

— Parte 4. —

Tidyverse

Paolo Bosetti (paolo.bosetti@unitn.it)

Data creazione: 2022-01-21 09:53:45

Indice

1	Introduzione al Tidyverse	1
1.1	Strutture dati	2
1.2	Lettura e importazione dati	3
1.3	Manipolazione dati	4
1.4	Stringhe	9
1.5	Grafici base	9
1.5.1	Aesthetics	10
1.5.2	Facets	12
1.5.3	Scale	13
1.5.4	Palette	13
1.5.5	Limiti degli assi	14
2	Esempi	16
2.1	Distribuzioni	16
2.1.1	Distribuzioni discrete	16
2.1.2	Distribuzioni continue	17
2.1.3	Iistogrammi e QQ-plot	18
2.1.4	Boxplot	21
2.1.5	Iistogrammi marginali	21
2.2	Modelli	22
2.3	Serie temporali	24
2.4	Diamanti grezzi!	32
2.4.1	Fattori confusi	32
2.4.2	Fattori non ortogonali	38

1 Introduzione al Tidyverse

Tidyverse rappresenta una collezione di librerie nate attorno a RStudio (spesso dagli stessi sviluppatori), allo scopo di modernizzare R sia come linguaggio sia come potenzialità soprattutto nella creazione di grafici complessi e nella gestione di *big data*.

Un elemento comune alle librerie Tidyverse è quello di sostituire funzioni nidificate, tipiche di R, con sequenze di operazioni: da `f(g(x))` a `g(x) %>% f()`. Lo scopo è generalmente rendere il codice più leggibile e flessibile.

Questo nuovo approccio è reso possibile dal due caratteristiche di R:

1. R è un linguaggio funzionale, in cui le funzioni sono *first class objects*;
2. Le funzioni accettano parametri nominati in qualsiasi ordine, anonimi nell'ordine in cui sono definiti;
3. Gli stessi operatori come `+`, `-`, ..., sono in realtà *funzioni* con due argomenti;
4. Gli operatori possono essere *ridefiniti* per nuovi oggetti, ed è possibile definire *nuovi operatori* del tipo `%op%`, dove `op` può essere un qualsiasi insieme di caratteri.

Grazie a queste caratteristiche, Tidyverse introduce la possibilità di accodare operazioni, avendo ridefinito l'operatore `+`, e di passare strutture dati mediante l'operatore `%>%` (*pipe*). In quest'ultimo caso un esempio è particolarmente efficace:

```
set.seed(123)
(x <- rnorm(10)) %>% mean %>% round(digits=2)

## [1] 0.07
```

che è equivalente alla “vecchia” sintassi:

```
set.seed(123)
round(mean(x <- rnorm(10)), digits=2)

## [1] 0.07
```

ma molto più leggibile.

Nell’ultimo esempio, si noti che il vettore `x` **rimane invariato**, mentre se avessimo scritto `x <- rnorm(10) %>% mean %>% round(digits=2)`, allora `x` conterrebbe la media del vettore generato (che a sua volta andrebbe perso).

In generale, tuttavia, la “nuova moda” non sostituisce completamente la vecchia, per alcuni motivi:

- Tidyverse è comunque abbastanza pesante
- se certe funzionalità (ad esempio il *pipe*) sostanzialmente non hanno svantaggi, altre vanno poco d'accordo con librerie e funzioni “vecchie”, ed è soprattutto il caso dei grafici
- in certi casi, ed è di nuovo il caso dei grafici, le vecchie funzioni sono semplicemente più concise e rapide, quindi preferibili se si generano grafici per analisi dati piuttosto che per la pubblicazione finale
- spesso le funzioni grafiche sono molto potenti, ma tendono a nascondere all’utente i dettagli degli algoritmi utilizzati

Le librerie che fanno parte di Tidyverse possono essere caricate individualmente oppure con un unica istruzione `library(tidyverse)`. Tuttavia, se se ne usa solo un limitato sottoinsieme spesso conviene caricarle individualmente.

1.1 Strutture dati

Le librerie Tidyverse si aspettano i dati in formato *tidy*, cioè un’osservazione per riga, un osservando per colonna. I dati sono tipicamente contenuti in data frame o, preferibilmente, nella nuova classe `tibble`, che è per lo più interscambiabile con i data frame, dato che ne eredita la classe. Una `tibble` può essere creata convertendo un data frame, oppure passando i dati per colonne (come per la funzione `data.frame()`), oppure ancora passando i dati **per righe**, mediante la funzione `tribble()` (notare la `r`, per *rows*, oppure come TRansposed tIBBLE):

```
n <- 10
a <- data.frame(
  In=1:n,
  Out=(1:n)^2 + rnorm(n, sd=1),
  Size=rnorm(n, 10, 0.1)) %>%
  tribble %>%
  print

## # A tibble: 10 x 3
##       In     Out   Size
##   <dbl> <dbl> <dbl>
#> 1     1     1.0  9.8 
#> 2     2     4.0  9.5 
#> 3     3     9.0  9.2 
#> 4     4    16.0  9.0 
#> 5     5    25.0  8.8 
#> 6     6    36.0  8.6 
#> 7     7    49.0  8.4 
#> 8     8    64.0  8.2 
#> 9     9    81.0  8.0 
#> 10    10   100.0  7.8
```

```
##      <int> <dbl> <dbl>
## 1     1   2.22  9.89
## 2     2   4.36  9.98
## 3     3   9.40  9.90
## 4     4  16.1   9.93
## 5     5  24.4   9.94
## 6     6  37.8   9.83
## 7     7  49.5  10.1
## 8     8  62.0  10.0
## 9     9  81.7  9.89
## 10    10 99.5  10.1

(b <- tribble(
  ~name, ~age,
  "Paolo", 50,
  "Luca", 45,
  "Lucia", 38,
  "Anna", 52
)) %>% knitr::kable()
```

	name	age
Paolo	50	
Luca	45	
Lucia	38	
Anna	52	

Gli argomenti della funzione `tibble()` sono valutati in maniera *lazy*, cioè solo quando sono effettivamente utilizzati, a differenza che in `data.frame()`:

```
rm(x, y)
try(
  df <- data.frame(x=1:10, y=x^2)
)

## Error in data.frame(x = 1:10, y = x^2) : object 'x' not found

tb <- tibble(x=1:5, y=x^2)
tb

## # A tibble: 5 x 2
##       x     y
##   <int> <dbl>
## 1     1     1
## 2     2     4
## 3     3     9
## 4     4    16
## 5     5    25
```

1.2 Lettura e importazione dati

La libreria `readr` mette a disposizione:

- `read_csv()/read_csv2()`: comma separated value (CSV) files
- `read_tsv()`: tab separated files
- `read_delim()`: general delimited files
- `read_fwf()`: fixed width files

- `read_table()`: tabular files where columns are separated by white-space.
- `read_log()`: web log files

Tutte queste funzioni restituiscono una `tibble`.

```
stat <- read_csv2(mydata("censimento_TAA_2011.csv"))

## i Using ',',',' as decimal and "'.'" as grouping mark. Use `read_delim()` for more control.

## Rows: 2055 Columns: 154

## -- Column specification -----
## Delimiter: ";"
## chr (5): REGIONE, PROVINCIA, COMUNE, LOCALITA, ALTITUDINE
## dbl (149): CODREG, CODPRO, CODCOM, PROCOM, LOC2011, CODLOC, TIPOLOC, AMPLOC, ...

##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.

trace <- read_csv2(mydata("tracciato_2011_localita.csv"))

## i Using ',',',' as decimal and "'.'" as grouping mark. Use `read_delim()` for more control.

## Rows: 154 Columns: 2

## -- Column specification -----
## Delimiter: ";"
## chr (2): Nome Campo, Definizione

##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Ovviamente esistono anche le controparti `write_*`(), che accettano come primo argomento un data frame oppure una `tibble`. Generalmente queste funzioni sono molto più veloci delle controparti della libreria `base` (`write.*()`).

1.3 Manipolazione dati

La libreria `dplyr` mette a disposizione un'ampia scelta di funzioni per la manipolazione di tabelle e `tibble`. In generale, queste funzioni semplificano e rendono più leggibili operazioni che sarebbero comunque realizzabili mediante approcci più standard, anche se con più passaggi e in maniera meno efficiente.

Nella nomenclatura tidy, una `tibble` contiene una o più *variabili* (colonne), ciascuna con valori per uno o più *casi* (righe). Secondo questa convenzione, le principali funzioni di `dplyr` sono:

- `mutate()` aggiunge nuove variabili, funzione di variabili esistenti
- `select()` seleziona variabili in base al loro nome
- `filter()` seleziona casi in base ai loro valori
- `summarise()` riporta più valori ad un unico indicatore
- `arrange()` riordina i casi (righe)

Vediamo esempi di filtraggio:

```
starwars %>%
  filter(homeworld == "Naboo" & species == "Human")

## # A tibble: 5 x 14
##   name      height  mass hair_color skin_color eye_color birth_year sex   gender
##   <chr>     <int> <dbl> <chr>       <chr>       <chr>           <dbl> <chr> <chr>
## 1 Palpati~    170     75 grey        pale       yellow          82 male  mascul-
## 2 Gregar ~    185     85 black       dark       brown            NA male  mascul-
```

```
## 3 Cordé      157    NA brown     light     brown      NA fema~ femini~
## 4 Dormé      165    NA brown     light     brown      NA fema~ femini~
## 5 Padmé A~   165     45 brown     light     brown      46 fema~ femini~
## # ... with 5 more variables: homeworld <chr>, species <chr>, films <list>,
## #   vehicles <list>, starships <list>
```

Selezione:

```
starwars %>%
  select(name, ends_with("color")) %>%
  names

## [1] "name"      "hair_color" "skin_color" "eye_color"
```

Modifica:

```
starwars %>%
  filter(species == "Human") %>%
  mutate(name, bmi = round(mass / ((height / 100) ^ 2), 1)) %>%
  select(name:mass, bmi) %>%
  arrange(desc(bmi), )

## # A tibble: 35 x 4
##       name           height   mass   bmi
##       <chr>        <int>  <dbl> <dbl>
## 1 Owen Lars         178    120  37.9
## 2 Jek Tono Porkins  180    110  34.0
## 3 Darth Vader       202    136  33.3
## 4 Beru Whitesun lars 165     75  27.5
## 5 Wedge Antilles    170     77  26.6
## 6 Luke Skywalker    172     77  26.0
## 7 Palpatine          170     75  26.0
## 8 Lobot              175     79  25.8
## 9 Lando Calrissian   177     79  25.2
## 10 Biggs Darklighter 183     84  25.1
## # ... with 25 more rows
```

Si noti che la variante %<>% dell'operatore pipe consente di fare una *modifica sul posto* di una tibble:

```
library(magrittr)

##
## Attaching package: 'magrittr'

## The following object is masked from 'package:purrr':
##
##     set_names

## The following object is masked from 'package:tidyr':
##
##     extract

humans <- starwars %>% filter(species == "Human")
humans %<>% mutate(name, bmi = round(mass / ((height / 100) ^ 2), 1))
humans

## # A tibble: 35 x 15
##       name   height   mass hair_color skin_color eye_color birth_year sex   gender
##       <chr>    <int>  <dbl> <chr>      <chr>      <chr>      <dbl> <chr> <chr>
## 1 Luke S~     172     77 blond     fair      blue        19 male  masculi~
```

```
## 2 Darth ~ 202 136 none white yellow 41.9 male mascu-
## 3 Leia O~ 150 49 brown light brown 19 fema~ femin-
## 4 Owen L~ 178 120 brown, grey light blue 52 male mascu-
## 5 Beru W~ 165 75 brown light blue 47 fema~ femin-
## 6 Biggs ~ 183 84 black light brown 24 male mascu-
## 7 Obi-Wa~ 182 77 auburn, wh~ fair blue-gray 57 male mascu-
## 8 Anakin~ 188 84 blond fair blue 41.9 male mascu-
## 9 Wilhuf~ 180 NA auburn, gr~ fair blue 64 male mascu-
## 10 Han So~ 180 80 brown fair brown 29 male mascu-
## # ... with 25 more rows, and 6 more variables: homeworld <chr>, species <chr>,
## #   films <list>, vehicles <list>, starships <list>, bmi <dbl>
```

Talvolta i dati sono inclusi come *nomi di riga*. Ciò può avvenire solo se i dati sono originariamente in un data frame, dato che le tibble non supportano i nomi di riga:

```
mtcars %>% row.names

## [1] "Mazda RX4"           "Mazda RX4 Wag"      "Datsun 710"
## [4] "Hornet 4 Drive"     "Hornet Sportabout" "Valiant"
## [7] "Duster 360"         "Merc 240D"        "Merc 230"
## [10] "Merc 280"           "Merc 280C"        "Merc 450SE"
## [13] "Merc 450SL"         "Merc 450SLC"       "Cadillac Fleetwood"
## [16] "Lincoln Continental" "Chrysler Imperial" "Fiat 128"
## [19] "Honda Civic"        "Toyota Corolla"    "Toyota Corona"
## [22] "Dodge Challenger"   "AMC Javelin"       "Camaro Z28"
## [25] "Pontiac Firebird"   "Fiat X1-9"         "Porsche 914-2"
## [28] "Lotus Europa"       "Ford Pantera L"    "Ferrari Dino"
## [31] "Maserati Bora"      "Volvo 142E"

mtcars %>% mutate(
  kpl=round(mpg*0.621371/3.78541, 1),
  lp100k=round(100/kpl, 1)) %>%
  rownames_to_column(var="Model") %>%
  head

## # Model mpg cyl disp hp drat wt qsec vs am gear carb kpl
## 1 Mazda RX4 21.0 6 160 110 3.90 2.620 16.46 0 1 4 4 3.4
## 2 Mazda RX4 Wag 21.0 6 160 110 3.90 2.875 17.02 0 1 4 4 3.4
## 3 Datsun 710 22.8 4 108 93 3.85 2.320 18.61 1 1 4 1 3.7
## 4 Hornet 4 Drive 21.4 6 258 110 3.08 3.215 19.44 1 0 3 1 3.5
## 5 Hornet Sportabout 18.7 8 360 175 3.15 3.440 17.02 0 0 3 2 3.1
## 6 Valiant 18.1 6 225 105 2.76 3.460 20.22 1 0 3 1 3.0
## # lp100k
## 1 29.4
## 2 29.4
## 3 27.0
## 4 28.6
## 5 32.3
## 6 33.3
```

La funzione `summarise()` è particolarmente utile assieme a `group_by()`:

```
starwars %>%
  group_by(species) %>%
  summarise(
    n = n(),
    mass = round(mean(mass, na.rm=T), 1),
```

```

  "max height" = max(height, na.rm=T),
  "min height" = min(height, na.rm=T)
) %>%
filter(
  n > 1,
  mass > 50
)

## # A tibble: 8 x 5
##   species     n  mass `max height` `min height`
##   <chr>     <int> <dbl>        <int>        <int>
## 1 Droid       6   69.8        200         96
## 2 Gungan      3    74          224        196
## 3 Human      35   82.8        202        150
## 4 Kaminoan    2    88          229        213
## 5 Mirialan   2   53.1        170        166
## 6 Twi'lek     2    55          180        178
## 7 Wookiee    2  124          234        228
## 8 Zabrak     2    80          175        171

```

Sono infine particolarmente utili le funzioni per **combinare tabelle di dati** secondo il paradigma relazionale:

```

persons <- tribble(
  ~name, ~surname, ~role,
  "Paolo", "Bosetti", 1,
  "John", "Smith", 2,
  "Phil", "Cameron", 1,
  "Eddy", "Hunt", 3,
  "Sebastian", "Hauer", 3
)

roles <- tribble(
  ~id, ~role,
  1, "attack",
  2, "play",
  3, "defense"
)

team <- left_join(persons, roles, by=c("role"="id"))
team %>% filter(role.y=="attack")

## # A tibble: 2 x 4
##   name   surname  role role.y
##   <chr>  <chr>    <dbl> <chr>
## 1 Paolo  Bosetti     1 attack
## 2 Phil   Cameron     1 attack

# Equivalente a:
team[team$role.y=="attack",]

## # A tibble: 2 x 4
##   name   surname  role role.y
##   <chr>  <chr>    <dbl> <chr>
## 1 Paolo  Bosetti     1 attack
## 2 Phil   Cameron     1 attack

```

La libreria `purrr` mette inoltre a disposizione funzioni di mapping più comode rispetto alle equivalenti di R base (`lapply` e simili), ed hanno il vantaggio di ritornare sempre lo stesso tipo di oggetto, a differenza delle funzioni base, e quindi di essere più adatte alla programmazione:

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
map(df, mean) %>% class
## [1] "list"
map_dbl(df, mean) %>% class
## [1] "numeric"
map_lgl(df$a, function(x) x>=0)
## [1] TRUE FALSE TRUE TRUE TRUE TRUE TRUE TRUE FALSE FALSE
```

Esistono anche versioni per funzioni a due parametri e a n parametri:

```
mu <- list(1, 3, -10)
sigma <- list(1, 0.2, 2)
map2(mu, sigma, rnorm, n=5)

## [[1]]
## [1] 0.5089688 -1.3091689 2.0057385 0.2907992 0.3119914
##
## [[2]]
## [1] 3.205114 2.943045 2.755856 3.036261 2.972222
##
## [[3]]
## [1] -9.988472 -9.229439 -10.741320 -8.711247 -10.440973

n <- 1:3
list(n=n, mean=mu, sd=sigma) %>%
  pmap(rnorm) %>%
  str

## List of 3
## $ : num 1.33
## $ : num [1:2] 3.22 3.09
## $ : num [1:3] -10.65 -7.7 -8.01
```

Se tutti gli argomenti hanno la stessa lunghezza si può usare anche un data frame:

```
params <- tribble(
  ~mean, ~sd, ~n,
  5,      1,   1,
  10,     5,   2,
  -3,    10,   3
)
params %>%
  pmap(rnorm) %>%
  str

## List of 3
## $ : num 5.55
```

```
## $ : num [1:2] 11.19 6.86
## $ : num [1:3] 10.6 -9 18.9
```

1.4 Stringhe

La manipolazione delle stringhe è tradizionalmente più laboriosa in R che in altri linguaggi dinamici (come Ruby o Python). Le librerie `glue` e `stringr` semplificano due classi di operazioni: la prima l'interpolazione di stringhe, la seconda il matching.

