R-Bootcamp - Notes

Nils Rechberger

2026-01-26

Intro

What is R?

R is a powerful programming language and software environment specifically designed for statistical computing, data analysis, and high-quality graphical visualization. It is widely used by researchers and data scientists due to its vast ecosystem of packages and its ability to handle complex data manipulation.

Steps of a data analysis

1. Import Data
2. Prepare Data:
3. Graphical Analysis
4. Fit Model
5. Check and interpret results
6. Report results

R Basics

R-Script

Statistical analyses are composed by many steps, where intermediate results are stored, inspected and commented. The sequence of instructions can be collected in a script.

Vectors

One-dimensional sequence of elements of a given type (numeric, character (or string), or logical). Can be created with the `c()` function ("c" stands for "combine").

```r
my_vector <- c(1, 2, 3, 4)
my_vector
```

```
[1] 1 2 3 4
```

Accessing elements of a vector can be done via indexing, which uses square brackets.

```r
v.age <- c(15, 25, 17, 34, 6)
v.age[c(2, 3, 5)] # Several elements accessed simultaneously
```

```
[1] 25 17  6
```

Data Frames

Data often comes in 2-dimensional tables. Data frames are 2-dimension objects. All elements of a given column in a data frame belong to the same class (e.g. numeric, character, etc.). Data frames can be created via the `data.frame()` function.

Lists

Lists are more flexible objects that can store objects of different classes and different dimension (e.g. models). All elements of a list can be accessed using the double squared brackets. Named elements of a list can be accessed via the `$` symbol.

Importing Data

Data sets can be imported into R from:

- An existing file
- An internet connection (url)
- A data base (not discussed here)
- etc.

Working Directory

R is working/pointing to a specific directory on your machine.

```r
getwd() # Check where your current working directory is
```

```
[1] "/home/nils/dev/mscids-notes/hs25/r_bootcamp"
```

The working directory can be changed with the `setwd()` function.

Data Manipulation using dplyr

```
Loading required package: dplyr


Attaching package: 'dplyr'

The following objects are masked from 'package:stats':

    filter, lag

The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union

Loading required package: tidyr
```

Long and wide format

- **Wide**: One row per subject, columns for repeated measures.
- **Long**: One row per measurement, with ID and "variable name" columns.

The function `pivot_longer` converts a data frame from wide to long format.

```r
diamonds$ID <- 1:nrow(diamonds)
diamonds.long.format <- pivot_longer(
    diamonds, cols = -c(ID, cut, color, clarity),
    names_to = "feature",
    values_to = "value"
)
```

The function `pivot_wider` converts a data frame from long to wide format.

```
pivot_wider(
    diamonds.long.format,
    names_from = feature,
    values_from = value
)
```

```
# A tibble: 53,940 x 11
   cut       color clarity    ID carat depth table price     x     y     z
   <ord>     <ord> <ord>   <int> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1 Ideal     E     SI2         1  0.23  61.5    55   326  3.95  3.98  2.43
 2 Premium   E     SI1         2  0.21  59.8    61   326  3.89  3.84  2.31
 3 Good      E     VS1         3  0.23  56.9    65   327  4.05  4.07  2.31
 4 Premium   I     VS2         4  0.29  62.4    58   334  4.2   4.23  2.63
 5 Good      J     SI2         5  0.31  63.3    58   335  4.34  4.35  2.75
 6 Very Good J     VVS2        6  0.24  62.8    57   336  3.94  3.96  2.48
 7 Very Good I     VVS1        7  0.24  62.3    57   336  3.95  3.98  2.47
 8 Very Good H     SI1         8  0.26  61.9    55   337  4.07  4.11  2.53
 9 Fair      E     VS2         9  0.22  65.1    61   337  3.87  3.78  2.49
10 Very Good H     VS1        10  0.23  59.4    61   338  4     4.05  2.39
# i 53,930 more rows
```

Note: `pivot_longer()` and `pivot_wider()` are function from `tidyr`

Function of dplyr

`select()`

Selecting specific columns in a dataset.

```
select(diamonds, x, y, z) # Select specific cols
```

```
# A tibble: 53,940 x 3
       x     y     z
   <dbl> <dbl> <dbl>
 1  3.95  3.98  2.43
 2  3.89  3.84  2.31
 3  4.05  4.07  2.31
 4  4.2   4.23  2.63
 5  4.34  4.35  2.75
 6  3.94  3.96  2.48
 7  3.95  3.98  2.47
 8  4.07  4.11  2.53
 9  3.87  3.78  2.49
10  4     4.05  2.39
# i 53,930 more rows
```

```r
select(diamonds, -ID) # All cols except "ID"
select(diamonds, contains("ce")) # Cols whose name contains "ce"
select(diamonds, matches("c.")) # Selecting using regex
```

`filter()`

We can use the following function to subset a data frame. It will retain all rows that satisfy the specified condition.

```r
filter(diamonds, `carat` > 4, `cut` == "Fair")
```

```
# A tibble: 3 x 11
  carat cut   color clarity depth table price     x     y     z    ID
  <dbl> <ord> <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl> <int>
1  4.13 Fair  H     I1       64.8    61 17329  10    9.85  6.43 27131
2  5.01 Fair  J     I1       65.5    59 18018  10.7 10.5   6.98 27416
3  4.5  Fair  J     I1       65.8    58 18531  10.2 10.2   6.72 27631
```

> Note: The comma `,` between the statements in the function `filter()` means **AND**.

