvt,fill=educ_cat)) +
  geom_density(alpha=0.5) +
  xlab('Childs IQ score') +
  theme_bw()
```



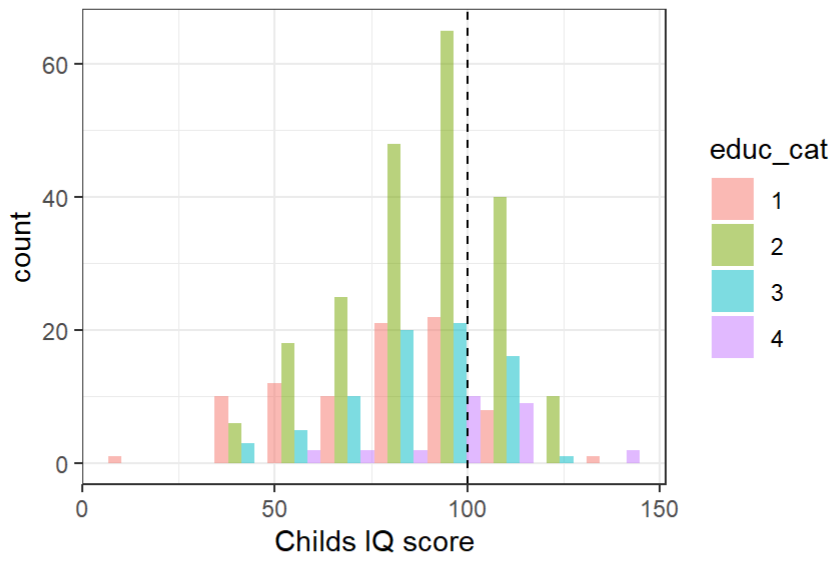
Insert a line. No problem.

```
# Insert a vertical line at x=100
ggplot(cm, aes(x=ppvt,fill=educ_cat)) +
  geom_density(alpha=0.5) +
  geom_vline(xintercept = 100, linetype='dashed') +
  xlab('Childs IQ score') +
  theme_bw()
```



Easy swap of plot-type (histogram)

