R-Bootcamp - Notes

Nils Rechberger

2026-01-01

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").

```
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.

```
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.

```
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.

```
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
```

```
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.

```
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.

```
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     SI2      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

```
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.

```
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`.

```
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.
>
> ```
> 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
    )
)
```

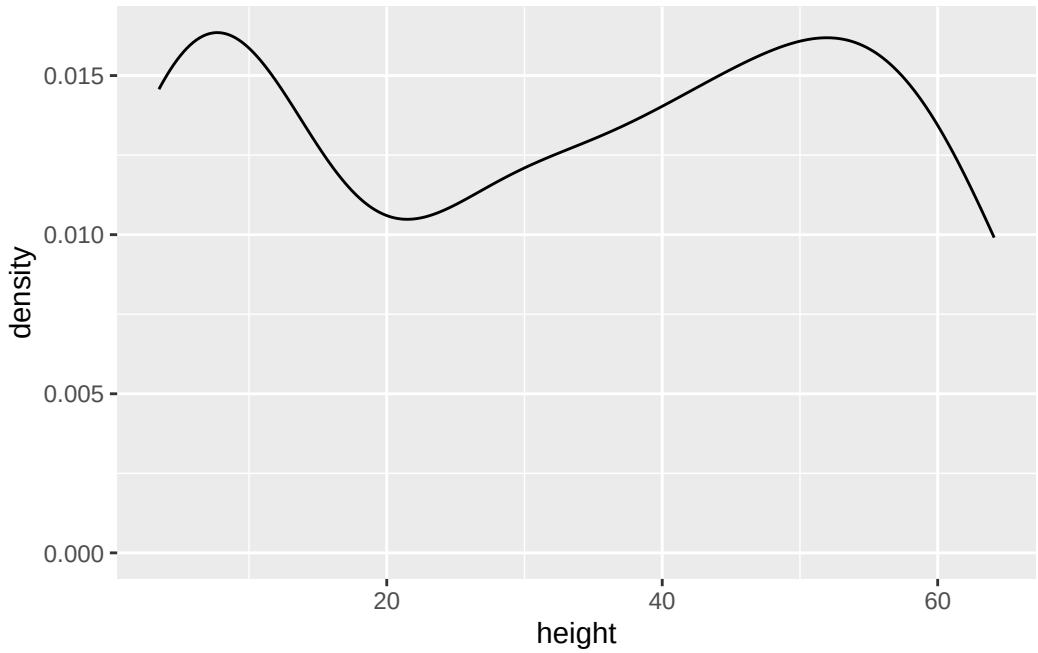
Afterwards, you add layers with the "+" symbol.

```
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.

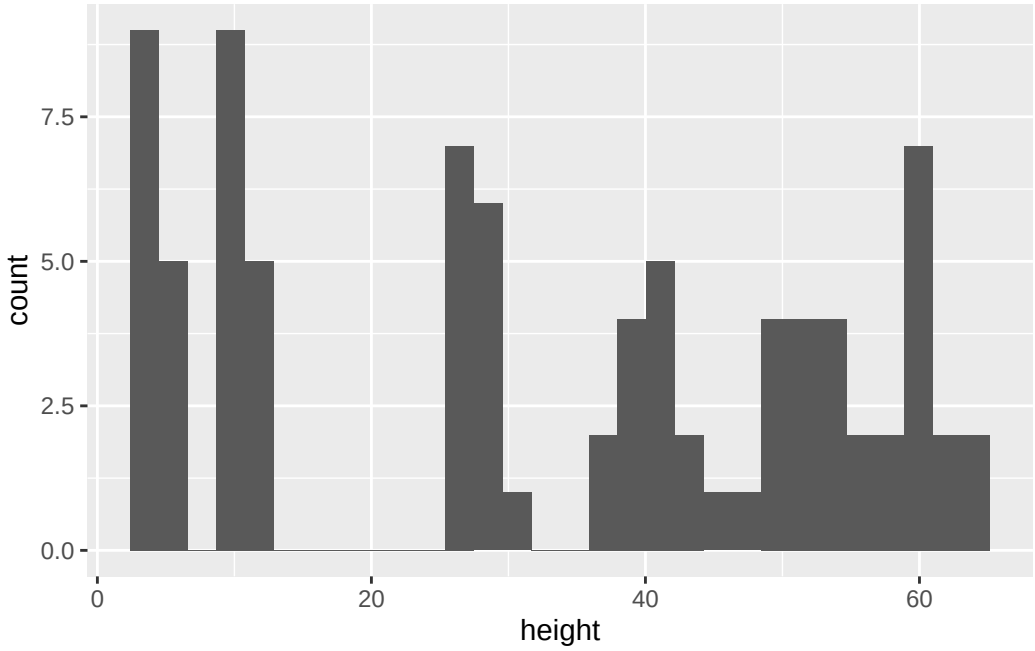
```
ggplot(
    data = Loblolly,
    mapping = aes(
        x = height
    )
) +
geom_density() +
geom_rug()
```

Histogram

Instead of a density plot, you can create histograms.

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

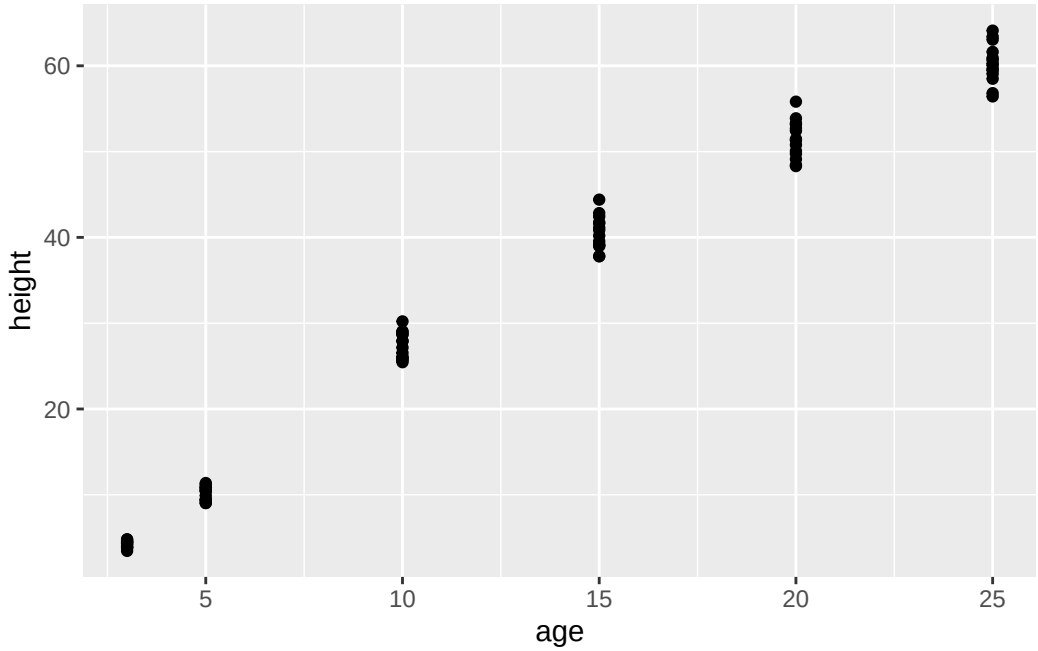
```