`mutate()` and `across()`

Create or calculate new columns.

```r
mutate(diamonds, xPLUSy = x + y, depth.check = 2 * z / (xPLUSy))
```

```
# A tibble: 53,940 x 13
   carat cut       color clarity depth table price     x     y     z    ID xPLUSy
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl> <int>  <dbl>
 1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43     1   7.93
 2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31     2   7.73
 3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31     3   8.12
 4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63     4   8.43
 5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75     5   8.69
 6  0.24 Very Go~ J     VVS2     62.8    57   336  3.94  3.96  2.48     6   7.9
 7  0.24 Very Go~ I     VVS1     62.3    57   336  3.95  3.98  2.47     7   7.93
 8  0.26 Very Go~ H     SI1      61.9    55   337  4.07  4.11  2.53     8   8.18
 9  0.22 Fair     E     VS2      65.1    61   337  3.87  3.78  2.49     9   7.65
10  0.23 Very Go~ H     VS1      59.4    61   338  4     4.05  2.39    10   8.05
# i 53,930 more rows
# i 1 more variable: depth.check <dbl>
```

Note: We can use the newly created `xPLUSy` to calculate `depth.check`.

By combining `mutate()` with `across()`, we can create or calculate new columns for each of the variables/columns.

```
mutate(
    diamonds,
    across(
        -c(cut, color, clarity),
        .fns = cumsum
    )
)
```

```
# A tibble: 53,940 x 11
   carat cut       color clarity depth table price     x     y     z    ID
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl> <int>
 1  0.23 Ideal     E     SI1      61.5    55   326  3.95  3.98  2.43     1
 2  0.44 Premium   E     SI1     121.    116   652  7.84  7.82  4.74     3
 3  0.67 Good      E     VS1     178.    181   979 11.9  11.9   7.05     6
 4  0.96 Premium   I     VS2     241.    239  1313 16.1  16.1   9.68    10
 5  1.27 Good      J     SI2     304.    297  1648 20.4  20.5  12.4     15
 6  1.51 Very Good J     VVS2    367.    354  1984 24.4  24.4  14.9     21
 7  1.75 Very Good I     VVS1    429     411  2320 28.3  28.4  17.4     28
 8  2.01 Very Good H     SI1     491.    466  2657 32.4  32.5  19.9     36
 9  2.23 Fair      E     VS2     556     527  2994 36.3  36.3  22.4     45
10  2.46 Very Good H     VS1     615.    588  3332 40.3  40.4  24.8     55
# i 53,930 more rows
```

`summarise()`

Summarise or aggregate.

```r
summarise(
    diamonds,
    mean.carat = mean(carat),
    mean.price = mean(price),
    median.carat = median(carat),
    median.price = median(price)
)
```

```
# A tibble: 1 x 4
  mean.carat mean.price median.carat median.price
       <dbl>      <dbl>        <dbl>        <dbl>
1      0.798      3933.          0.7         2401
```

Again we can apply the `across()` function to summarise or aggregate each of the variables/columns.

```r
summarise(
    diamonds,
    across(
        -c(cut, color, clarity),
        .fns = list(
            mean = ~ mean(.x, na.rm = TRUE),
            sum = ~ sum(.x, na.rm = TRUE)
        )
    )
)
```

```
# A tibble: 1 x 16
  carat_mean carat_sum depth_mean depth_sum table_mean table_sum price_mean
       <dbl>     <dbl>      <dbl>     <dbl>      <dbl>     <dbl>      <dbl>
1      0.798    43041.       61.7  3330763.       57.5  3099240.      3933.
# i 9 more variables: price_sum <int>, x_mean <dbl>, x_sum <dbl>, y_mean <dbl>,
#   y_sum <dbl>, z_mean <dbl>, z_sum <dbl>, ID_mean <dbl>, ID_sum <int>
```

`group_by()`

Group the data such that all functions of `dplyr` are applied to each group of the data separately

```r
group_by(
    diamonds,
    cut
)
```

```
# A tibble: 53,940 x 11
# Groups:   cut [5]
   carat cut       color clarity depth table price     x     y     z    ID
   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl> <int>
 1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43     1
 2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31     2
 3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31     3
 4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63     4
 5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75     5
 6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48     6
 7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47     7
 8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53     8
 9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49     9
10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39    10
# i 53,930 more rows
```

> Note: If you don't use the `ungroup()` function then the rows of the data frame are still grouped.

The pipe operator %>%

The pipe operator `%>%` makes it easy to read code. Read from left to right, like the natural functions call order.

```r
diamonds %>%
    select(x, y, z)
```

```
# A tibble: 53,940 x 3
       x     y     z
   <dbl> <dbl> <dbl>
 1  3.95  3.98  2.43
 2  3.89  3.84  2.31
 3  4.05  4.07  2.31
 4  4.2   4.23  2.63
 5  4.34  4.35  2.75
 6  3.94  3.96  2.48
 7  3.95  3.98  2.47
 8  4.07  4.11  2.53
 9  3.87  3.78  2.49
10  4     4.05  2.39
# i 53,930 more rows
```

Note: In base R, there is already a pipe operator: `|>`

Join or Combine Data Sets

`left_join()`

A `left_join` returns all observations from `x` and from `y` which could be matched to `x`.

```
left_join(
    x = data.A,
    y = data.B,
    by = "name"
)
```

`right_join()`

A `right_join` returns all observations from `y` and from `x` which could be matched to `y`.

```
right_join(
    x = data.A,
    y = data.B,
    by = "name"
)
```

`inner_join()`

An `inner_join` returns all observations which could be matched in both data sets.

```
inner_join(
    x = data.A,
    y = data.B,
    by = "name"
)
```

`full_join()`

A `full_join` returns all observations from `x` and `y`.

```r
full_join(
    x = data.A,
    y = data.B,
    by = "name"
)
```

Note

Sometimes, variables are named differently in the data sets that we wish to join. We can still join these data sets as long as we indicate what matches what.

```r
full_join(
    x = data.A,
    y = data.B,
    by = c("name.A" = "name.B")
)
```

ggplot2

```
Attaching package: 'ggplot2'


The following object is masked _by_ '.GlobalEnv':

    diamonds
```

`ggplot2` is a powerful and widely-used R package for data visualization based on the Grammar of Graphics, allowing users to create complex plots by layering components like scales, layers, and aesthetics. It excels at handling data in Long format, as shown in your table, making it easy to map variables to visual properties like color, size, and shape.