```
# HISTOGRAM
ggplot(cm, aes(x=ppvt,fill=educ_cat)) +
  geom_histogram(alpha=0.5,position = "dodge",bins = 10) +
  geom_vline(xintercept = 100, linetype='dashed') +
  xlab('Childs IQ score') +
  theme_bw()
```



data.table (teaser)

R for “big data”

Why move beyond `data.frames`?

Although `data.frames` work well for:

- Small datasets
- Simple data wrangling tasks

They…

- **Struggle** with **larger datasets**
- Have **limited functionality** $\rightarrow$ slows workflow

To address these limitations, **extensions** have been developed:

- `data.table` (considered here)
- `dplyr` & `tidyr`

`data.tables`

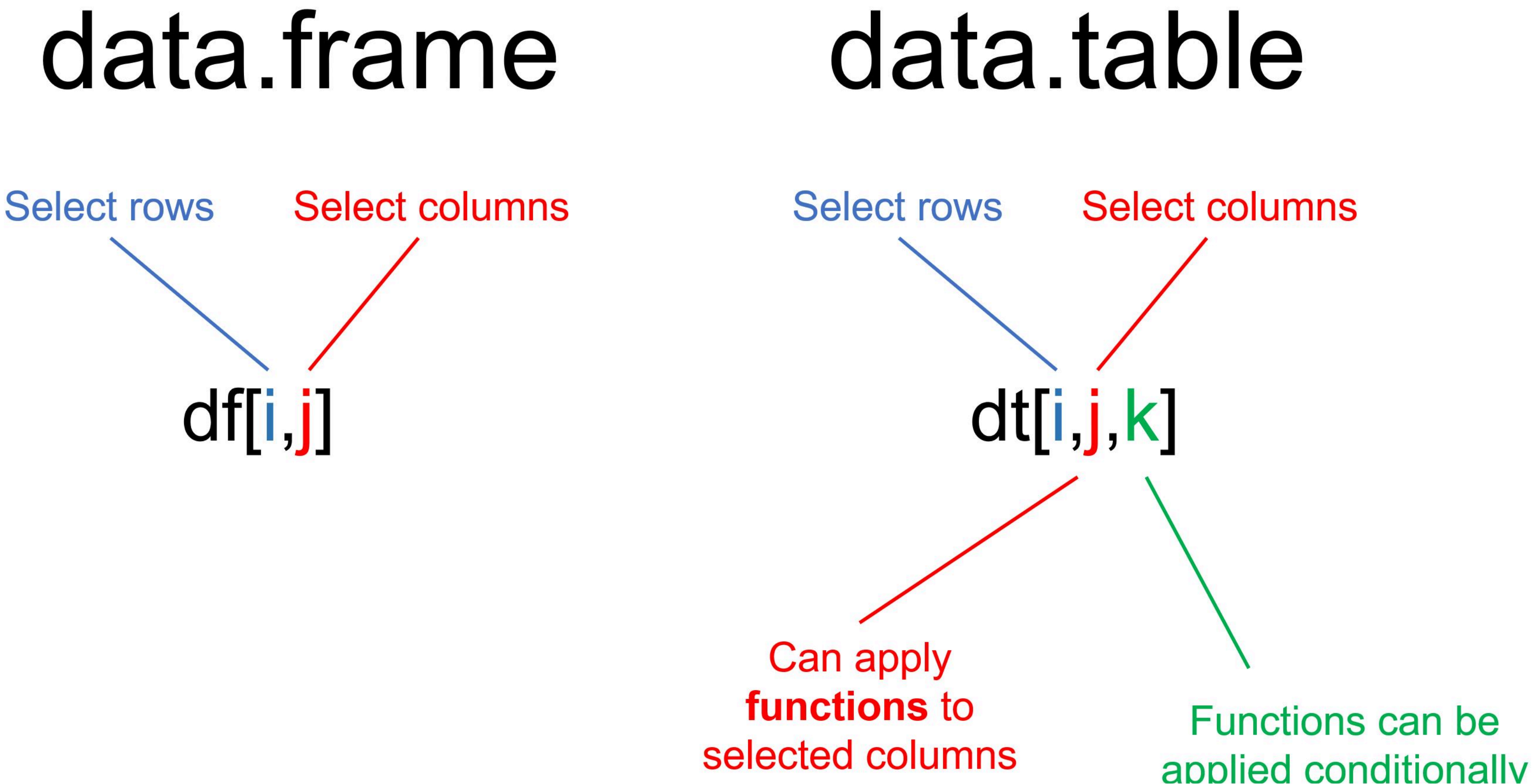
- Developed by **Matt Dowle et al.** (starting in 2008)
- Directly extends upon `data.frames`, s.t.
  - All functions consid. today works for `data.tables` too
  - Plus: **lots of new functionality**

In short, *objective*:

- Reduce **programming time**
- Reduce **computation time**

What distinguishes `data.tables`?

- *Key practical difference*: subsetting mechanism:
  - Easier to subset rows (as we'll see)
  - **Functions** can be applied to **selected columns** within `[]`
  - And, optionally, **conditional** on other variables.



Creating a `data.table`

- Very similar to how we create a `data.frame`:

```
library(data.table) # Need to install & load package first
dt <- data.table(x = c(1,2,3),
                 y = c("a","b","c"),
                 z = c(TRUE,FALSE,TRUE))
print(dt)
```

```
##    x y     z
## 1: 1 a  TRUE
## 2: 2 b FALSE
## 3: 3 c  TRUE
```

Importing `data.tables`

- Again, we rarely create new data sets.
- To import `data.tables`, we use the function `fread()`
- On larger data, it is much (!) faster than `base R` versions
- Another (+): *auto-detect* of what separates columns.