In R base non c'è un meccanismo per l'interpolazione di stringhe, ma esse vanno combinate con ripetute chiamate a `cat()`. `glue` invece introduce l'interpolazione:

```
library(glue)
glue("Questo istante: {date()}. Autore: {author}", author="Paolo Bosetti")
## Questo istante: Fri Jan 21 09:53:48 2022. Autore: Paolo Bosetti
```

La versione `glue_data()` è fatta per accettare *pipe*:

```
head(mtcars) %>% glue_data("{rownames(.)} has {hp} hp")
## Mazda RX4 has 110 hp
## Mazda RX4 Wag has 110 hp
## Datsun 710 has 93 hp
## Hornet 4 Drive has 110 hp
## Hornet Sportabout has 175 hp
## Valiant has 105 hp
```

La libreria `stringr` offre numerose funzioni, tra cui le più utili sono quelle per spezzare, per cambiare *case* (maiusscolo/minuscolo) e per effettuare il *pattern matching* con espressioni regolari. Per queste ultime, è utile installare l'add-in di RStudio `RegExplain`:

```
devtools::install_github("gadenbuie/regexplain")
library(stringr)
str_split(c("a,b", "c,d,e"), ",")
## [[1]]
## [1] "a" "b"
##
## [[2]]
## [1] "c" "d" "e"

c("Paolo Bosetti", "Bosetti, Paolo") %>%
  str_match_all("(\\w+)[,\\s]+(\\w+)")
## [[1]]
## [,1]      [,2]      [,3]
## [1,] "Paolo" "Bosetti" "Paolo"
##
## [[2]]
## [,1]      [,2]      [,3]
## [1,] "Bosetti" "Paolo" "Bosetti"
```

1.5 Grafici base

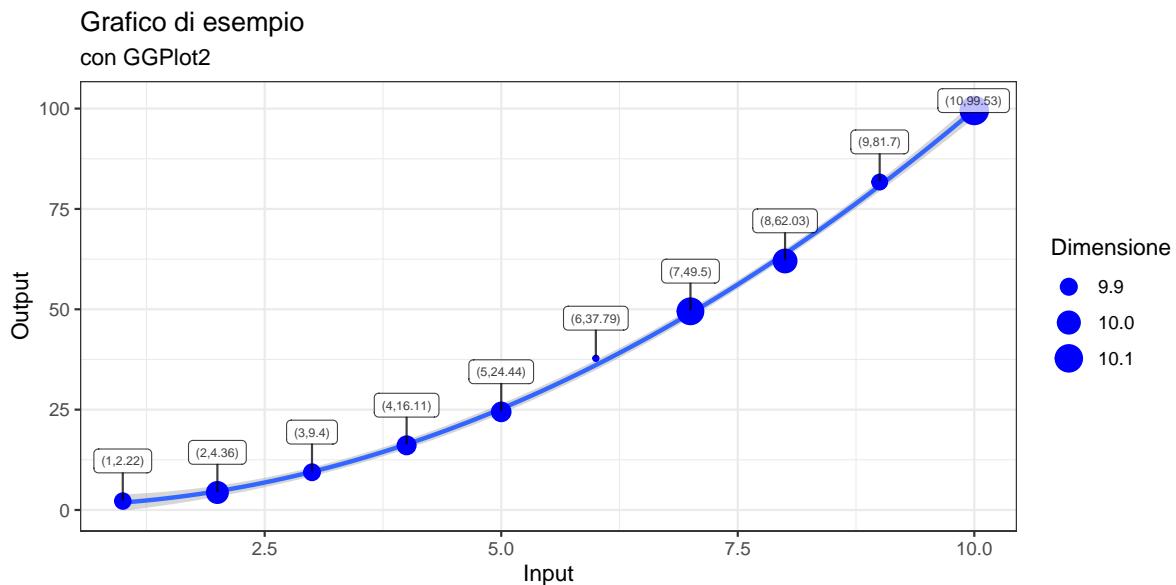
Forse una delle innovazioni più importanti di Tidyverse è la nuova interfaccia di plotting, che è fornita dalla libreria `ggplot2`, dove la doppia `g` sta per *Grammar of Graphics*, secondo l'idea di comporre un grafico accostando funzioni secondo una grammatica base che consente di tenere separati i dati dagli algoritmi e dall'estetica.

Una interessante caratteristica di `ggplot2` è che consente di adottare dei temi (personalizzabili) in modo da controllare l'aspetto estetico dei grafici realizzati. I temi vengono caricati così:

```
theme_set(theme_bw())
```

Un GGPlot viene creato inizialmente con la funzione `ggplot()` che richiede la struttura dati (`tibble` o `dataframe`) e gli *aesthetics*, cioè un comando che specifica *cosa* deve essere visualizzato nel grafico (funzione `aes()`). Quest'unico comando tuttavia produce esclusivamente un grafico vuoto: per riempirlo è necessario aggiungere dei *layer*, **scommendo** al `ggplot` opportuni comandi:

```
library(ggrepel)
ggplot(a, aes(x=In, y=Out, label=glue("({In},{y})", y=round(Out,2)))) +
  geom_smooth(formula = y~poly(x,2), method="lm") +
  geom_point(aes(size=Size), color="blue") +
  geom_label_repel(alpha=3/4, nudge_x=0, nudge_y=10, size=2) +
  labs(title="Grafico di esempio", subtitle = "con GGPlot2") +
  xlab("Input") +
  ylab("Output") +
  labs(size="Dimensione")
```

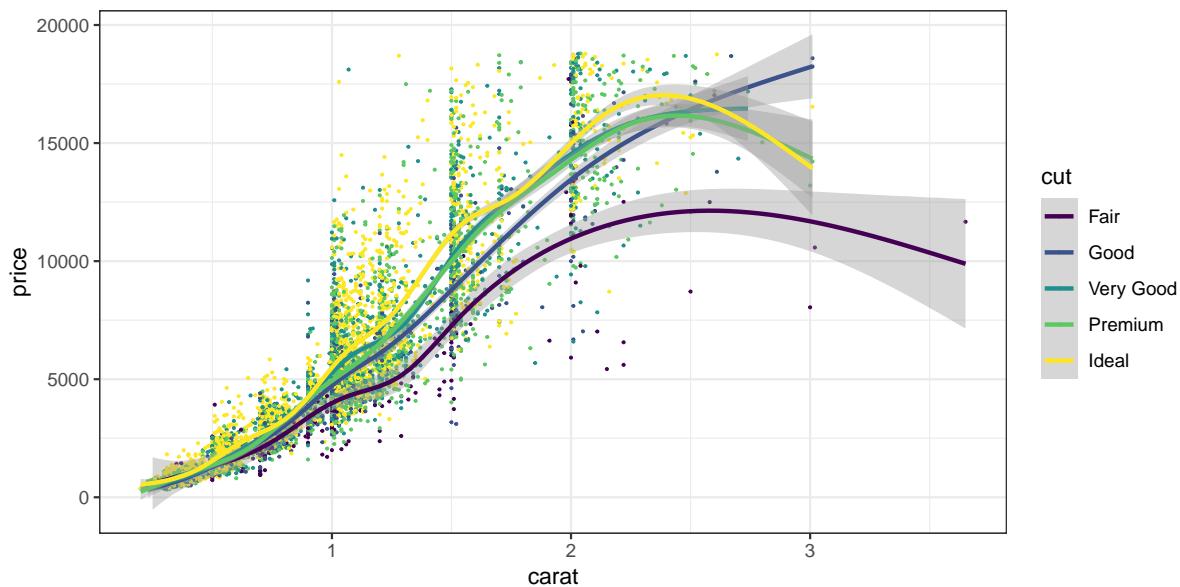


1.5.1 Aesthetics

È importante comprendere il ruolo degli *aesthetics*: essi definiscono quali *variabili* nei dati originali vengono mappati sui due assi e sui vari indicatori del grafico (colori, dimensioni, forma, riempimento, ecc.). A seconda dei casi `aes()` può essere indicata solo in `ggplot()` oppure anche nei comandi `geom_*`() successivi:

```
gp <- diamonds %>%
  slice_sample(prop=0.2) %>%
  ggplot(aes(x=carat, y=price, color=cut)) +
  geom_point(size=1/4) +
  geom_smooth()
print(gp)

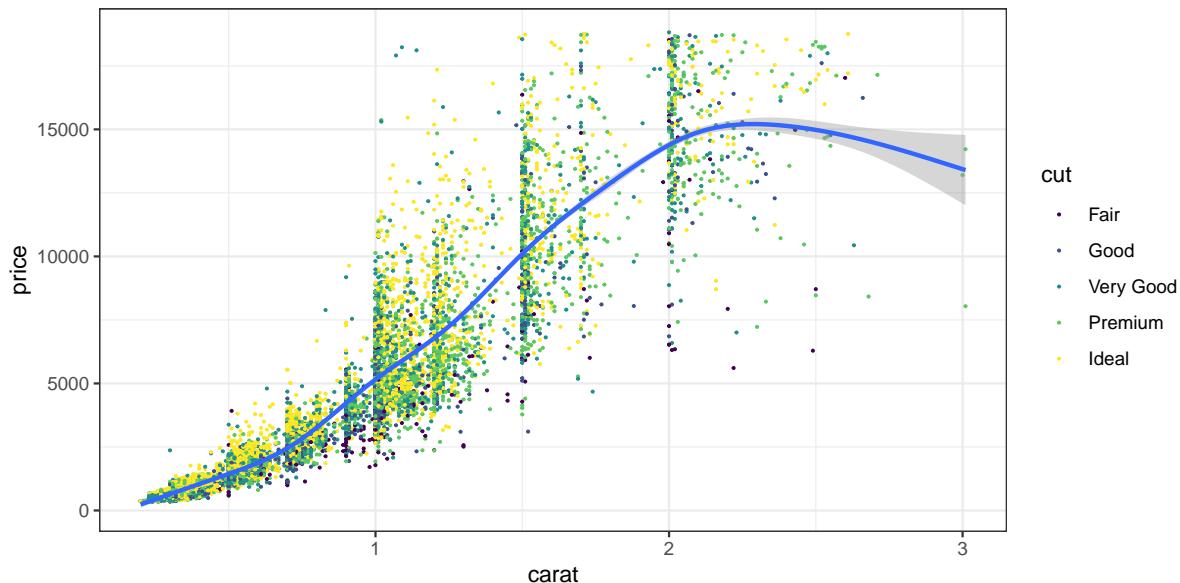
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



Come si vede, siccome l'estetica include i tre parametri, se non si specifica nessuna estetica per `geom_smooth()` essa viene applicata per ciascun raggruppamento previsto dall'estetica `color`. Se invece volessimo un'unica linea di tendenza dovremmo specificare un'estetica apposita e **separata** per `geom_point()` e per `geom_smooth()`:

```
diamonds %>%
  slice_sample(prop=0.2) %>%
  ggplot() +
  geom_point(aes(x=carat, y=price, color=cut), size=1/4) +
  geom_smooth(aes(x=carat, y=price))

## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```

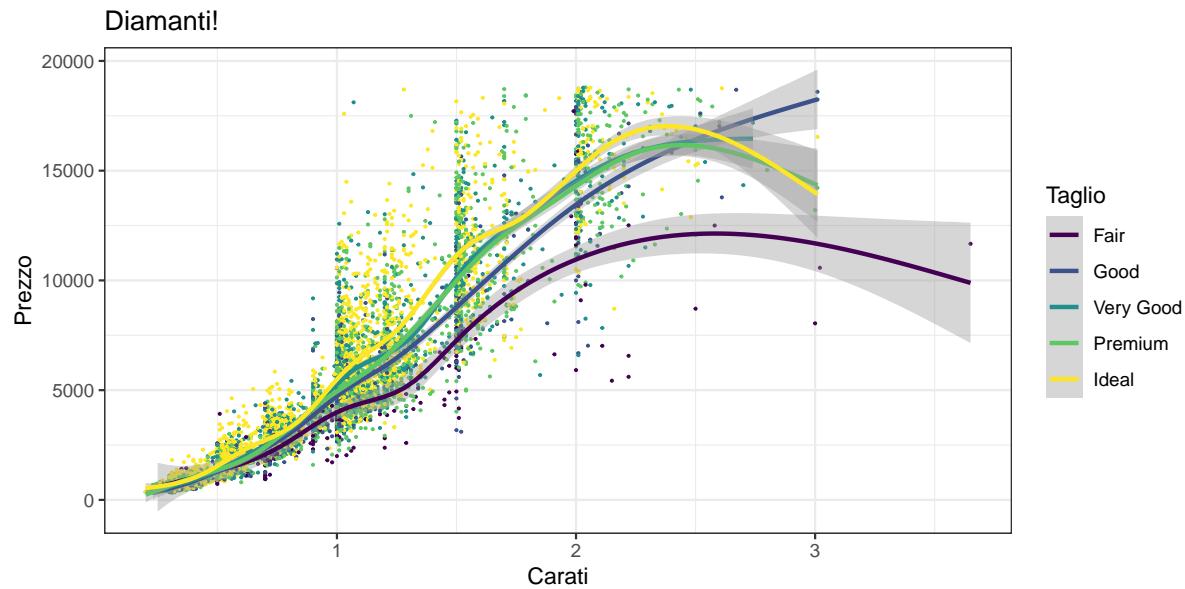


Si noti l'uso della funzione `slice_sample()` (in `dplyr`) per estrarre a caso solo il 20% dei dati, in modo da velocizzare la creazione del grafico.

Le etichette di testo possono essere modificate con il verbo `labs()`:

```
gp + labs(title="Diamanti!", x="Carati", y="Prezzo", color="Taglio")
```

```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```

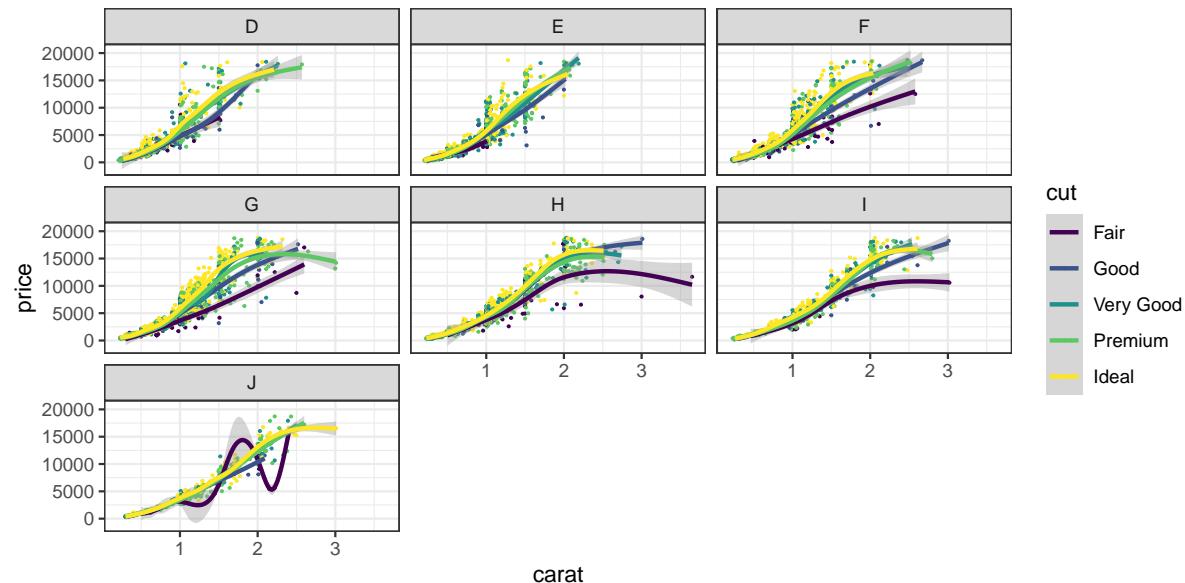


1.5.2 Facets

Altri termini di raggruppamento possono essere inclusi variando ad esempio `size`, `shape`, `fill` dei punti del grafico, oppure creando matrici di grafici in cui le variabili da utilizzare sulle righe e sulle colonne della matrice vanno indicate mediante una `formula` di tipo `row ~ column`, omettendo eventualmente la prima o la seconda:

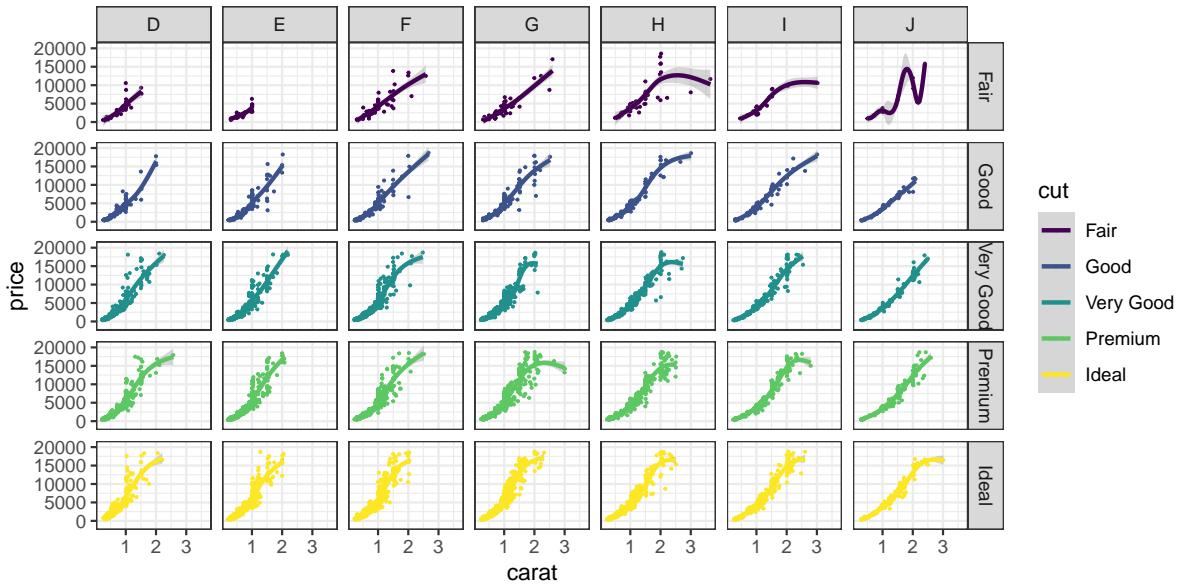
```
gp + facet_wrap(~color)
```