Initial Function Call

`ggplot()` function initializes a ggplot object (or conceptually a plot), which can then be modified to create a plot using `ggplot2`.

```
ggplot(
    data,
    mapping = aes(
        x = x_var,
        y = y_var
    )
)
```

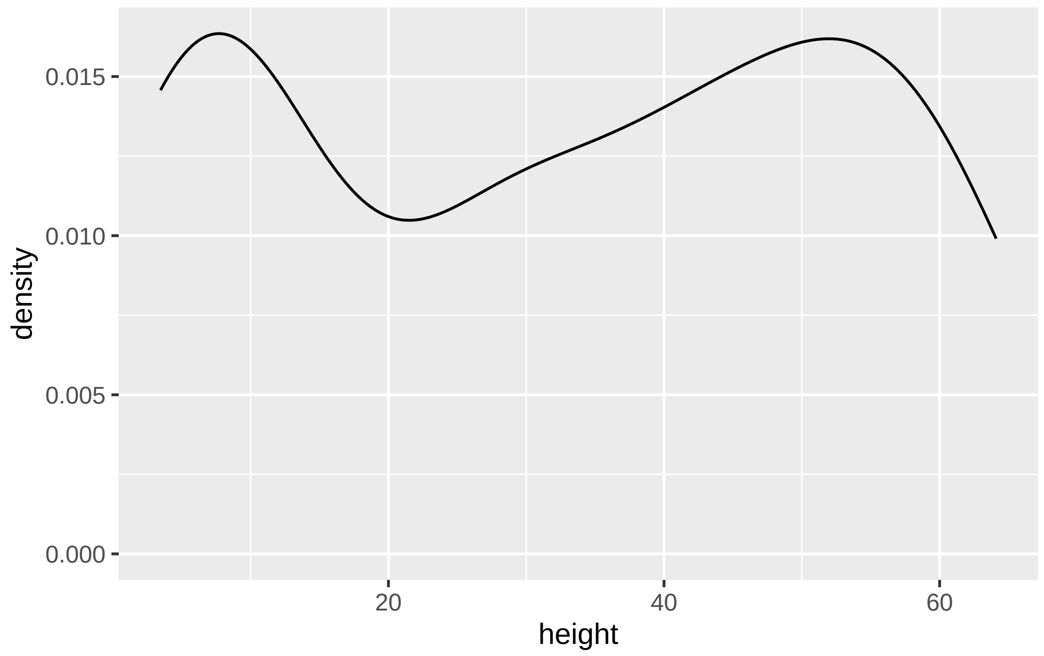
Afterwards, you add layers with the "+" symbol.

```
ggplot(
    data,
    mapping = aes(
        x = x_var,
        y = y_var
    ) +
) + geom_point()
```

Density Plot

You can create density plots. In this case the y argument is not needed.

```
ggplot(
    data = Loblolly,
    mapping = aes(
        x = height
    )
) + geom_density()
```



It is possible to add a “rug” to the plot, showing the value taken by each observation.

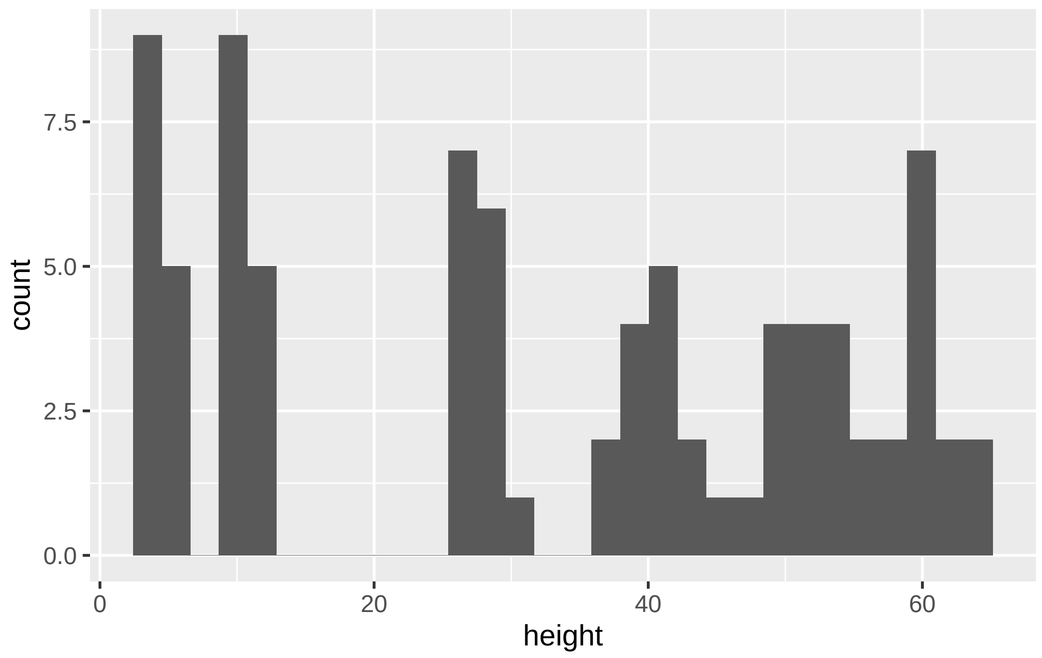
```
ggplot(
    data = Loblolly,
    mapping = aes(
        x = height
    )
) +
geom_density() +
geom_rug()
```

Histogram

Instead of a density plot, you can create histograms.

```
ggplot(
    data = Loblolly,
    mapping = aes(
        x = height
    )
) +
geom_histogram()
```