```
# Import datasets (automatic detection of what separates the columns)
childiq_dt <- fread(file = "/source/to/childiq.csv")
motherinfo_dt <- fread(file = "/source/to/motherinfo.txt")

# Inspect
head(childiq_dt, 3)
```

```
##     id ppvt test_month
## 1: 186   80          1
## 2:  18   79          2
## 3: 226   50          3
```

Merging `data.tables`

The same syntax that we used to merge `data.frames` also applies to `data.tables`

```
# Merge data.tables
cm_dt <- merge(x = childiq_dt,
               y = motherinfo_dt,
               by='id',
               all=FALSE)
head(cm_dt,n=3)
```

```
##    id ppvt test_month momage educ_cat
## 1:  1  120          2     21        2
## 2:  2   89          3     17        1
## 3:  3   78          1     19        2
```

Subsetting rows

Index-subsetting can be done exactly as with `data.frames`, e.g.

```
cm_dt[c(2:4),]
```

```
##    id ppvt test_month momage educ_cat
## 1:  2   89          3     17        1
## 2:  3   78          1     19        2
## 3:  4   42          3     20        1
```

However (!) **Conditional subsetting** is **much cleaner** — due to ability to “look inside”: no need for `$`

```
cm_dt[educ_cat==2 & momage==29 & test_month == 1,]
```

```
##     id ppvt test_month momage educ_cat
## 1:  69  107          1     29        2
## 2: 312  108          1     29        2
```

Subsetting columns

We can also select colums the same way we do with `data.frames`:

```
cm_dt[,3]
cm_dt[,c("momage")]
cm_dt[["momage"]]
```

But, additionally, we can also use `lists` (again, due to the ability to directly target elements within)

```
cm_dt[,list(momage)]
```

```
##      momage
##   1:     21
##   2:     17
##   3:     19
##   4:     20
##   5:     26
##  ---
## 396:     21
## 397:     20
## 398:     25
## 399:     18
```

Select + compute on column

The big difference — computing on columns within `[]`

Unconditional means:

```
cm_dt[,list(mean_momage = mean(momage))]
```

```
##    mean_momage
## 1:       22.79
```

Conditional means (just add a `by`):

```
cm_dt[,list(mean_momage = mean(momage)), by = 'educ_cat']
```

```
##    educ_cat mean_momage
## 1:        2    22.69811
## 2:        1    21.58824
## 3:        4    25.77778
## 4:        3    23.32895
```

Compute on (pot.) many columns

- Use `lapply()` within (columns are list objects)
- `.SD` allows us to specify which columns we want **apply** on

```
# Unconditional means
cm_dt[,lapply(.SD,mean),.SDcols=c("ppvt","momage")]
```

```
##       ppvt momage
## 1: 86.9325  22.79
```

```
# Conditional means
cm_dt[,lapply(.SD,mean),.SDcols=c("ppvt","momage"), by="educ_cat"]
```

```
##    educ_cat     ppvt   momage
## 1:        2 88.70283 22.69811
## 2:        1 78.44706 21.58824
## 3:        4 97.33333 25.77778
## 4:        3 87.78947 23.32895
```

Chaining

Another cool feature of `data.tables` is that we can **stack** `[]`, i.e. `[][][]`, and continue applying functions.

```
cm_dt[,lapply(.SD,mean),.SDcols=c("ppvt","momage"),
              by="educ_cat"][order(educ_cat,decreasing = F)]
```

```
##    educ_cat     ppvt   momage
## 1:        1 78.44706 21.58824
## 2:        2 88.70283 22.69811
## 3:        3 87.78947 23.32895
## 4:        4 97.33333 25.77778
```

I.e. here we **first** compute means of `ppvt` and `momage` by `educ_cat`, and **then** order the result according to `educ_cat`.

Updating a `data.table`

- **Note:** In previous slides, we **never modified** the **original** data.table `cm_dt`.
- To add new columns to `data.tables`, we use the `:=` operator:

```
# Create new variable
cm_dt[,logmomage := log(momage)]
head(cm_dt,3)
```

```
##    id ppvt test_month momage educ_cat logmomage
## 1:  1  120          2     21        2  3.044522
## 2:  2   89          3     17        1  2.833213
## 3:  3   78          1     19        2  2.944439
```

```
# Remove variable
cm_dt[,logmomage := NULL]
```

This was just a tease…

- `data.table` provides an extremely rich functionality – most of which were not covered here.
- This part of the presentation is aimed to encouraging you to **explore beyond base R**.
- Especially if you want to work on “big data”.

Next up — assignments!