```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



```
gp + facet_grid(cut~color)
```

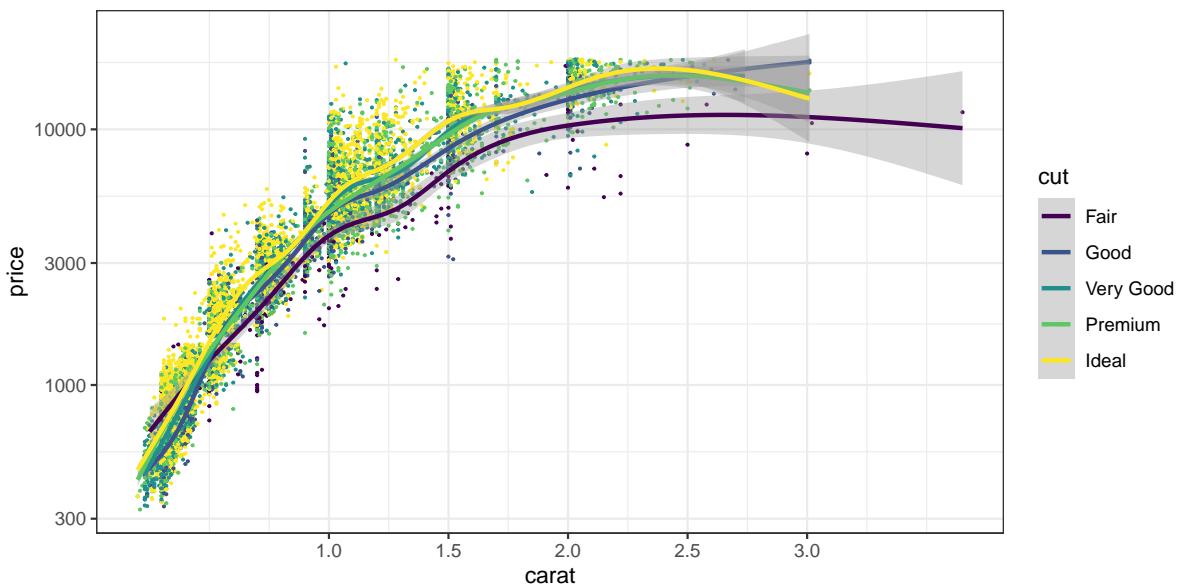
```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



1.5.3 Scale

I verbi ggplot che cominciano con `scale_` possono essere utilizzati per manipolare le scale degli assi (sia gli assi `x` e `y` che gli assi virtuali). Ad esempio è possibile cambiare la spaziatura delle etichette degli assi (`scale_*_continuous(breaks=c(...))`) che il tipo di scala (`scale_*_log10()` piuttosto che `scale_*_reverse()`):

```
gp + scale_x_continuous(breaks = seq(1, 3, 0.5)) +
  scale_y_log10()
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



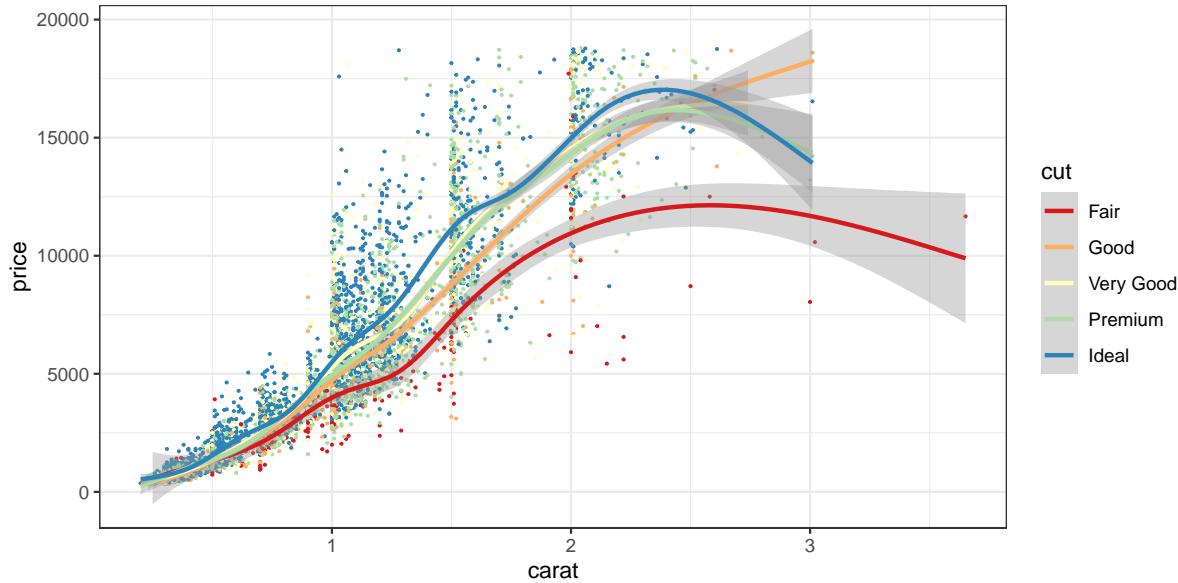
1.5.4 Palette

Un particolare tipo di scala è quella dei colori. Le palette di colori disponibili sono visualizzabili con il comando (libreria RColorBrewer):

```
display.brewer.all()
```

Ad esempio:

```
gp +
  scale_color_brewer(palette = "Spectral")
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



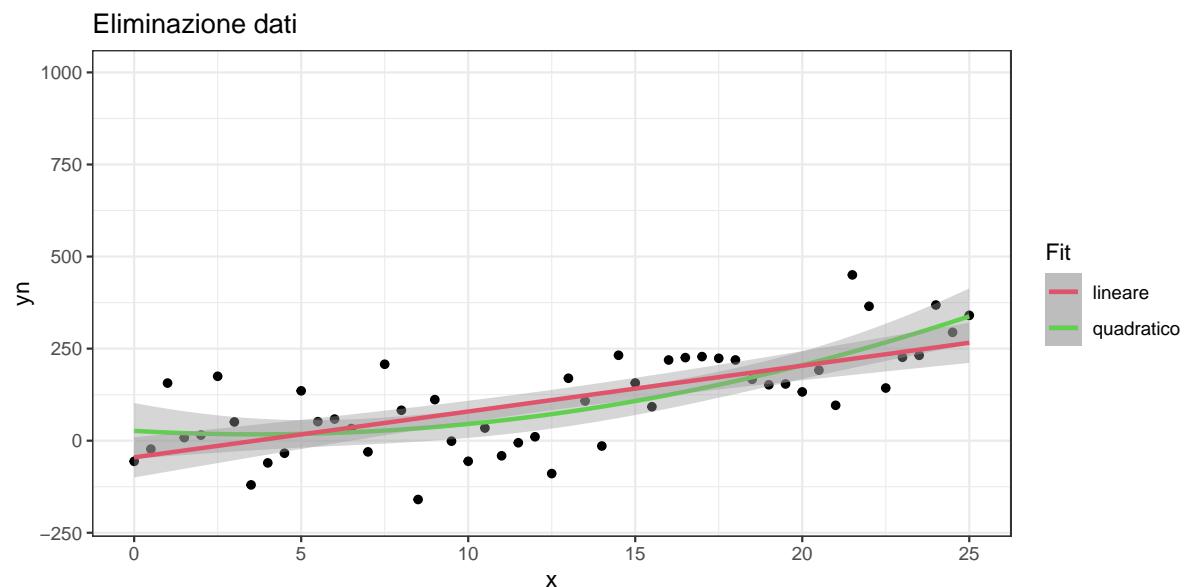
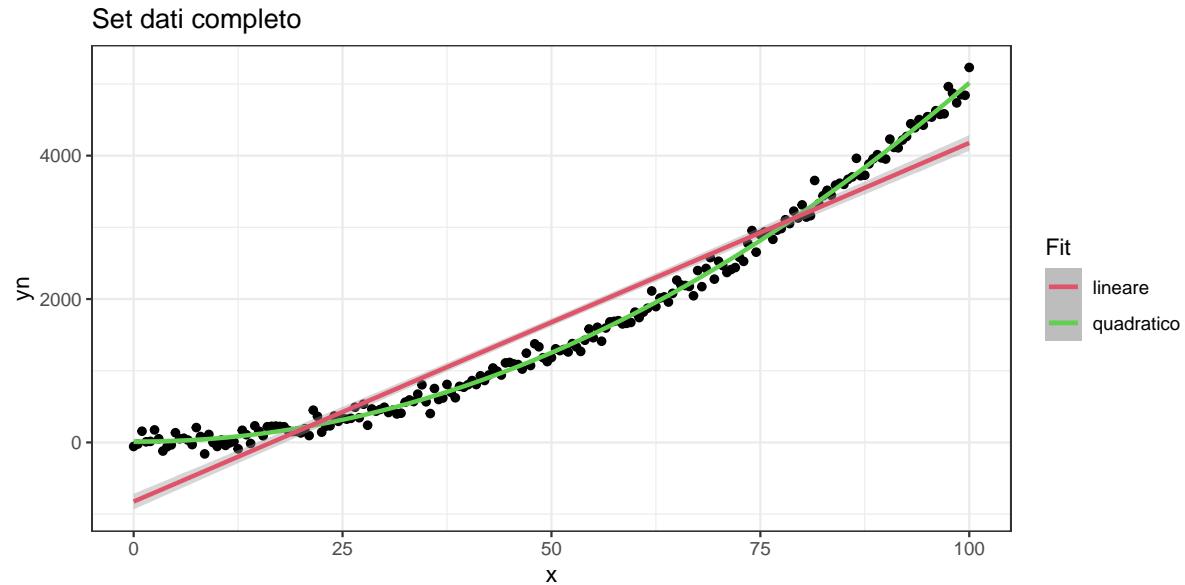
1.5.5 Limiti degli assi

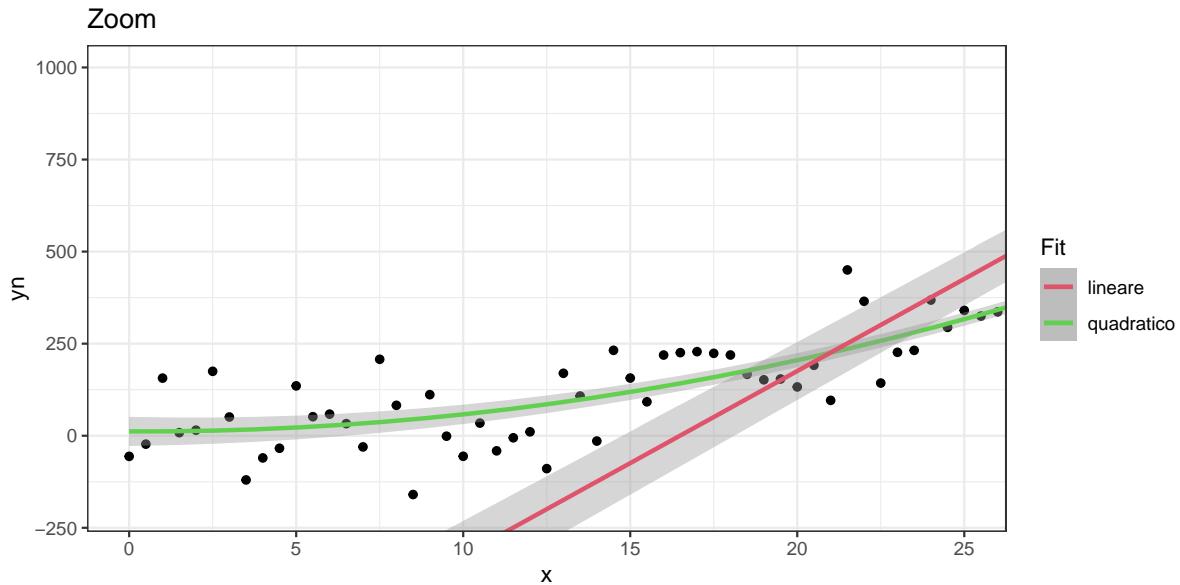
In GGplot2 ci sono due modi per modificare i limiti degli assi:

1. *rimuovendo* i dati esterni ad un intervallo: questo modifica il set di dati iniziali, e quindi cambia il comportamento di funzioni come `geom_smooth()`
2. *zoomando* sul grafico, mantenendo intatto il set di dati.

```
set.seed(123)
data <- tibble(
  x=seq(0,100,0.5),
  y=0.5*x^2+0.1*x,
  yn=y+rnorm(length(x), 0, 100)
)

(gp <- data %>% ggplot(aes(x=x, y=yn)) +
  geom_point() +
  geom_smooth(method="lm", formula=y~poly(x, 2)+x, aes(color="quadratico")) +
  geom_smooth(method="lm", formula=y~x, aes(color="lineare")) +
  scale_color_manual(name="Fit", values=c(2, 3)) +
  ggtitle("Set dati completo"))
```





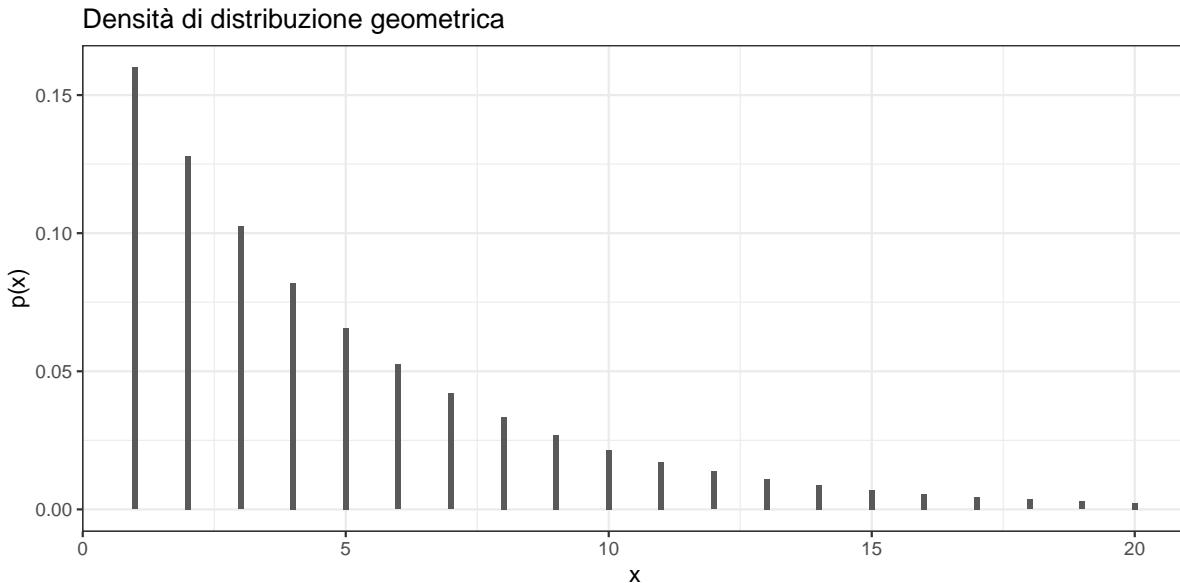
2 Esempi

In questo capitolo riprendiamo alcuni concetti espressi nelle parti precedenti e ricreiamo gli stessi grafici in GGplot2.

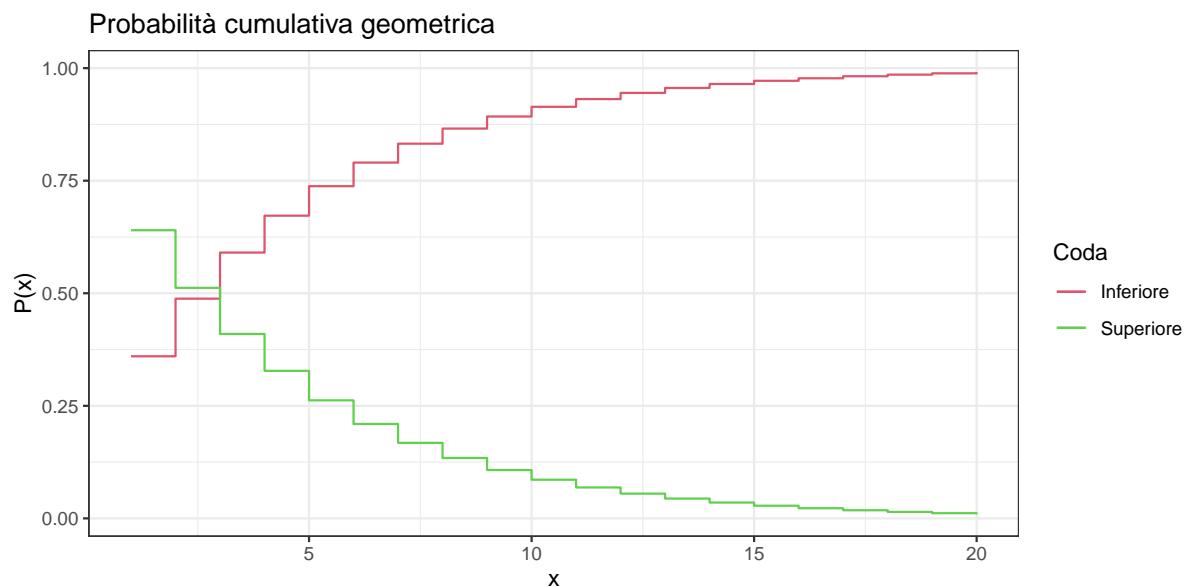
2.1 Distribuzioni

2.1.1 Distribuzioni discrete