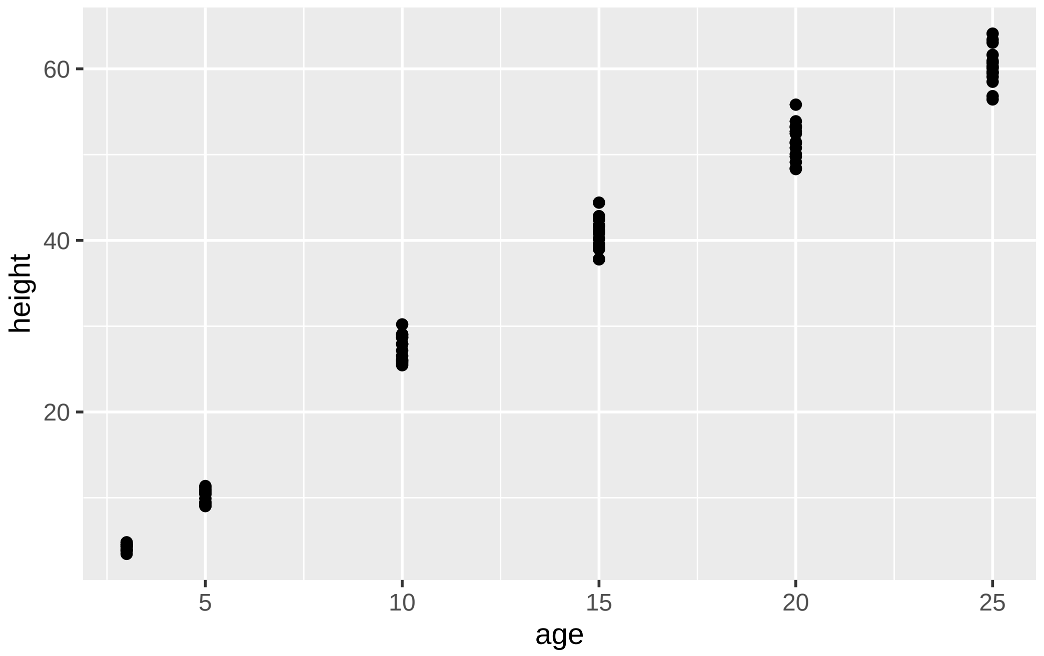
```
`stat_bin()` using `bins = 30`. Pick better value `binwidth`.
```



Scatterplot

To create a scatter plot, you need to define x and y, and then the points layer.

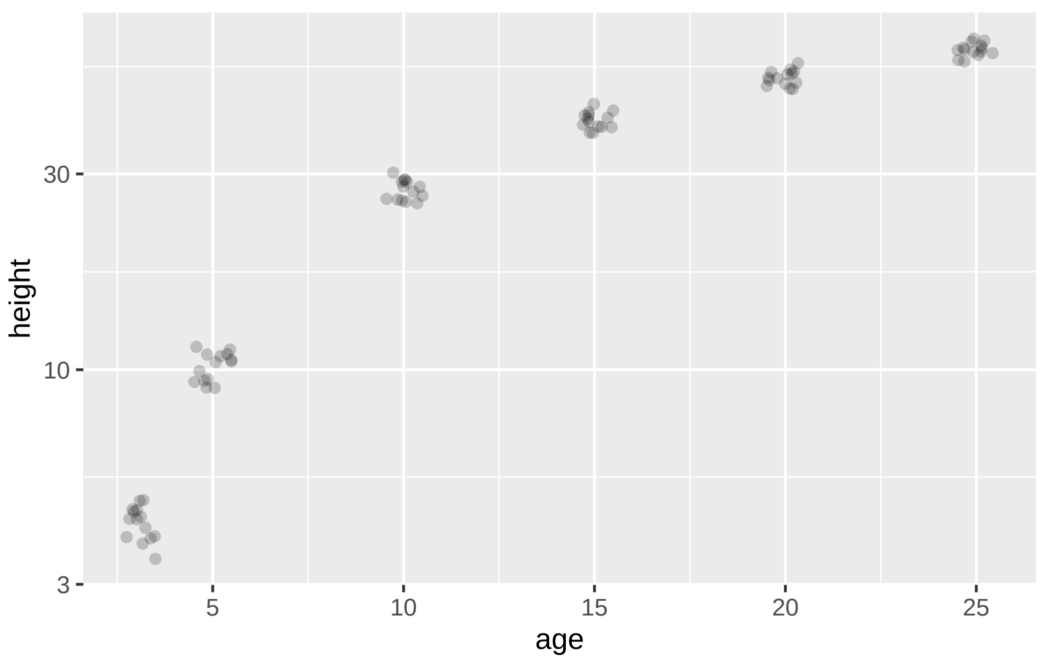
```r
ggplot(
    data = Loblolly,
    mapping = aes(
        x = age,
        y = height
    )
) +
geom_point()
```



Jitter can be used if over-plotting is a problem.

> Note: We are log-transforming the x-axis.

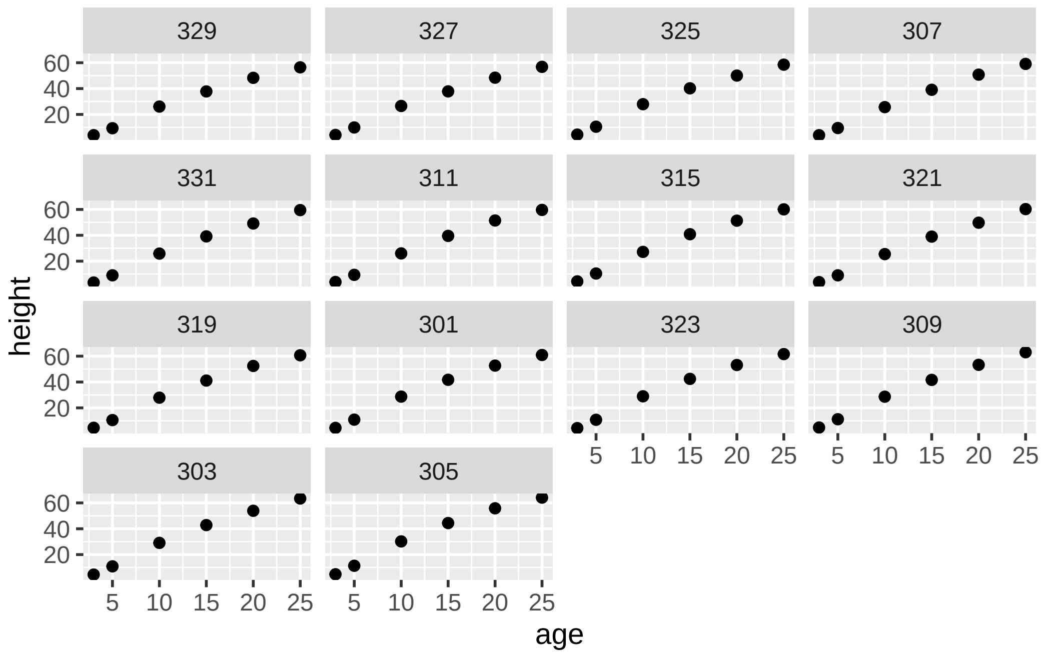
```
ggplot(
    data = Loblolly,
    mapping = aes(
        x = age,
        y = height
    )
) +
geom_jitter(
    alpha = 0.2,
    width = 0.5,
    height = 0
) +
scale_y_log10()
```



Facetting

Use facetting or panelling to create multiple plots that display different subsets of your data for clearer comparisons.

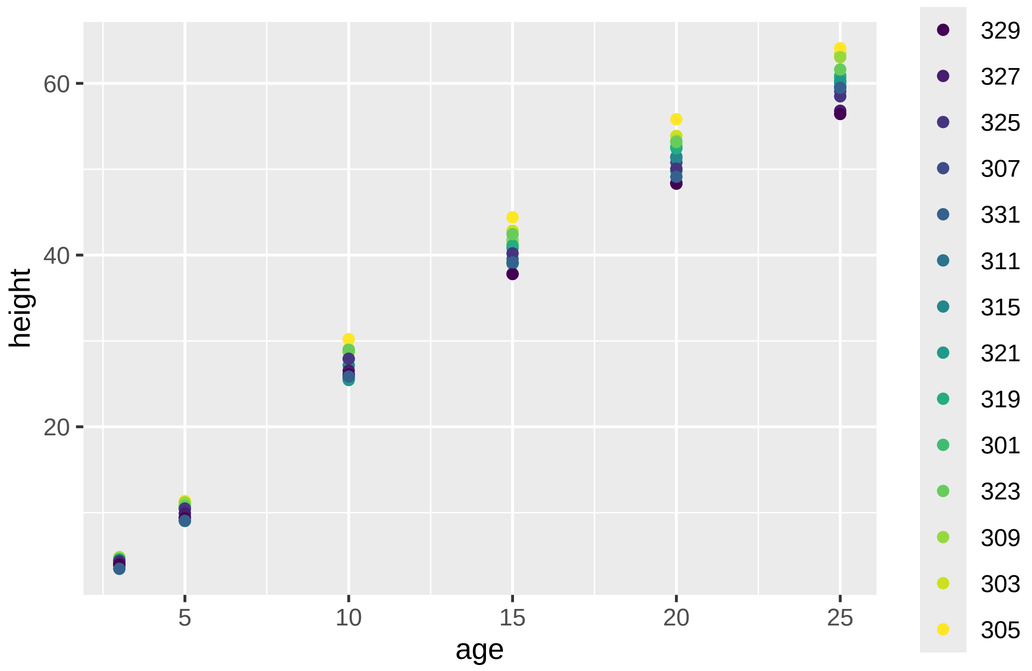
```r
ggplot(
    data = Loblolly,
    mapping = aes(
        y = height,
        x = age
    )
) +
geom_point() +
facet_wrap(~ Seed)
```



Colour

Points can be coloured **according** to a certain variable.

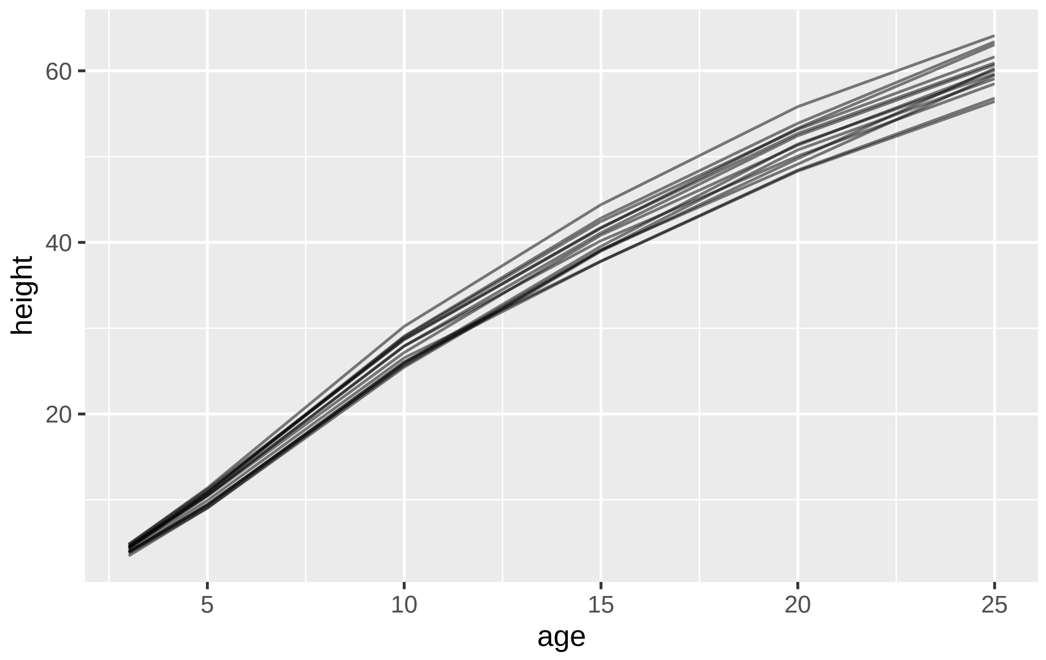
```
ggplot(
    data = Loblolly,
    mapping = aes(
        y = height,
        x = age,
        colour = Seed
    )
) +
geom_point()
```