```
df <- tibble(
  x=1:20,
  d=dgeom(x, prob=0.2),
  pl=pgeom(x, prob=0.2, lower.tail = T),
  pu=pgeom(x, prob=0.2, lower.tail = F)
)
ggplot(df, aes(x=x, y=d)) +
  geom_col(width=0.1) +
  labs(title="Densità di distribuzione geometrica", y="p(x)")
```

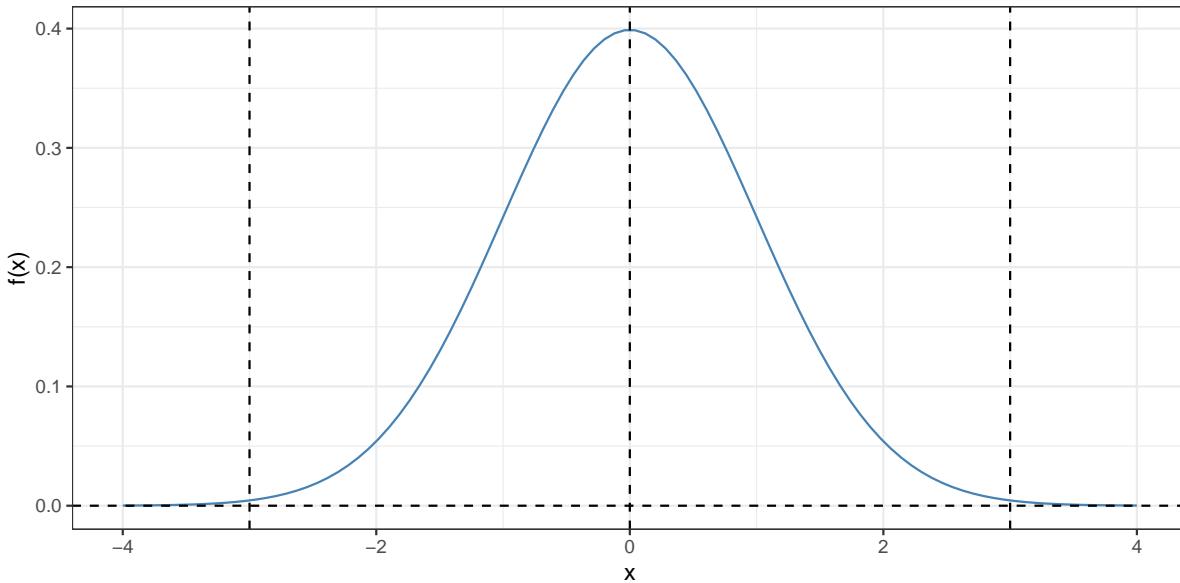


```
ggplot(df) +
  geom_step(aes(x=x, y=pl, col="Inferiore")) +
  geom_step(aes(x=x, y=pu, col="Superiore")) +
  scale_color_manual(name="Coda", values=c(2,3)) +
  labs(title="Probabilità cumulativa geometrica", y="P(x)")
```



2.1.2 Distribuzioni continue

```
df <- tibble(
  x=seq(-4,4,length.out=100),
  f=dnorm(x, 0, 1)
)
df %>% ggplot(aes(x=x, y=f)) +
  geom_line(color="steelblue") +
  geom_hline(yintercept=0, linetype=2) +
  geom_vline(xintercept=c(-3, 0, 3), linetype=2) +
  labs("Standard normal density function", y="f(x)")
```

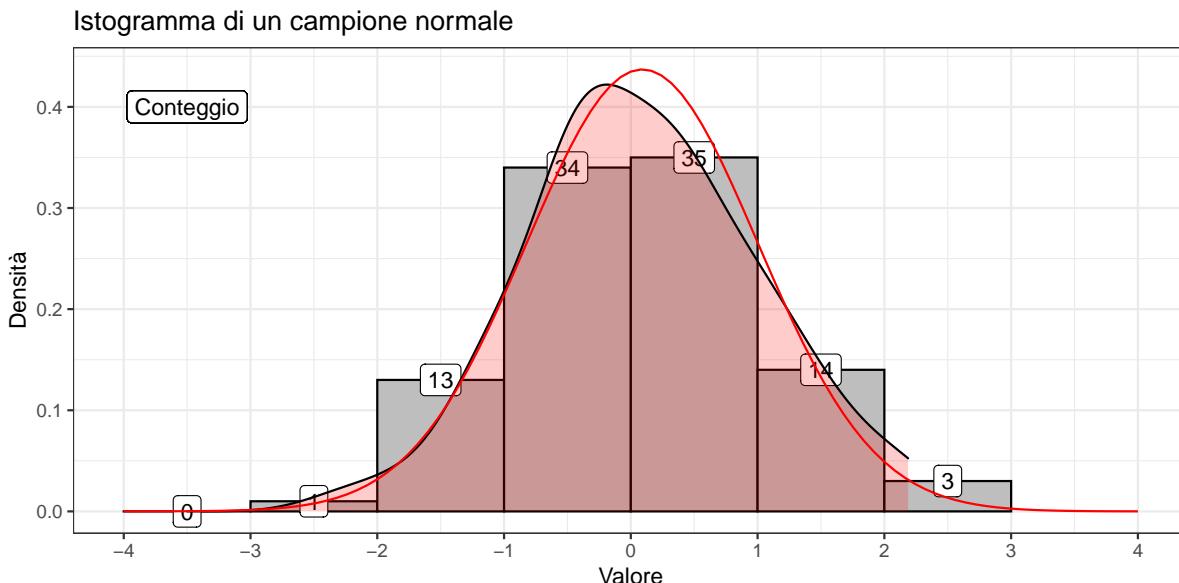


2.1.3 Istogrammi e QQ-plot

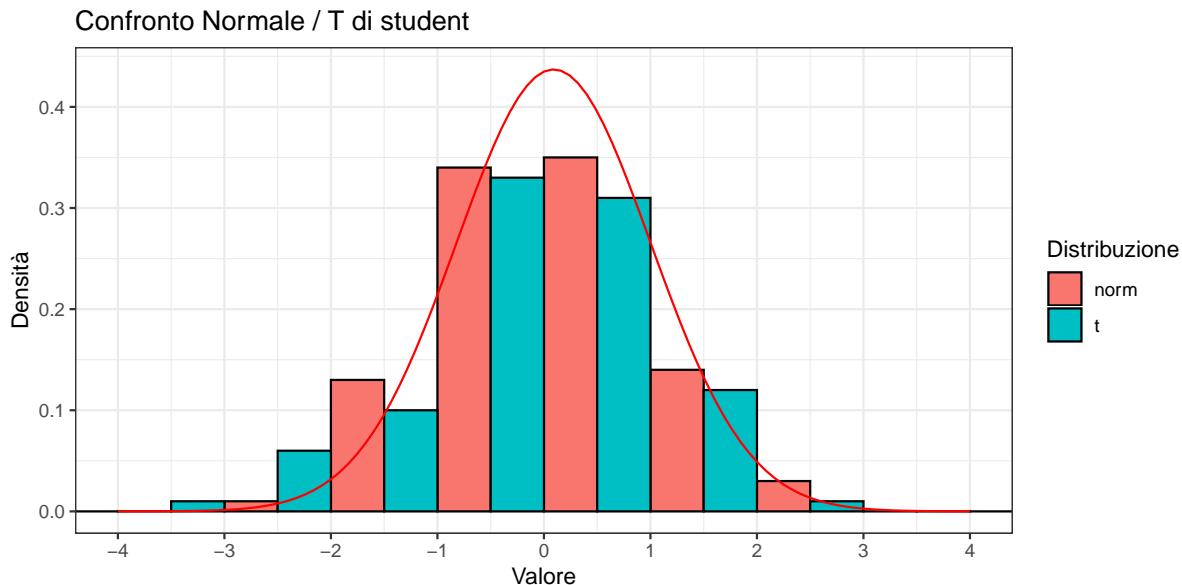
Ottenere istogrammi simili a quelli prodotti dalla funzione `hist()` non è immediato, a causa di alcune differenze nei default. In particolare, `geom_histogram()`:

- assume sempre un default di 30 bins, mentre `hist()` li calcola automaticamente con la formula di Sturges, $k = \lceil \log_2(n) \rceil + 1$, fornita da `nclass.Sturges()`;
- ciascuna barra è centrata sul suo intervallo, mentre in `hist()` va da estremo sinistro a estremo destro

```
set.seed(123)
df <- (data <- tibble(
  x=1:100,
  norm=rnorm(length(x)),
  t=rt(length(x), 3)
))
df %>% ggplot(aes(x=norm)) +
  geom_histogram(aes(y=..density..),
    fill="gray",
    color="black",
    binwidth=1,
    boundary=0
  ) +
  geom_label(aes(y=..density.., label=..count..),
    stat="bin",
    binwidth=1,
    boundary=0) +
  geom_label(aes(x=-3.5, y=0.4, label="Conteggio")) +
  geom_density(fill="red", alpha=0.2) +
  geom_function(fun=dnorm,
    args=list(
      mean=mean(df$norm),
      sd=sd(df$norm)),
    color="red",
    xlim=c(-4,4)) +
  scale_x_continuous(breaks=-4:4) +
  labs(x="Valore", y="Densità", title="Istogramma di un campione normale")
```



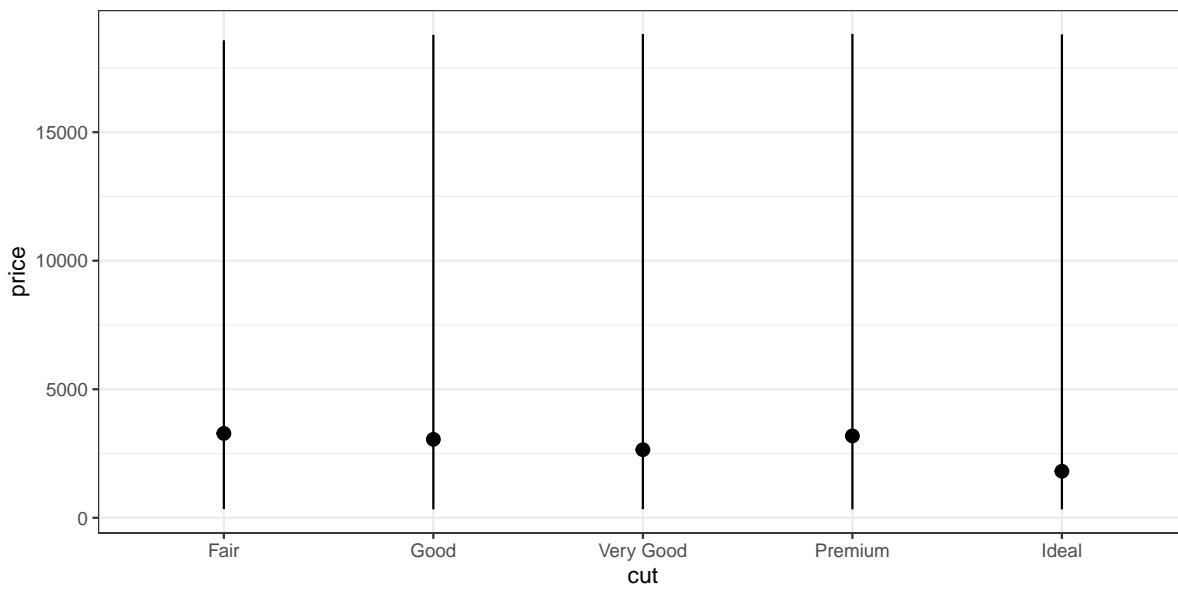
```
df %>%
  pivot_longer(c("norm", "t"), names_to = "type", values_to = "value") %>%
  ggplot(aes(x=value)) +
  geom_histogram(aes(y=..density.., fill=type),
                 color="black",
                 binwidth=1,
                 boundary=0,
                 position="dodge"
  ) +
  geom_function(fun=dnorm,
                args=list(
                  mean=mean(df$norm),
                  sd=sd(df$norm)
                ),
                color="red",
                xlim=c(-4,4)) +
  scale_x_continuous(breaks=-4:4) +
  coord_cartesian(xlim=c(-4,4)) +
  labs(x="Valore", y="Densità",
       fill="Distribuzione",
       title="Confronto Normale / T di student")
```



Nota: in `aes()` la notazione `..density..` sta a indicare “applica la *statistica density* ai dati in ingresso”. Le statistiche disponibili sono elencate nella sezione *Computed variables* dell’help delle funzioni `geom_*`.

A volte si desidera una raffigurazione comparativa:

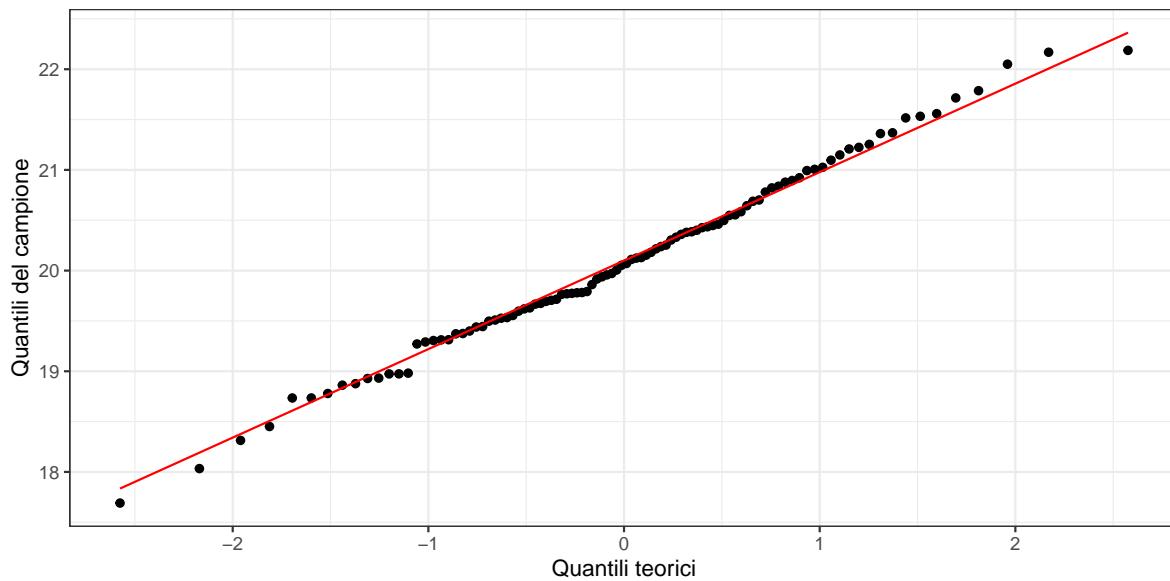
```
ggplot(data=diamonds) +
  stat_summary(mapping=aes(x=cut, y=price),
              fun.min=min,
              fun.max=max,
              fun=median)
```



Infine, la distribuzione può essere analizzata mediante i diagrammi quantile-quantile:

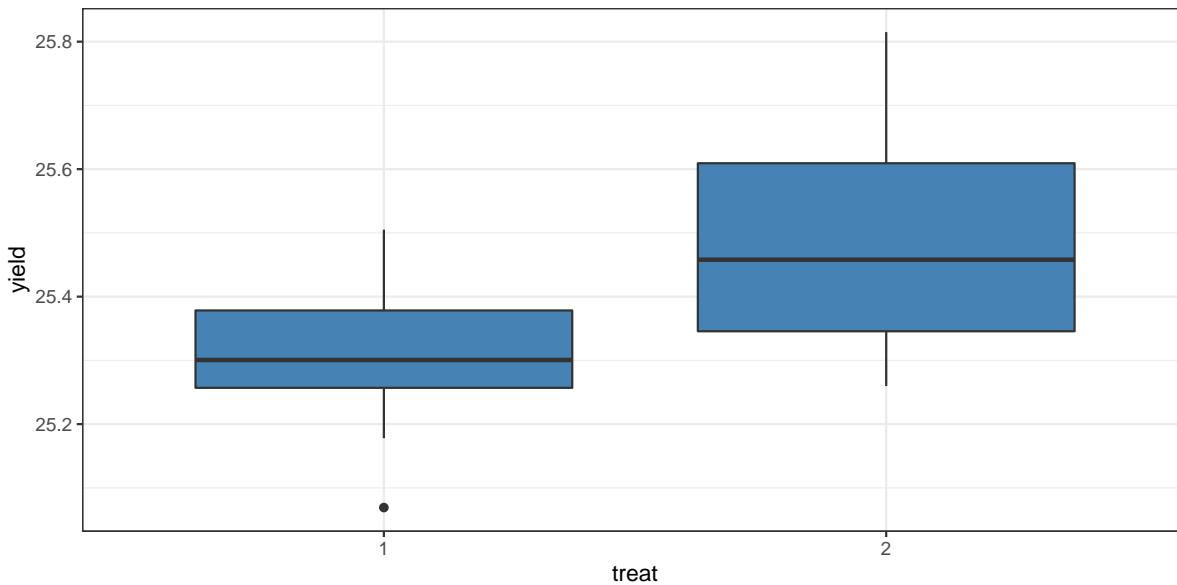
```
set.seed(123)
tibble(
  x=rnorm(100, mean=20)
) %>%
  ggplot(aes(sample=x)) +
```