Lines

Lines can be drawn by connecting points for each level of a variable by using the "group" argument.

```
ggplot(
    data = Loblolly,
    mapping = aes(
        y = height,
        x = age,
        group = Seed
    )
) +
geom_line(alpha = 0.5)
```

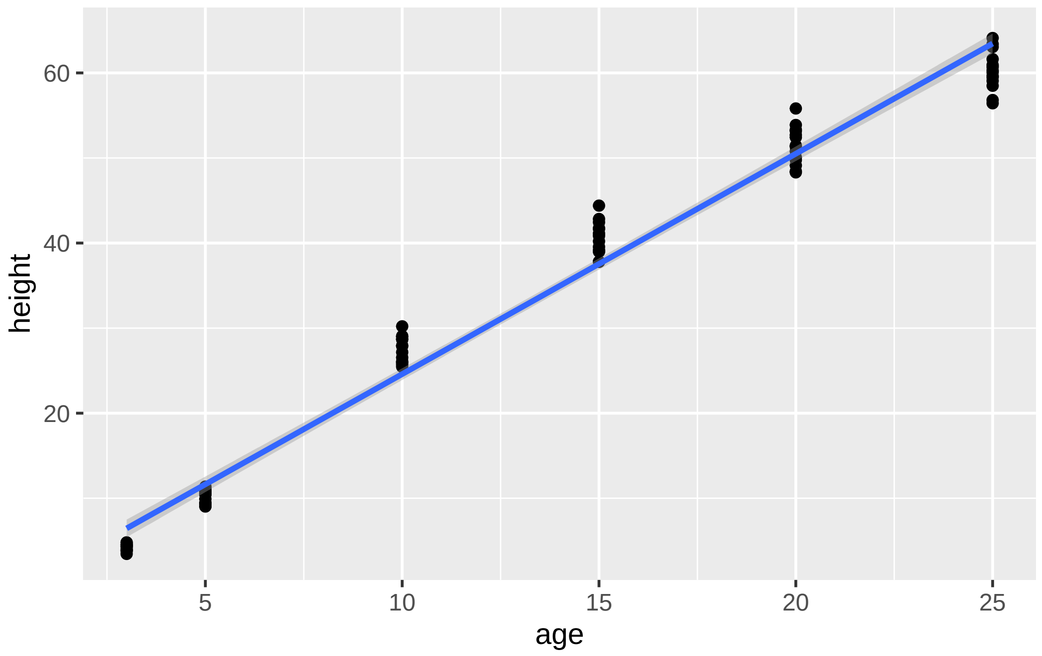


Smoothing Methods: Linear Regression

Smoothing line can reveal trends in your data.

```
ggplot(
    data = Loblolly,
    mapping = aes(
        y = height,
        x = age
    )
) +
geom_point() +
geom_smooth(method="lm")
```

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



❗ Important

The order of the levels is important. If we swap the last two layers of the previous graph, the points become partly obscured by the regression lines drawn on top.

Save a Plot

```
ggsave(
    filename = "Loblolly_6.png", # Path
    plot = gg.tmp # Graph object
)
```

Markdown

Markdown is a lightweight markup language used to format plain text into structured documents using simple symbols like asterisks and hashtags. It allows users to create rich content—such as headers, lists, and links—that remains easily readable in its raw form and can be seamlessly converted to HTML.

R Markdown

R Markdown extends standard Markdown by integrating executable code chunks, allowing you to combine narrative text with live results from R, Python, or SQL. It is a powerful tool for reproducible research, enabling the automatic generation of data-driven reports in formats like PDF, HTML, or Word.

How to Start

Markdown documents do not need a specific header. R Markdown documents begin with a YAML header. The basic form is:

```
---
title: ""
author: ""
date: ""
output: html_document
---
```

Structuring the Document

Headings

Headings for sections can be created using the `#`. You can use up to 6 heading levels.

```
# Heading level 1
## Heading level 2
### Heading level 3
```

Parallel Sections

Sections can be organised in parallel: the content is put under horizontal tabs instead of vertically (less scrolling).

```
## Results {.tabset}
### Tables
...
### Plots
...
## Discussion
```

The next section heading equal (same number of `#`) or higher (= fewer `#`) than where `{.tabset}` was used turns tabset mode off.

> Caution
>
> Text search will not find any results in tabs that are not on display!

YAML

The YAML at the beginning allows you to add more options to your file such as a table of content or define multiple output formats.

```
output:
    html_document:
        number_sections: true
        toc: true
        toc_float:
            collapsed: false
            smooth_scroll: true
```

Cross References

Headings can be labelled and referenced via a link. After the heading, on the same line, add an anchor (also called target or link): `{#marker}`

> Note: Markers in tabs that are not on display can not be reached.

Style Elements

Style Elements

As mark-up, only italic and bold are possible.

```
_italic_
**Bold**
```

> Note: Bold and italic can be combined.

Subscript and Superscript

- Subscript: a tilde ~ before and after an expression.
- Superscript: a caret ^ before and after an expression.

Blockquotes

To create a blockquote, add an angle bracket `>`in front of a paragraph. Multiple paragraphs require a `>` on the empty line.

```
The following quote is from the lost novel of Kleist:
> Lorem ipsum dolor sit amet, consectetur adipiscing eo,
> sed do eiusmod tempor incididunt ut labore et dolore:
>
>> Exercitation ullamco laboris nisi ut aliquip ex ea
>> commodo consequat. Duis aute irure dolor in reprehensi
>
> Finis terrae hic advenit.
```

Note: Nested blockquotes (`>>`) appear only in HTML.

Ordered Lists

Ordered lists start with a number, followed by a period.

```
1. First item
2. Second item
3. Third item
```

Nested ordered lists can be created by inserting 4 spaces for indentation.

Ordered Lists

For an unordered list, use hyphens -, plus signs +, or asterisks *.

```
- Mangos
- Oranges
- Apples
    - Granny Smith
    - McIntosh
    - Gala
```

Tables

Tables are created by stating the column names, separated by the pipe sign | and three or more hyphens — on line 2, thus defining the header.

```
A nice table

| Col 1     | Col 2      |
| :-------- | :--------- |
| word      | number     |
| booktitle | Quite long |
```

A colon :– at the left end of the hyphens aligns the text in the column to the left (default). A colon –: at the right end of the hyphens aligns the column to the right. Colons at both ends :–: center the text.

Images

Images with a caption can be inserted anywhere. Syntax: `![caption ](filename "tooltip title ")`

Footnotes

A footnote appears as a superscript number and the text is added at the bottom of the page. Within brackets, a is caret followed by an identifier: `[^identifier ]`.

Links

To insert a link, use the syntax `[link text ](URL)`

Mathematical Expressions

R Markdown offers many of LATEX's mathematical typesetting features.

- Inline-syntax: Enclose the expression within two dollar signs `$ $`
- Formula-syntax: Use two dollar signs on each side `$$ $$`

LaTeX Code	Symbol
`a^2`	a^2
`e^{c + d}`	e^{c+d}
`x_i`	x_i
`y_{j, k}`	$y_{j,k}$
`\ldots`	$\ldots$
`a \cdot b`	$a \cdot b$
`x_1 \cdots x_n`	$x_1 \cdots x_n$

LaTeX Code	Symbol
`\sum_{i = 1}^n`	$\sum_{i=1}^n$
`\prod_{i = 1}^n`	$\prod_{i=1}^n$
`\frac{a + b}{cd}`	$\frac{a+b}{cd}$
`\leftarrow`	$\leftarrow$
`\rightarrow`	$\rightarrow$
`\alpha \ldots \omega`	$\alpha \ldots \omega$
`A \ldots \Omega`	$A \ldots \Omega$

Escapes, Displaying Code, Line Breaks, Comments, Deep Dive

The Great Escape

To display the signs used for controlling Markdown, use the backslash \ as escape character.

```
# Header 1
\# Just a hashtag
```

Display code

Within text, use backtick marks ‘ before and after: `print("hello World!")`. Or use “‘ for a code cell.

Commments

Comments are enclosed in `<!-- -->` and can be multiline.

```
<!-- This is an out-commented line -->
<!--## Beginning
Stately, plump Buck Mulligan came from the stairhead, bearing a
bowl of lather on which a mirror and a razor lay crossed.
-->
```

knitr

A kntr file (.Rmd) contains R Markdown code and R code.

In-line R Code

In-line R code is enclosed in an opening and closing backtick

R Chunks

An R chunk begins with “‘{r } and ends with “‘

Chunk Options

Option	Description
`eval`	Evaluate/run the chunk of R code?
`include`	Display the *result* of the R code?
`echo`	Display the *R code itself* in the document?
`message`	Show messages in the document?
`warning`	Show warning messages in the document?
`error`	Show error messages in the document?

Line Numbers for Code Blocks

With the chunk option `attr.source = ".numberLines"` you can add line numbers to code. In the YAML section at the beginning of the R Markdown document, a syntax highlight theme needs to be provided (notice the mandatory indentation with 2 and 4 spaces).

```
output:
    html_document:
        highlight: tango
```

Note

For a list of syntax highlight themes, see: https://bookdown.org/yihui/rmarkdown/appearance-and-style-1.html

Set General Options

To set knitr options for the entire document, use the function `opts_chunk$set()`

```
library(knitr)
opts_chunk$set(echo = FALSE, include = FALSE) # Set you default options
```

Tables

Creating tables is a bit of a challenge. knitr includes the function `kable()`, which produces simple tables.

```
iris$noise <- rnorm(n = nrow(iris))
kable(iris[1:10, -5], digits = 4,
align = c("llccr"),
caption = "Tab. 1: First 10 Observations
in the Famous Iris Data Set.")
```

Plots

To include plots in the document, the chunk options need to be set to `include = TRUE`.

```
plot(sin, -pi, 2 * pi)
title(main = substitute(paste("Sine from ", -pi, " to 2", pi)))
```

Caching

The option cache allows knitr to store the results of the chunk. The next time the file is knitted, the results are obtained from the cache (files on your computer) instead of running the R code again: `cache = TRUE`.

> Note: Cached results can save a lot of time!

The option `dependson` allows to define dependencies. Overrules the option cache: `dependson = "Chunk-1"`.

Coding Style Guidelines

While it may be true that "ugly code runs," it is also difficult to read, frustrating to extend, and tiring to debug. All code must be human readable. There are a lot of style guides. Just pick the one you like best.

Comments

The more the better! Think about others reading your code. Think also about your future self. It's worth the effort! In the end, it safes time and avoids errors.