```
geom_qq() +
  geom_qq_line(color="red") +
  xlab("Quantili teorici") +
  ylab("Quantili del campione")
```



2.1.4 Boxplot

```
df <- read_table(mydata("twosample.dat"), col_types=cols(
  treat=col_factor(),
  yield=col_double()))
df %>% ggplot(aes(x=treat, y=yield)) +
  geom_boxplot(fill="steelblue")
```

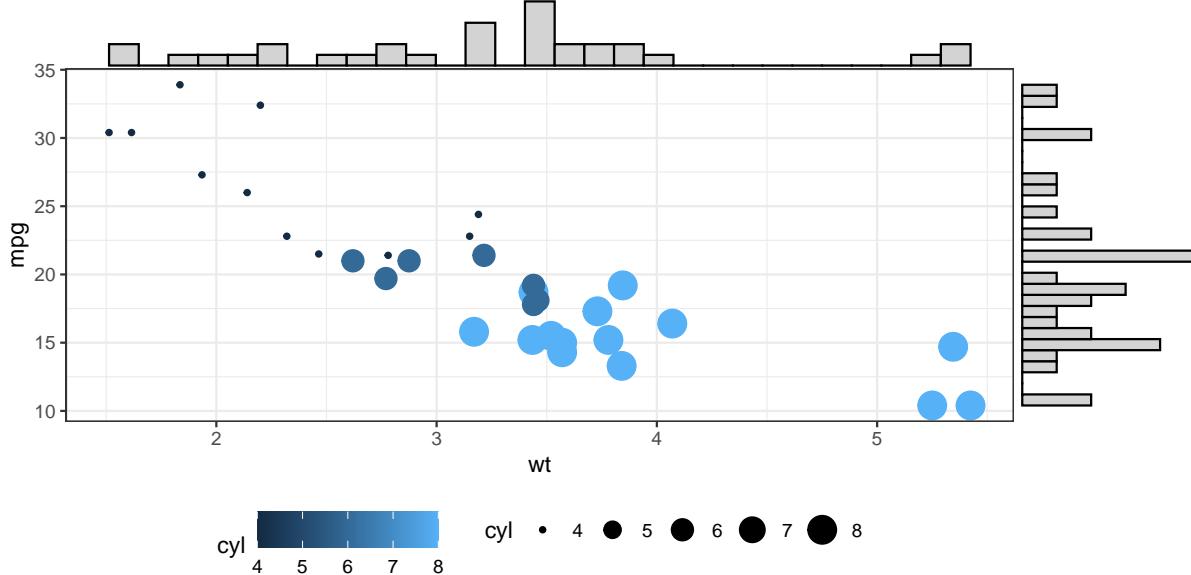


2.1.5 Istogrammi marginali

```
df <- tibble(mtcars)
gp <- df %>% ggplot(aes(x=wt, y=mpg, color=cyl, size=cyl)) +
```

```
geom_point() +
theme(legend.position="bottom")

gp %>% ggMarginal(type="histogram", fill="lightgray")
```



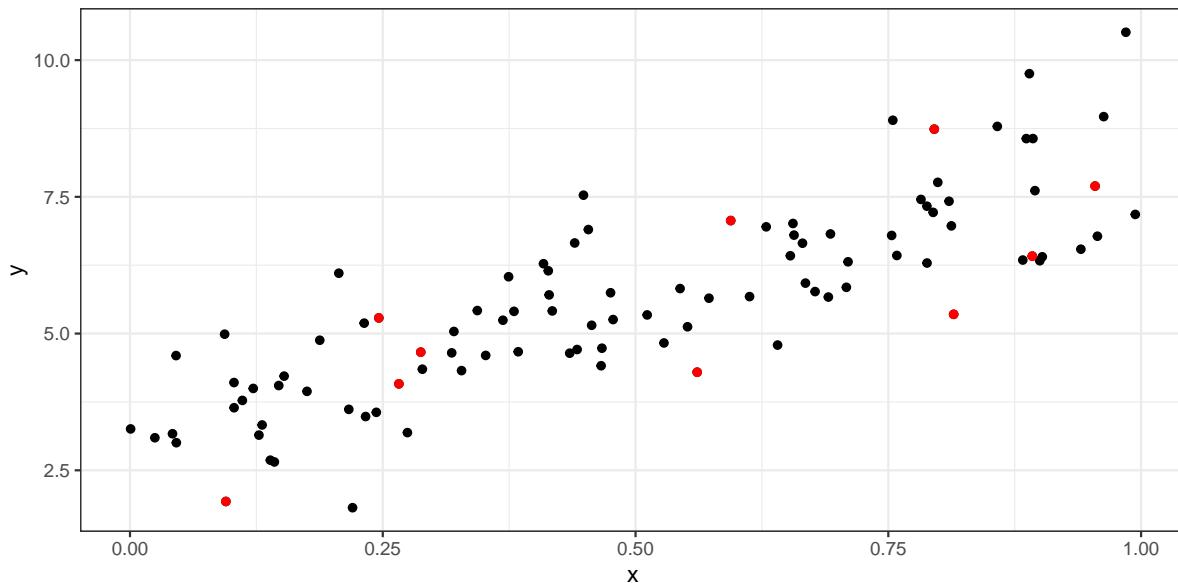
2.2 Modelli

La libreria `modelr` è utile nella costruzione e analisi di modelli lineari.

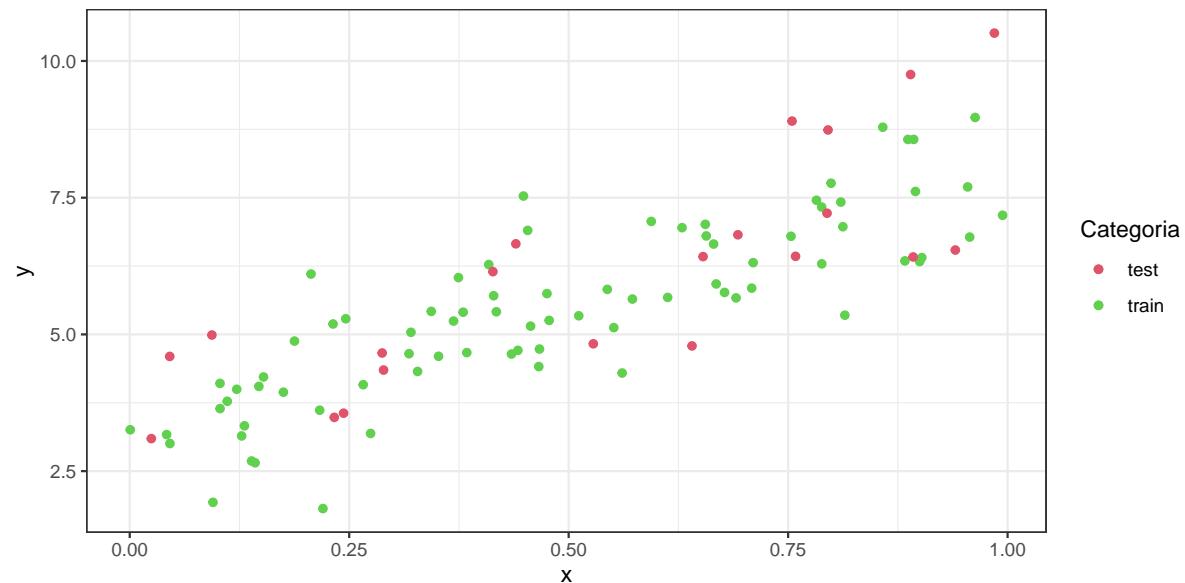
```
set.seed(123)
df <- tibble::tibble(
  x = sort(runif(100)),
  y = 5 * x + 0.5 * x ^ 2 + 3 + rnorm(length(x))
)
```

Il *subsetting* viene effettuato mediante le funzioni `resample()` e `resample_partition()`. Esse ritornano oggetti `resample`, che contengono solo la lista degli indici campionati e un *puntatore* ai dati originali; in questo modo, ri-campionamenti successivi su larghe quantità di dati sono più efficienti:

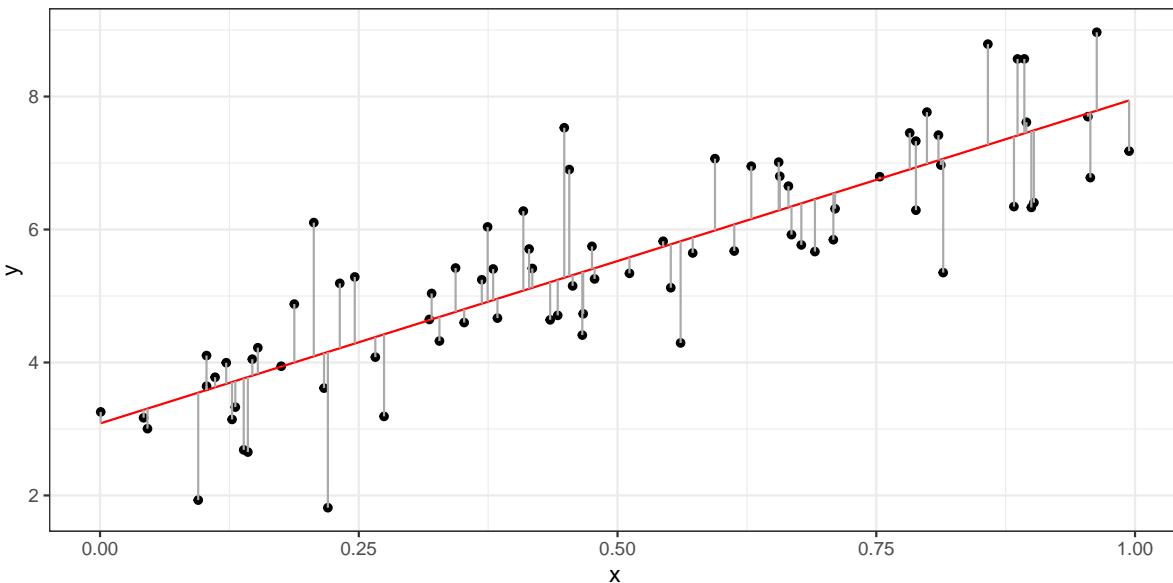
```
df1 <- resample(df, sample(seq_along(df$x), 10)) # 10 elementi casuali
ggplot(df, aes(x=x, y=y)) + geom_point() +
  geom_point(data=as_tibble(df1), mapping=aes(x=x, y=y), color="red")
```



```
dfp <- resample_partition(df, c(train=0.8, test=0.2))
ggplot(as_tibble(dfp$train)) +
  geom_point(aes(x=x, y=y, color="train")) +
  geom_point(aes(x=x, y=y, color="test"), as_tibble(dfp$test)) +
  scale_color_manual(name="Categoria", values=c(2,3))
```



```
df.train <- as_tibble(dfp$train)
df.lm <- lm(y~x, data=df.train)
df.train %>% add_predictions(df.lm)
df.train %>% add_residuals(df.lm)
ggplot(df.train) +
  geom_point(aes(x=x, y=y)) +
  geom_line(aes(x=x, y=pred), col="red") +
  geom_linerange(aes(x=x, ymin=y, ymax=y-resid), color=gray(2/3))
```



2.3 Serie temporali

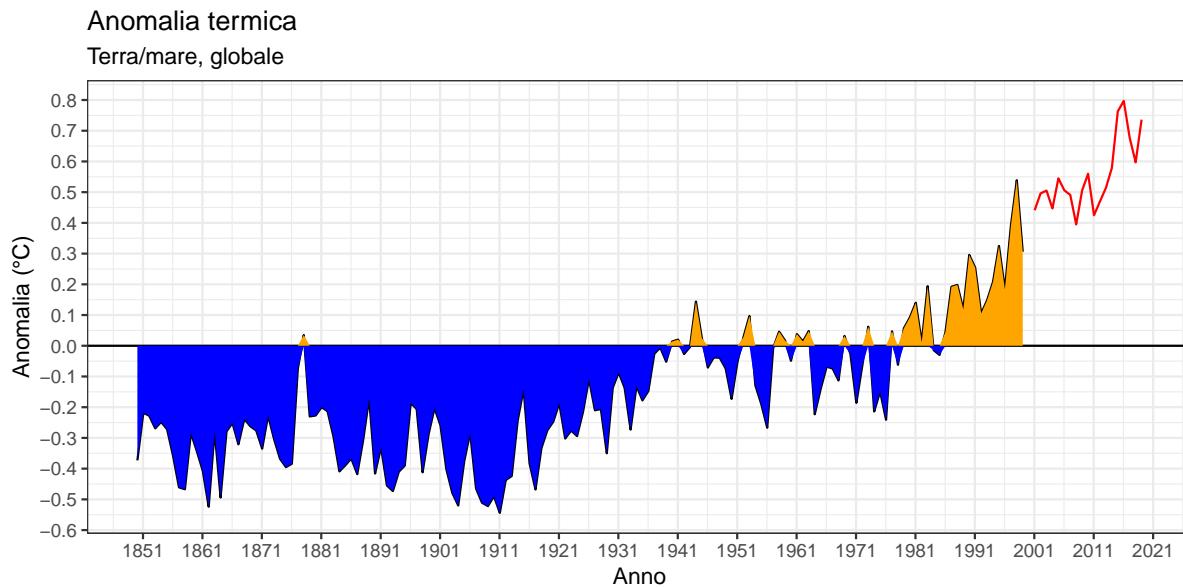
GGPlot2 coesiste abbastanza bene con gli oggetti `xts`: le funzioni della libreria convertono in `tibble` ogni oggetto `xts` chiamando automaticamente la funzione `fortify()`. Vediamo un esempio.

```
data <- read.csv(mydata<-"temperature-anomaly.csv")
data <- data[data$Entity=="Global",]
t.global <- xts(data$Median.temp,
                  order.by=as.Date(as.character(data$Year), format="%Y"),
                  frequency = 1)
```

Realizziamo un grafico separando visivamente le anomalie negative da quelle positive, e indicando in maniera differente i valori successivi al primo Gennaio 2000:

```
x1 <- t.global["/1999-12-31"]
p1 <- autoplot(x1) +
  geom_hline(yintercept = 0) +
  geom_area(aes(x=index(x1), y=ifelse(x1<0, x1, 0)), fill="blue") +
  geom_area(aes(x=index(x1), y=ifelse(x1>0, x1, 0)), fill="orange") +
  labs(title="Anomalia termica", subtitle = "Terra/mare, globale") +
  xlab("Anno") +
  ylab("Anomalia (°C)")

x2 <- t.global["2000-1-1/"]
p1 + geom_line(data=x2, aes(Index, x2), color="red") +
  #scale_x_continuous(breaks=seq(start(x1), end(x2), by="20 years")) +
  scale_x_date(breaks="10 years", date_labels="%Y") +
  scale_y_continuous(
    breaks=seq(round(min(t.global), 0.1), round(max(t.global), 0.1), by=0.1)
  )
```



Si noti che gli oggetti `xts` vengono automaticamente convertiti in `data.frame` invocando automaticamente la funzione `fortify()`. Quest'ultima crea la colonna dei tempi con il nome `Index`:

```
x2 %>% fortify %>% str
## 'data.frame':    19 obs. of  2 variables:
##   $ Index: Date, format: "2001-01-21" "2002-01-21" ...
##   $ .    : num  0.441 0.496 0.505 0.447 0.545 0.506 0.491 0.395 0.506 0.56 ...
```

Per questo motivo, si usa l'estetica `aes(x=Index, y=x2)`

Ricreiamo ora lo stesso grafico di predizione dei passeggeri delle compagnie aeree visto in Parte 3:

```
passl <- AirPassengers %>% ts_xts

gp <- passl %>%
  ggplot(aes(x=Index, y=value)) +
  geom_line()

(fit <- auto.arima(ts_ts(passl), lambda="auto"))

## Series: ts_ts(passl)
## ARIMA(0,1,1)(0,1,1)[12]
## Box Cox transformation: lambda= -0.2947046
##
## Coefficients:
##             ma1      sma1
##           -0.4355  -0.5847
## s.e.     0.0908   0.0725
##
## sigma^2 estimated as 5.856e-05: log likelihood=451.59
## AIC=-897.18  AICc=-896.99  BIC=-888.55