Note

Do not explain what the code is doing when it is already self-explanatory.

```r
# Read Data
readr::read_csv("data.csv")
```

Instead, focus on the why—documenting the intent or technical reasoning behind specific choices:

```r
# Use lazy loading as the dataset size is expected to grow significantly
readr::read_csv("data.csv", lazy = TRUE)
```

Modeling

Continuous Variables

Linear effects

Very intuitive and simple to interpret. Straight lines as humans wish, nature is often more complex.

$$y = \beta_0 + \beta_{\text{Edu}} \cdot x_{\text{Edu}} + \epsilon$$

Log-linear effect

Intuitive (the effect is linear in the log-scale). Work well in many situations, especially with "amounts". The coefficient interpretation is hard: trade-off between model fit and interpretability.

Quadratic effects

Less intuitive than linear or log-linear. Work well in many situations. Are often a fair approximation of true non-linear effects. The coefficient interpretation is harder. One more parameter needs to be estimated: trade-off between model fit and interpretability.

Discrete Predictors as a Factor

Fairly intuitive. Makes no assumptions on the form of the effect. Only applicable to discrete variables. Expensive in terms of parameters estimated.

Smoothers

Very flexible as they make no assumptions on the form of the effect. Somehow expensive in terms of parameters estimated coefficients are not directly interpretable are implemented in GAMs (an extension of the Linear Model).

Deriving new predictors

Enhances interpretabilty and acceptance. May improve model fit (as closer to reality). It is difficult to choose the “best” new predictor. Leads to temptation.

Maps

Basic Elements of Map

- Polygons: closed shapes such as country borders
- Lines: linear shapes that are not filled
- Points: used to specify positions

Mapping Pagages

```r
library(ggplot2)
library(dplyr)
library(maptiles)
library(sf)
library(tidyterra)
```

Download tiles

Define the map area to download the tiles. Easy option:

- Convert your data (e.g., latitude and longitude of points of interest) into a simple feature object, which can include just the corners of the map area.
- Use the simple feature object to download the tiles for your area of interest.

```r
d.lat.long.sf <- st_as_sf(
    data.frame(
        Latitude = c(46.18166, 46.1652),
        Longitude = c(8.92637, 8.96011)
        ),
        coords = c("Longitude", "Latitude"),
        crs = "+proj=lonlat"
)
## Download tiles
example.map <- get_tiles(d.lat.long.sf,
provider = "OpenStreetMap")
## Get limits
bbox.limits <- st_bbox(d.lat.long.sf)
```

Plot a map

```r
ggplot() +
geom_spatraster_rgb(
    data = example.map,
    maxcell = 5e6
) +
coord_sf(
    xlim = bbox.limits[c(1, 3)],
    ylim = bbox.limits[c(2, 4)]
)
```

R Programming

Functions

Set of statements organised together to perform a specific task. In R, there are built-in functions, e.g., summary() or mean() As users, we can specify our own functions. Function components:

- Function name: The function is stored in the R environment as an object with this name.
- Arguments: An argument is a placeholder. When a function is invoked, you pass a value to the argument.
- Arguments are optional; a function may contain no arguments.
- Arguments can have default values.

- Function body — The function body contains a collection of statements that defines what the function does.
- Return value — The return value of a function is the last expression in the function body.

```r
my.mean <- function(vector) {
    N <- length(vector)
    my.sum <- sum(vector)
    return(my.sum / N)
}
my.vec <- c(2, 4, 3, 5.5)
my.mean(my.vec) ## returns a number
```

```
[1] 3.625
```

For-loops

Used when we want to perform a task multiple times of iterate over something (e.g vector).

```r
for (i in 1:3) {
    print(paste0("iteration number: ", i))
    vec <- runif(10)
    print(my.mean(vec))
}
```

```
[1] "iteration number: 1"
[1] 0.3902467
[1] "iteration number: 2"
[1] 0.5512646
[1] "iteration number: 3"
[1] 0.4459961
```

`apply()` function

The apply function is based on the same idea as for-loops but optimized for data frames. For each row / column, applies the statement.

```r
## Create a 2 x 4 matrix
mat.runif <- matrix(runif(8), nrow = 2, ncol = 4)
## Apply my.mean to each row
apply(mat.runif, MARGIN = 1, FUN = my.mean)
```

```
[1] 0.4853973 0.6861863
```

If-else statments

Perform assignments depending on conditions. If only one condition (either TRUE or FALSE).

```r
num <- 6

ifelse(num < 6,
    yes = "less than 6",
    no = ifelse(num > 6,
        yes = "greater than 6",
        no = "equal to 6")
    )
```

```
[1] "equal to 6"
```

Shiny App

R Shiny is an open-source R package that enables the creation of interactive web applications directly from R without requiring HTML, CSS, or JavaScript knowledge. It combines the computational power of R with a reactive user interface, allowing users to explore data and visualize results in real time through a browser.

Components

- Front end (what the user sees): The UI. defines the 'look' of the app (e.g. text, inputs, outputs, etc.)
- Back end logic (app's behaviour): the server() function. Contains instructions controlling the outputs, performs all calculations via R command.

The function shinyApp() combines both UI and Server.

```r
library(shiny)
## front end
ui <- fluidPage(
## nested UI functions
)
## back end
server <- function(input, output, session) {
## logic how output is generated, input processed, etc.
}
```

```r
## runs app by combining UI and server
shinyApp(ui, server)
```

API

An API (Application Programming Interface) is a set of defined rules and protocols that allows different software applications to communicate and exchange data with each other. It acts as an intermediary, enabling one system to request services or information from another without needing to understand its internal code.

What is a API

- Sending a Request: You initiate a request to the API, specifying what information or action you need.
- Processing the Request: The API forwards your request to the server, which processes it and retrieves the necessary data.
- Receiving a Response: The server sends the response back to the API, which then delivers the information to you.

Structure

- Endpoint: The specific digital location (URL) where the API receives requests.
- Method: The HTTP verb (GET, POST, PUT, DELETE) that tells the API what action to perform.
- Headers: Key-value pairs that provide extra context, such as security tokens or data formats.
- Body: The payload or data sent to the server, typically formatted in JSON.