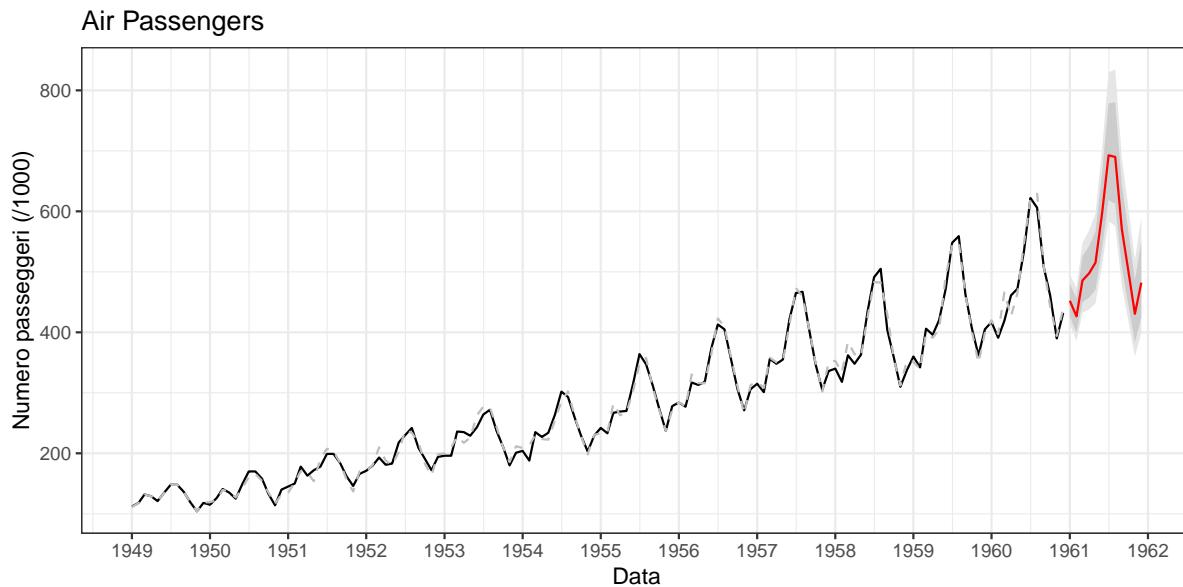
pred <- forecast(fit, h=12)

gp +
  geom_line(aes(x=Index, y=value),
            data=ts_xts(pred$fit), color="gray", lty=2) +
  geom_ribbon(aes(x=Index,
```

```

ymin=ts_xts(pred$lower)[,2],
ymax=ts_xts(pred$upper)[,2]),
data=ts_xts(pred$mean), fill=gray(0.9)) +
geom_ribbon(aes(x=Index,
ymin=ts_xts(pred$lower)[,1],
ymax=ts_xts(pred$upper)[,1]),
data=ts_xts(pred$mean), fill=gray(0.8)) +
geom_line(aes(x=Index, y=value), data=ts_xts(pred$mean), color="red") +
labs(title="Air Passengers", x="Data", y="Numero passeggeri (/1000)") +
scale_x_date(breaks="1 year", date_labels="%Y")

```



In conclusione, è opportuno ricordarsi quanto segue:

- le funzioni ARIMA supportano gli oggetti `ts`, e *non* gli oggetti `xts`: se si passa un `xts` a `arima()` esso viene convertito in un `data.frame`, perdendo l'informazione temporale. È quindi **sempre opportuno convertire un `xts` mediante la funzione `tsbox::ts_xts()`**;
- viceversa, GGplot2 opera su oggetti `xts` e **non su `ts`**; gli oggetti `xts` vengono convertiti in `tibble` automaticamente invocando la funzione `ggplot2::fortify()`;
- la predizione restituisce un oggetto `predict`, che al suo interno contiene `ts` per il fit, la predizione e gli intervalli di confidenza. Per essere plottati essi devono prima essere convertiti nuovamente in un `xts` mediante `tsbox::ts_xts()`.

Infine, quando si costruisce l'estetica con `aes()` è utile verificare i *nomi* delle variabili messe a disposizione dai dati in questo modo:

```

ggplot(ts_xts(pred$mean))$data %>% str
## 'data.frame':   12 obs. of  2 variables:
## $ Index: Date, format: "1961-01-01" "1961-02-01" ...
## $ value: num  452 426 486 498 515 ...

```

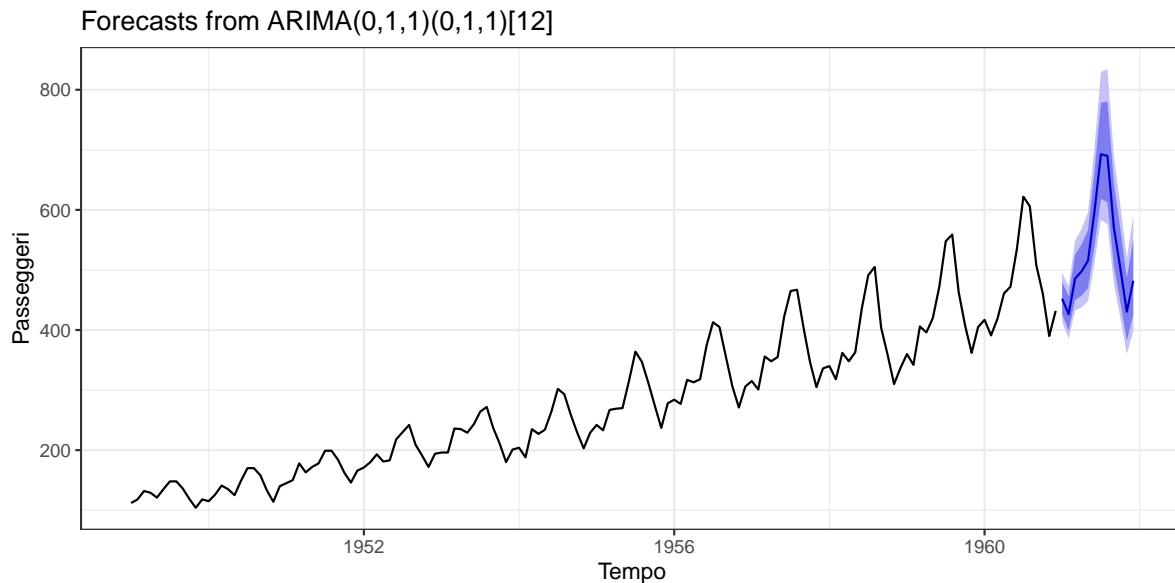
Come si vede, usando `ts_xts(pred$mean)` come parametro `data` i tempi vengono nominati `Index` e i valori `value`.

Ovviamente quanto sopra è già fornito mediante un'unica funzione, `forecast::autoforecast()`, che accetta come argomento sia `ts` che `xts`, che addirittura `forecast`:

```

autoforecast(pred) +
  labs(x="Tempo", y="Passeggeri")

```



Riprendiamo ora i dati COVID-19 per illustrare alcuni dettagli dell'uso di oggetti `xts` con `ggplot`:

```
library(lubridate)
##
## Attaching package: 'lubridate'
## The following object is masked from 'package:cowplot':
##       stamp
## The following objects are masked from 'package:base':
##       date, intersect, setdiff, union
library(scales)
##
## Attaching package: 'scales'
## The following object is masked from 'package:purrr':
##       discard
## The following object is masked from 'package:readr':
##       col_factor
url <- "https://covid.ourworldindata.org/data/owid-covid-data.csv"
datafile <- basename(url)
if (!file.exists(datafile) | difftime(now(), file.mtime(datafile), units="hours") > 24 ) {
  print("Downloading new data from the Internet")
  download.file(url, datafile)
}
covid <- read_csv(datafile) %>%
  filter(location=="Italy") %>%
  select(c("date", "new_cases", "new_deaths",
          "new_tests", "people_vaccinated_per_hundred",
          "positive_rate", "icu_patients"))
```

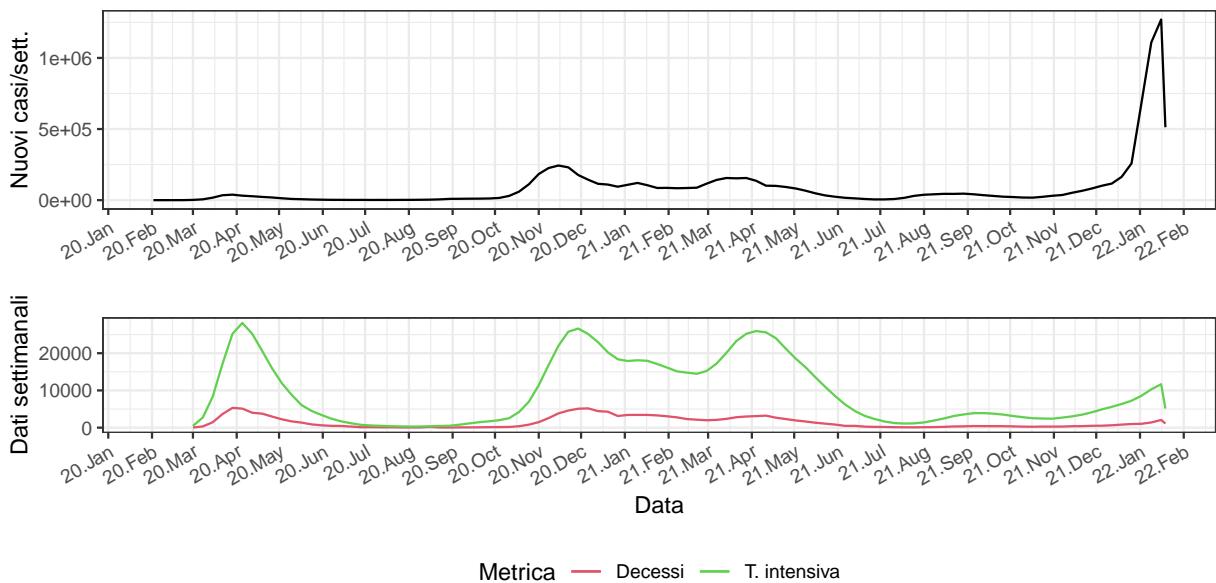
```
## Rows: 155857 Columns: 67
## -- Column specification -----
## Delimiter: ","
## chr (4): iso_code, continent, location, tests_units
## dbl (62): total_cases, new_cases, new_cases_smoothed, total_deaths, new_dea...
## date (1): date
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
cts <- xts(select(covid, new_cases:icu_patients), order.by = covid$date)
```

Una delle utili funzionalità di `xts` è quella di semplificare l'applicazione di funzioni a sotto-periodi, ad esempio per sommarizzare l'andamento settimanale (non risentendo in questo caso della periodicità settimanale dei test). In questo caso, però, l'oggetto `cts` che abbiamo creato è *multi-variato*, e le funzioni `apply.*` si applicano solo ad una colonna. Quindi bisogna procedere con alcuni accorgimenti:

```
p1 <- apply.weekly(cts$new_cases, sum) %>% ggplot(aes(x=Index, y=new_cases)) +
  geom_line() +
  labs(x="", y="Nuovi casi/sett.") +
  scale_x_date(date_breaks = "1 month", labels=label_date("%y.%b")) +
  theme(axis.text.x = element_text(angle = 30, hjust = 1))
p2 <- cbind(
  apply.weekly(cts$new_deaths, sum),
  apply.weekly(cts$icu_patients, sum)
) %>% ggplot() +
  geom_line(aes(x=Index, y=new_deaths, color="Decessi")) +
  geom_line(aes(x=Index, y=icu_patients, color="T. intensiva")) +
  labs(x="Data", y="Dati settimanali") +
  scale_x_date(date_breaks = "1 month", labels=label_date("%y.%b")) +
  scale_color_manual(name="Metrica", values=c(2,3)) +
  theme(axis.text.x = element_text(angle = 30, hjust = 1)) +
  theme(legend.position="bottom")
plot_grid(p1, p2, align="v", nrow=2)

## Warning: Removed 4 row(s) containing missing values (geom_path).

## Warning: Removed 4 row(s) containing missing values (geom_path).
```



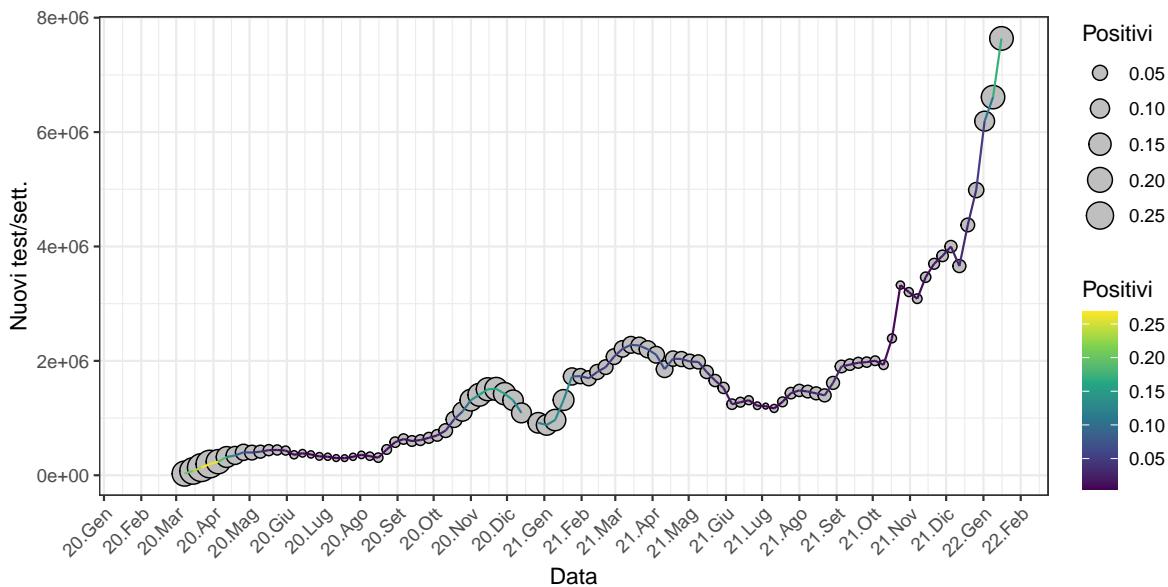
Come si vede, si può usare `cbind` per ricostruire un `xts` dopo aver applicato le funzioni di somma settimanale alle colonne di interesse.

In questo modo si possono realizzare anche grafici abbastanza complessi, *localizzando ad esempio l'asse dei tempi*:

```
Sys.setlocale("LC_TIME", "it_IT.UTF-8")
## [1] "it_IT.UTF-8"

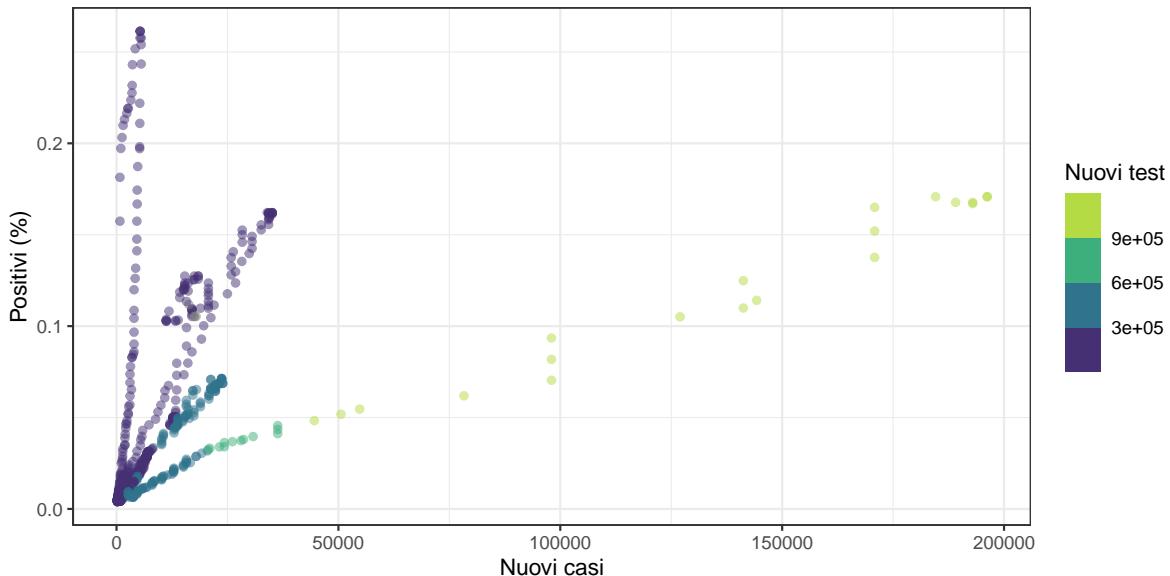
cbind(
  apply.weekly(cts$new_tests, sum),
  apply.weekly(cts$positive_rate, max)
) %>%
  ggplot() +
  geom_point(aes(x=Index, y=new_tests, size=positive_rate),
             pch=21, color="black", fill=gray(0.75)) +
  geom_line(aes(x=Index, y=new_tests, color=positive_rate)) +
  scale_color_viridis_c() +
  scale_x_date(date_breaks = "1 month", labels=label_date("%y.%b")) +
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
  labs(x="Data", y="Nuovi test/sett.", size="Positivi", color="Positivi")

## Warning: Removed 7 rows containing missing values (geom_point).
## Warning: Removed 6 row(s) containing missing values (geom_path).
```



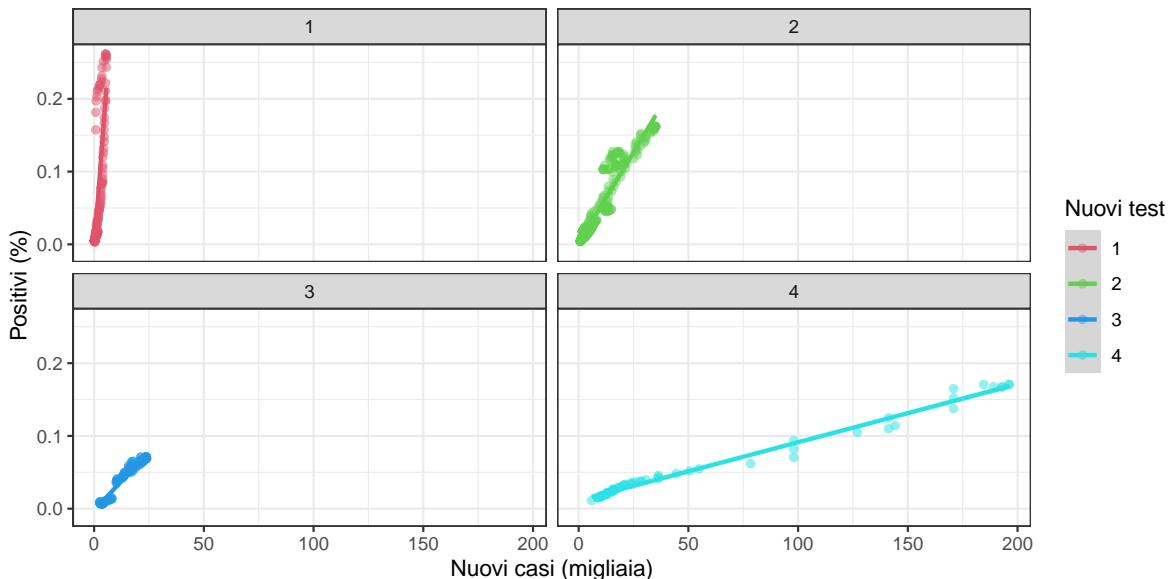
Oltre che mediante la somma settimanale, i dati possono essere trattati anche mediante estimatori a finestra mobile di 7 giorni:

```
ctsm <- rollmedian(cts, 7)
ctsm %>%
  ggplot(aes(x=new_cases, y=positive_rate, color=new_tests)) +
  geom_point(alpha=0.5) +
  scale_color_viridis_b() +
  labs(x="Nuovi casi", y="Positivi (%)", color="Nuovi test")
## Warning: Removed 34 rows containing missing values (geom_point).
```



In questo caso particolare possiamo notare una clusterizzazione dei dati. Proviamo ad approfondire per capirne il motivo. Useremo la funzione base `cut` per assegnare un indice di classe in corrispondenza di un dato intervallo sul numero di test effettuati al giorno. Si noti inoltre che se vogliamo usare verbi `dplyr` su un `xts` dobbiamo prima convertirlo in data frame mediante `fortify`: mentre infatti questo passaggio è implicito in `ggplot`, non lo è nelle funzioni `dplyr`.

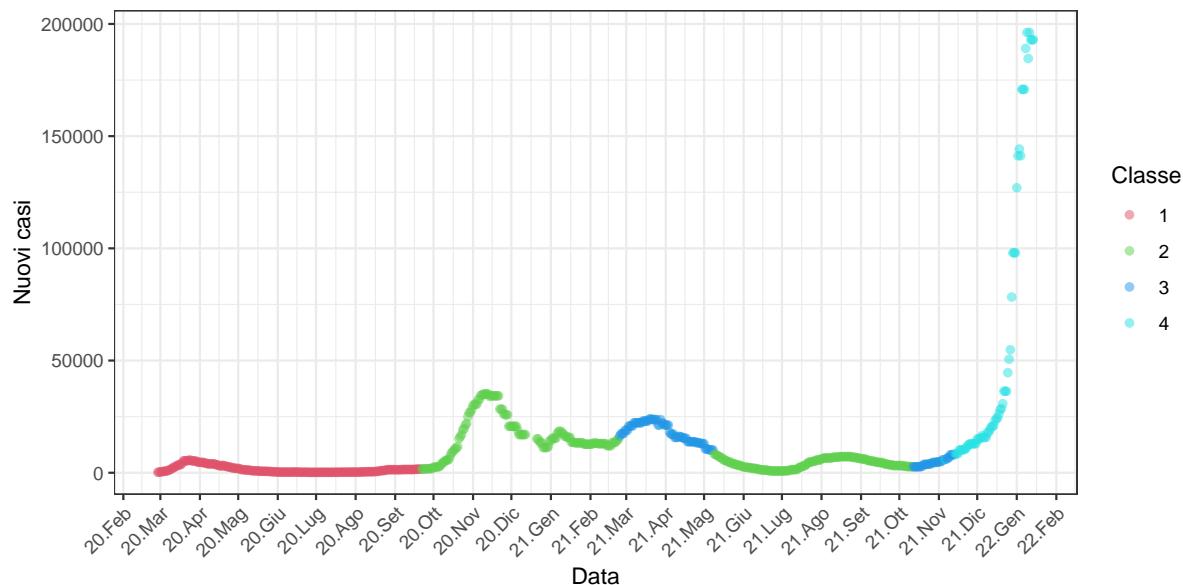
```
ctsm$cat <- cut(ctsm$new_tests,
                 breaks=c(0, 100, 310, 500, 5000) * 1000)
ctsm %>% fortify() %>%
  filter(!is.na(new_tests)) %>%
  ggplot(aes(x=new_cases, y=positive_rate, color=factor(cat))) +
  geom_point(alpha=0.5) +
  geom_smooth(method="lm") +
  scale_x_continuous(labels=~ .x/1000) +
  scale_color_manual(values = c(2:6)) +
  facet_wrap(~cat) +
  labs(x="Nuovi casi (migliaia)", y="Positivi (%)", color="Nuovi test")
## `geom_smooth()` using formula 'y ~ x'
## Warning: Removed 7 rows containing non-finite values (stat_smooth).
## Warning: Removed 7 rows containing missing values (geom_point).
```



Cioè con numeri diversi di campioni ci sono diverse relazioni tra percentuale di positivi e nuovi casi individuati.

Vediamo ora di vedere queste classi come sono mappate sul tempo:

```
ctsm %>% fortify() %>%
  filter(!is.na(new_tests)) %>%
  ggplot(aes(x=Index, y=new_cases, color=factor(cat))) +
  geom_point(alpha=0.5) +
  scale_color_manual(values = c(2:6)) +
  scale_x_date(date_breaks = "1 month", labels=label_date("%y.%b")) +
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
  labs(x="Data", y="Nuovi casi", color="Classe")
```



2.4 Diamanti grezzi!

L'analisi dei sistemi multi-variati richiede sempre particolare attenzione. Sono due i casi in cui l'analista può essere fuorviato dai dati:

1. fattori confusi
2. fattori non ortogonali

In entrambi i casi le tecniche di *Design of Experiments* viste nella Parte 1 sono d'aiuto.

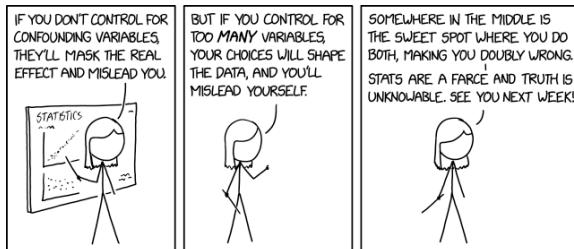
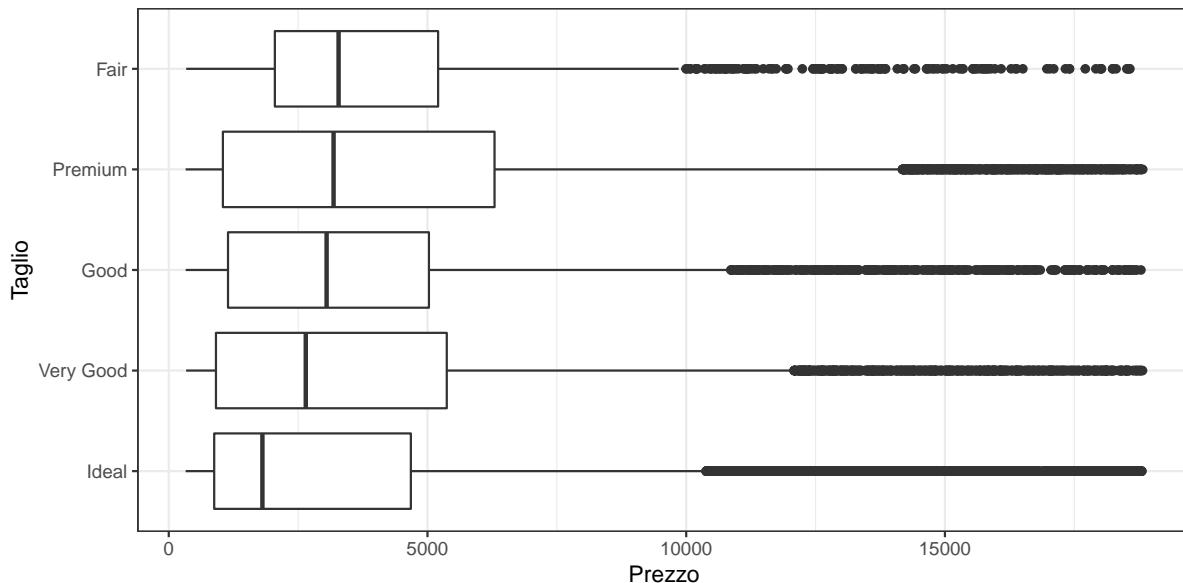


Figura 1: <https://xkcd.com/2560/>

2.4.1 Fattori confusi

La libreria `tidyverse` contiene la tibble `diamonds`, un elenco di caratteristiche di 53940 diamanti. Vediamo come il prezzo dipende da alcuni fattori:

```
ggplot(diamonds, aes(x=reorder(cut, price, FUN=median), y=price)) +
  geom_boxplot() +
  coord_flip() +
  labs(x="Taglio", y="Prezzo")
```

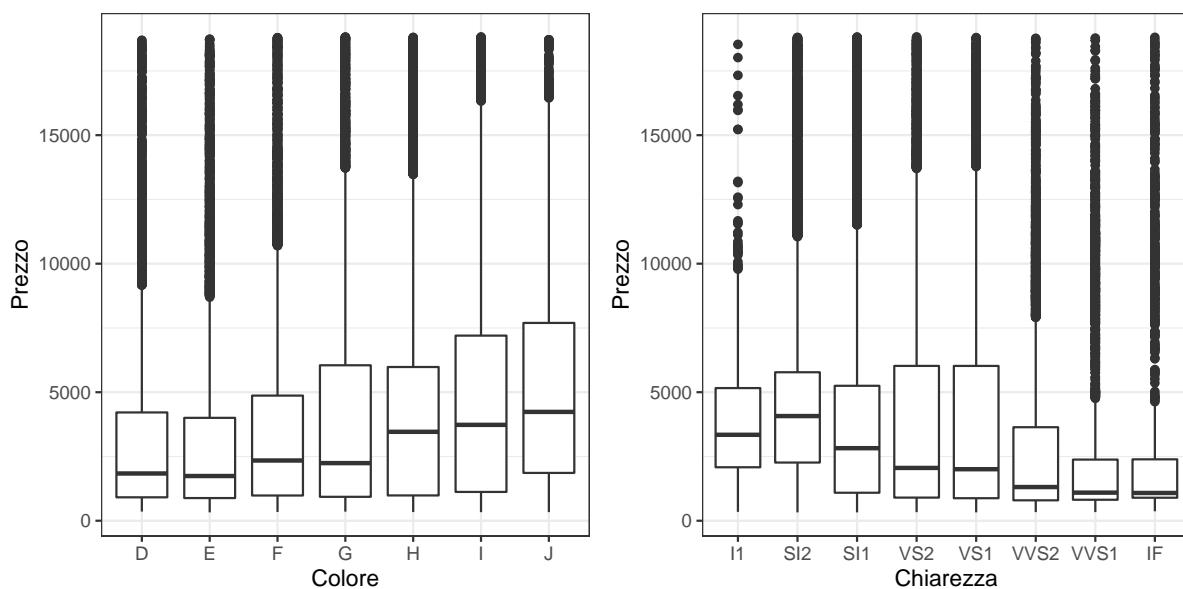


```
head(diamonds$cut)
## [1] Ideal      Premium    Good       Premium    Good       Very Good
## Levels: Fair < Good < Very Good < Premium < Ideal
```

Come si vede, nonostante **Ideal** sia il taglio migliore e **Fair** il peggiore, *in media* quest'ultimo è associato ai prezzi maggiori, e il primo ai minori.

Analogamente, in funzione di colore e chiarezza:

```
p1 <- ggplot(diamonds, aes(x=color, y=price)) +
  geom_boxplot() +
  labs(x="Colore", y="Prezzo")
p2 <- ggplot(diamonds, aes(x=clarity, y=price)) +
  geom_boxplot() +
  labs(x="Chiarezza", y="Prezzo")
plot_grid(p1, p2)
```

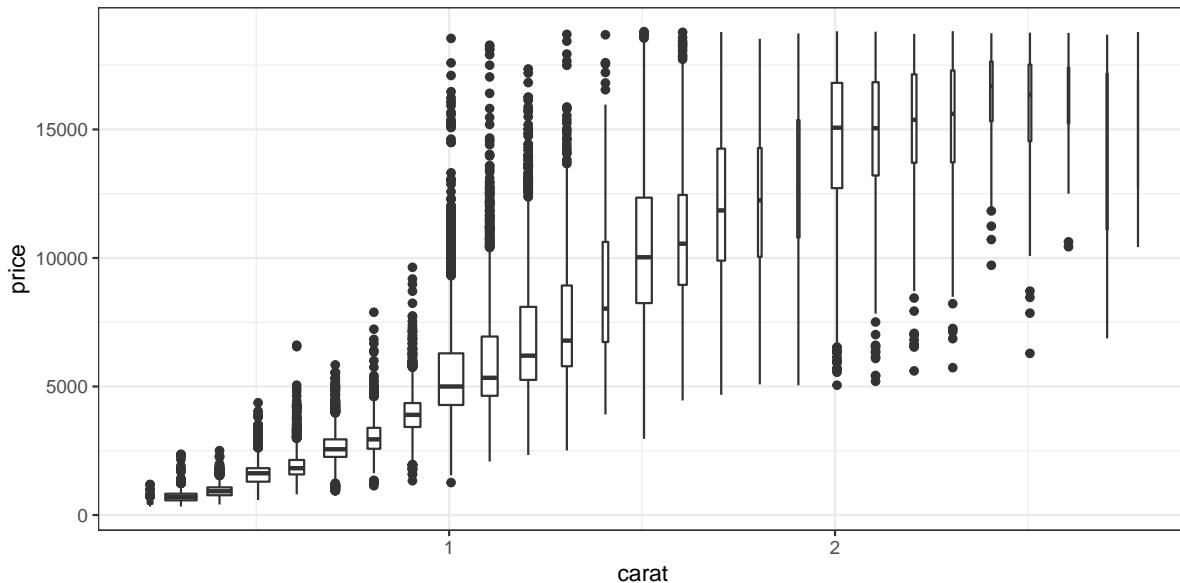


In questo caso i diamanti peggiori sono di colore J (giallognoli) e chiarezza I1, cioè con inclusioni visibili a occhio nudo. **Eppure sono tra i più cari. Perché?**

Il motivo di questa conclusione fuorviante è che stiamo trascurando un fattore confuso: la dimensione (**carat**).

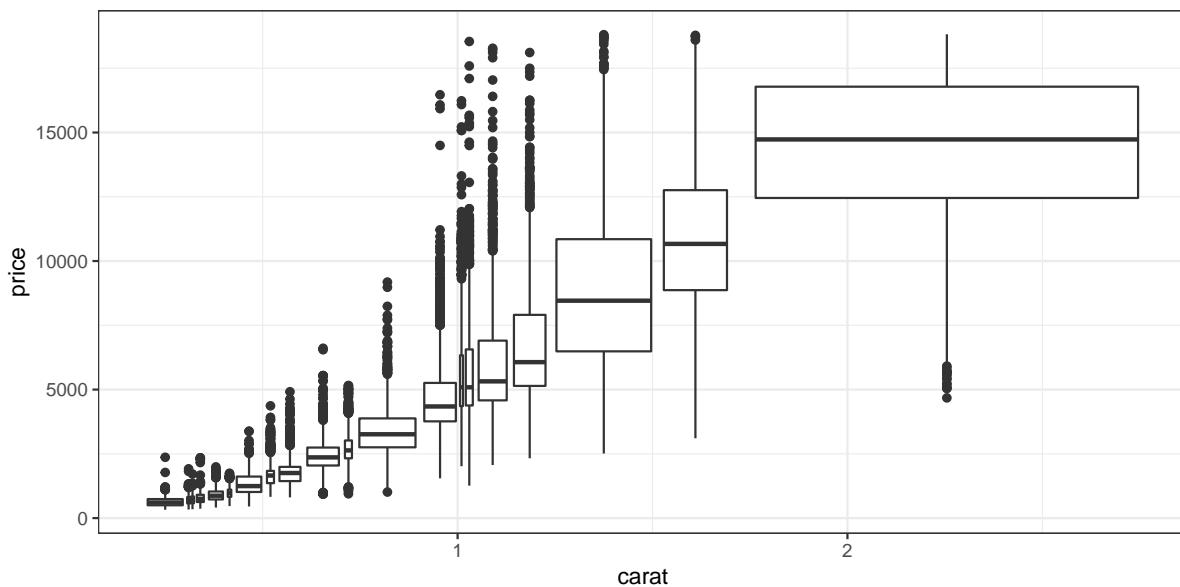
Possiamo studiare l'effetto della dimensione con grafici come quelli realizzati più su, oppure possiamo realizzare un boxplot, raggruppandoli per classi spaziate di 0.1 carati:

```
diamonds %>%
  filter(carat < 3) %>%
  ggplot(aes(x = carat, y = price)) +
  geom_boxplot(mapping = aes(group = cut_width(carat, 0.1)), varwidth = T)
```



Il parametro **varwidth=T** adatta la larghezza dei boxplot in funzione del numero di osservazioni per ogni classe. Un altro modo per visualizzare la numerosità delle classi è raggruppando le osservazioni per classi di uguale numerosità, diciamo 20 diamanti; in questo caso i box più larghi sono quelli meno numerosi:

```
diamonds %>%
  filter(carat < 3) %>%
  ggplot(aes(x = carat, y = price)) +
  geom_boxplot(mapping = aes(group = cut_number(carat, 20)))
```



Focalizziamoci sui diamanti con carati inferiori al 99-esimo percentile, cioè

```
(cmax <- quantile(diamonds$carat, p=0.99))

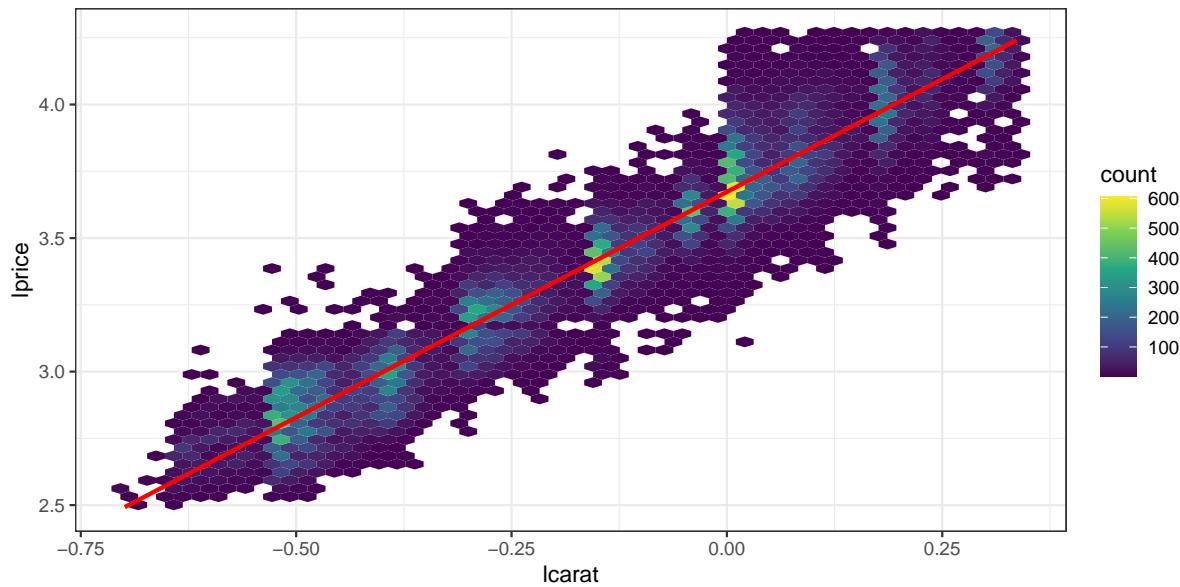
## 99%
## 2.18
```

Inoltre, data la grande variazione di prezzo trasformiamo il modello in base al logaritmo del prezzo:

```
diamonds2 <- diamonds %>%
  filter(carat < cmax) %>%
  mutate(lprice = log10(price), lcarat=log10(carat))

diamonds2 %>%
  ggplot(aes(x=lcarat, y=lprice)) +
  geom_hex(bins=50) +
  geom_smooth(method="lm", color="red") +
  scale_fill_viridis_c()

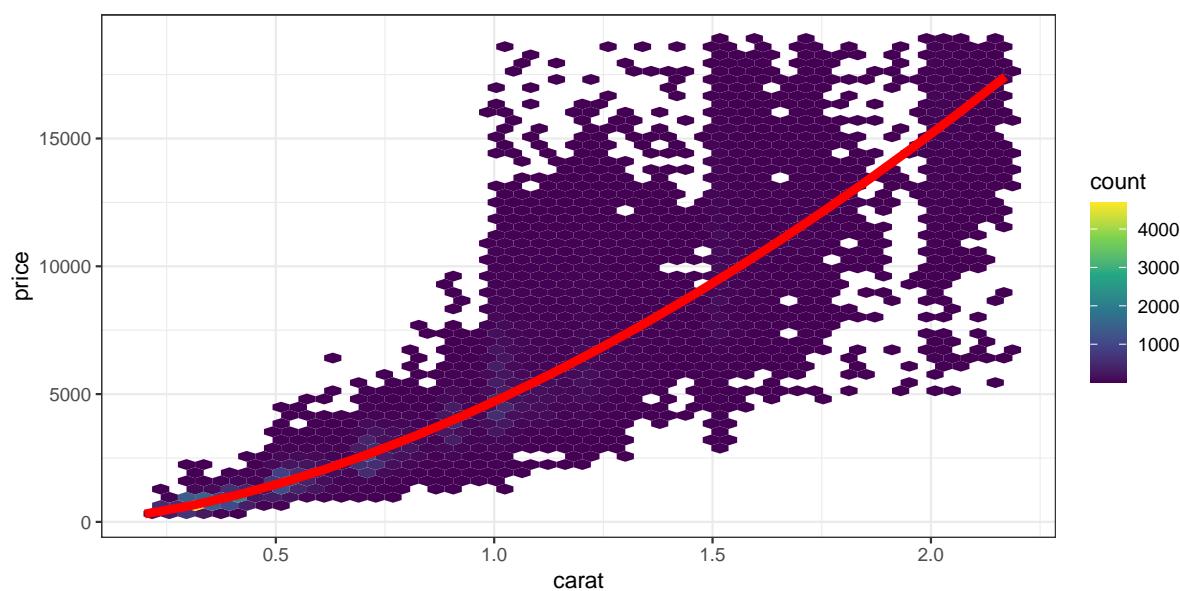
## `geom_smooth()` using formula 'y ~ x'
```



È evidente un andamento lineare sul piano bi-logaritmico. Ora possiamo creare un modello lineare sui dati trasformati e *sottrarlo* ai dati originali, in modo da poter studiare, indipendentemente dalla dimensione, l'effetto di colore, taglio e chiarezza sul prezzo.

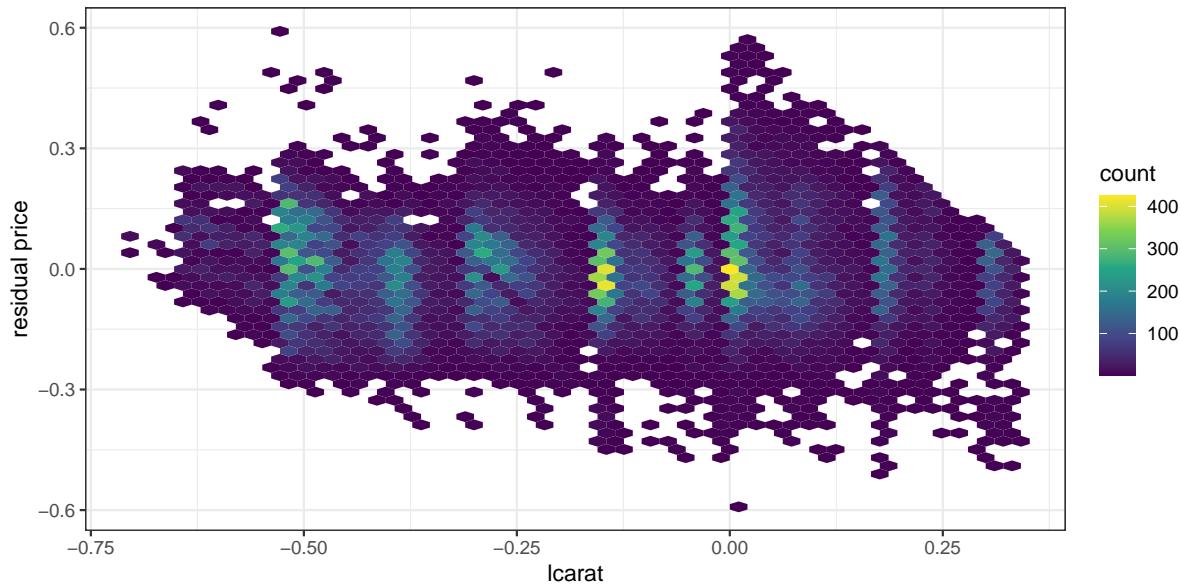
```
diamonds2.lm <- lm(lprice~lcarat, data=diamonds2)
fit <- diamonds2 %>%
  data_grid(carat = seq_range(carat, 20)) %>%
  mutate(lcarat = log10(carat)) %>%
  add_predictions(diamonds2.lm, "lprice") %>%
  mutate(price = 10^lprice)

ggplot(diamonds2, aes(x=carat, y=price)) +
  geom_hex(bins=50) +
  geom_line(data=fit, color="red", size=2) +
  scale_fill_viridis_c()
```



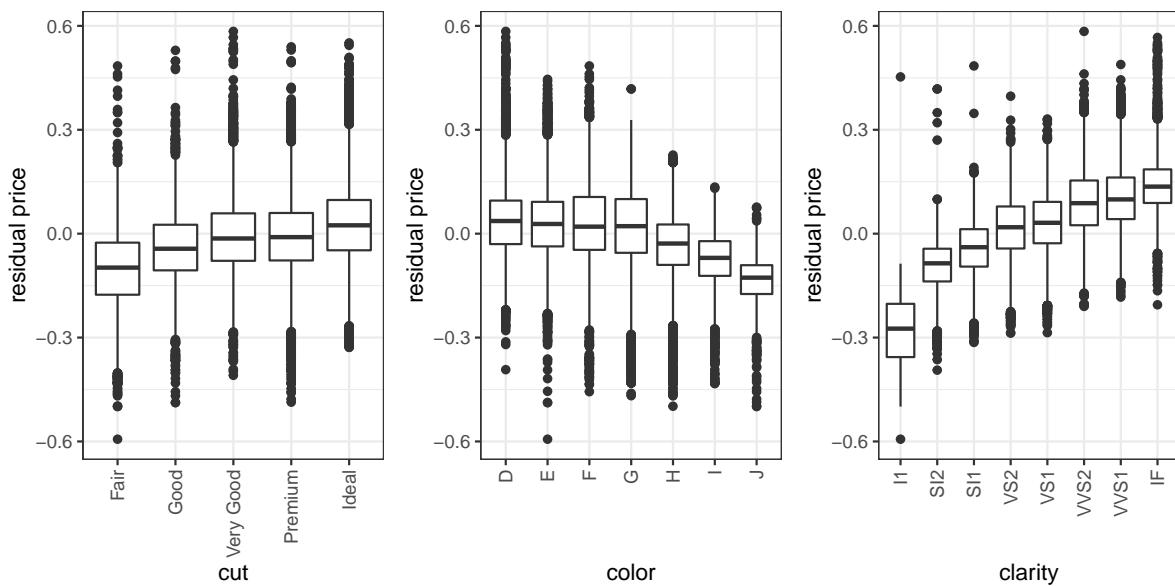
Ora aggiungiamo i residui a diamonds2:

```
(diamonds2 <- diamonds2 %>%
  add_residuals(diamonds2.lm, "residual price")) %>%
  ggplot(aes(x=lcarat, y=`residual price`)) +
  geom_hex(bins=50) +
  scale_fill_viridis_c()
```



Finalmente possiamo studiare l'effetto di taglio, colore e chiarezza:

```
p1 <- ggplot(diamonds2, aes(x=cut, y=`residual price`)) + geom_boxplot() +
  theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust=1))
p2 <- ggplot(diamonds2, aes(x=color, y=`residual price`)) + geom_boxplot() +
  theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust=1))
p3 <- ggplot(diamonds2, aes(x=clarity, y=`residual price`)) + geom_boxplot() +
  theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust=1))
plot_grid(p1, p2, p3, nrow=1, align="h")
```



E questi grafici, depurati dal fattore confuso carat, mostrano l'effettiva influenza di questi fattori sulla componente residua del prezzo.

È evidente, quindi, che in generale una analisi preliminare dei fattori che influenzano un dato processo è fondamentale per assicurarsi che non ci siano fattori confusi che mascherano l'effetto di altri fattori.

2.4.2 Fattori non ortogonali

In certi casi l'interazione tra due fattori rilevata da un'analisi ANOVA significa che si sono scelti fattori correlati.

Consideriamo l'esempio di un processo di stampa 3D a deposizione di fuso (FDM): due importanti parametri della macchina sono la velocità di estrusione (mm^3/min) e la velocità di spostamento della testina (mm/min). Carichiamo i dati di un piano fattoriale a due livelli, ripetuto tre volte, per i due fattori `Extr` e `Feed`, in cui la resa è la resistenza del materiale.

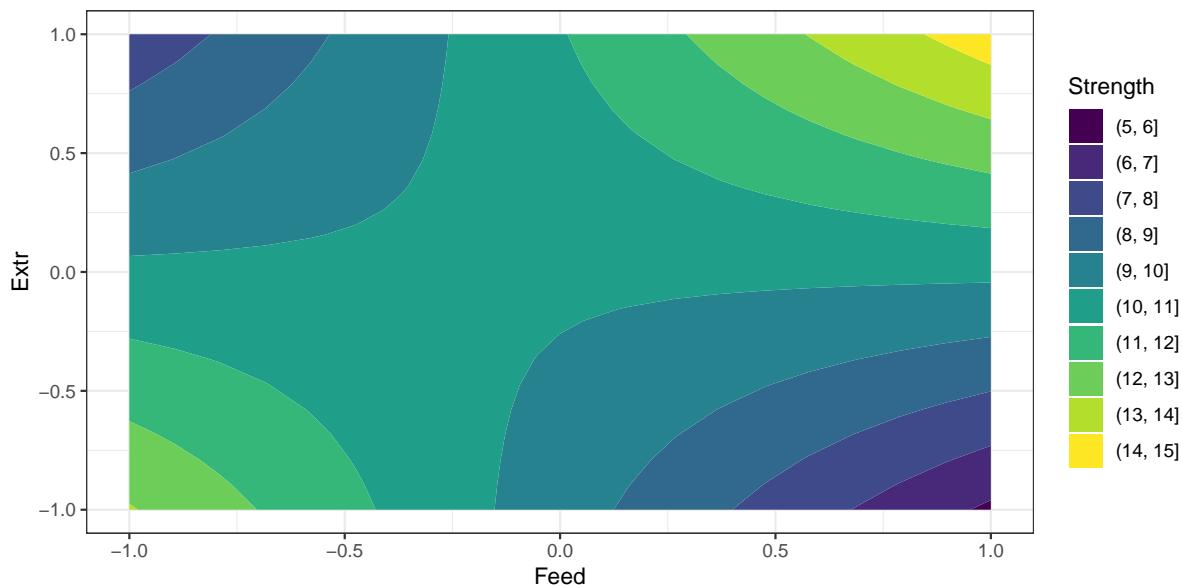
```
df <- read_delim("3dprint.txt", comment="#", col_types = c(rep="i", Feed="d", Extr="d"))
df.lm <- lm(Strength~Feed*Extr, data=df)
anova(df.lm)

## Analysis of Variance Table
##
## Response: Strength
##             Df  Sum Sq Mean Sq  F value    Pr(>F)
## Feed          1   1.191   1.191   1.3050   0.2863
## Extr          1   6.660   6.660   7.2996   0.0270 *
## Feed:Extr    1 157.543 157.543 172.6651 1.07e-06 ***
## Residuals    8   7.299   0.912
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

La tabella ANOVA indica un'elevata significatività dell'interazione tra i due parametri e una modesta significatività della velocità di estrusione, mentre la velocità di avanzamento risulta non significativa. Come mai l'effetto più intenso è quello di un'interazione? come mai la velocità non ha effetto, ma lo ha la sua interazione?

Interpoliamo il modello sul dominio di analisi e realizziamo un diagramma a livelli della *superficie di risposta*:

```
df.lm <- lm(Strength~Feed:Extr + Extr, data=df)
df %>%
  data_grid(Feed=seq_range(Feed, 20), Extr=seq_range(Extr, 20)) %>%
  add_predictions(df.lm, "Strength_pred") %>%
  ggplot(aes(x=Feed, y=Extr, z=Strength_pred)) +
  geom_contour_filled() +
  labs(fill="Strength")
```



La superficie di risposta ha una forma a sella: la porosità è alta quando entrambi i fattori sono bassi o quando entrambi sono alti, mentre è bassa quando sono di segno opposto. Avere una forte interazione significa che l'effetto di un aumento di velocità dipende dal *livello* dell'altro fattore, velocità di estrusione.

Ragionando sul processo fisico, ci rendiamo conto che `Feed` e `Extr` si combinano nella *sezione* di materiale depositato (come `Extr/Feed`), e tanto più essa è piccola, tanto più alta è la probabilità di avere pori, quindi minore resistenza. Altresì, le combinazioni $(-1, -1)$ e $(+1, +1)$ dei due fattori producono di fatto la stessa sezione depositata, e quindi rappresentano la stessa condizione di prova.

È quindi preferibile considerare come fattore la sezione depositata e la velocità di estrusione:

```
(df2 <- df %>% mutate(Xsec = Extr/Feed)) %>% filter(rep==1) %>% select(Feed:Xsec)

## # A tibble: 4 x 4
##   Feed  Extr Strength Xsec
##   <dbl> <dbl>    <dbl> <dbl>
## 1    -1    -1     11.9    1
## 2     1    -1      5.57   -1
## 3    -1     1      7.96   -1
## 4     1     1     14.0    1

df2.lm <- lm(Strength~Extr*Xsec, data=df2)
anova(df2.lm)

## Analysis of Variance Table
##
## Response: Strength
##             Df  Sum Sq Mean Sq F value Pr(>F)
## Extr          1  6.660  6.660  7.2996  0.0270 *
## Xsec          1 157.543 157.543 172.6651 1.07e-06 ***
## Extr:Xsec    1  1.191  1.191  1.3050  0.2863
## Residuals    8   7.299   0.912
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Come si vede, il modello ora contiene solo l'effetto dei due fattori, mentre l'interazione risulta non significativa. In grafico:

```
df2.lm <- lm(Strength~Extr + Xsec, data=df2)
df2 %>%
  data_grid(Extr=seq_range(Extr, 20), Xsec=seq_range(Xsec, 20)) %>%
  add_predictions(df2.lm, "Strength_pred") %>%
  ggplot(aes(x=Extr, y=Xsec, z=Strength_pred)) +
  geom_contour_filled() +
  labs(fill="Strength")
```