`stat_bin()` using `bins = 30`. Pick better value `binwidth`.
```



Scatterplot

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

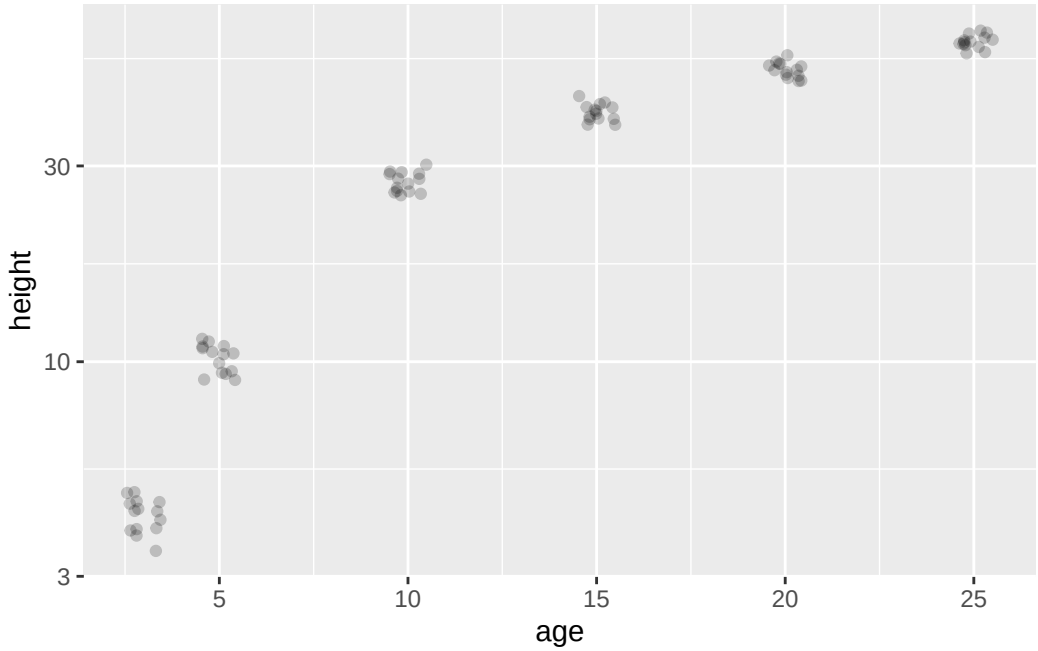
```
ggplot(
    data = Loblolly,
    mapping = aes(
        x = age,
        y = height
    )
) +
geom_point()
```



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

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

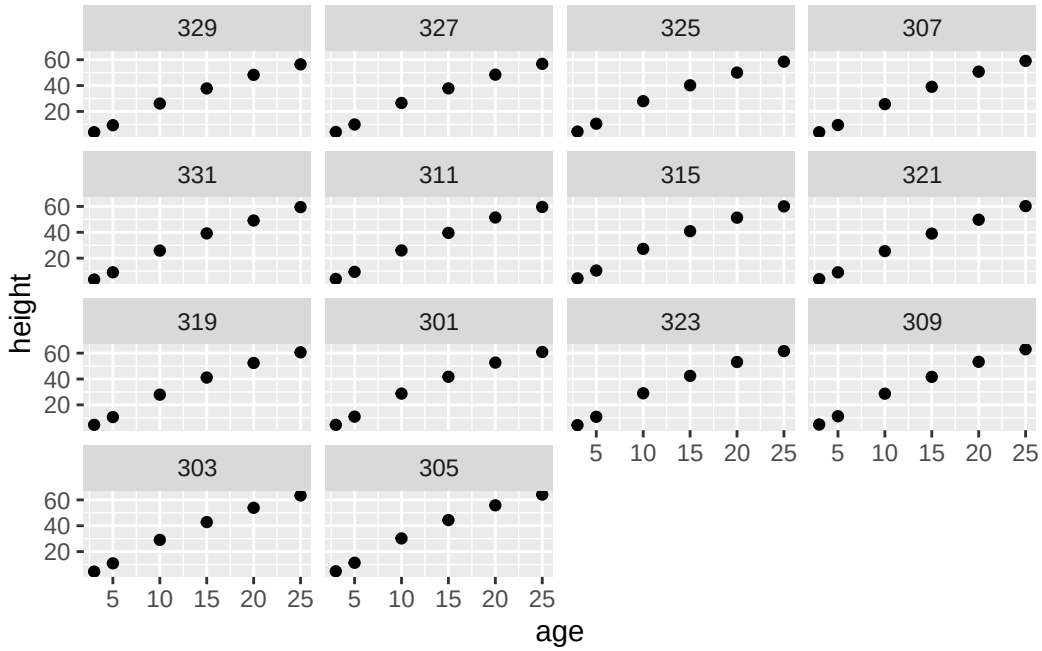
```
ggplot(
    data = Loblolly,
    mapping = aes(
        x = age,
        y = height
    )
) +
geom_jitter(
    alpha = 0.2,
    width = 0.5,
    height = 0
) +
scale_y_log10()
```



Facetting

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

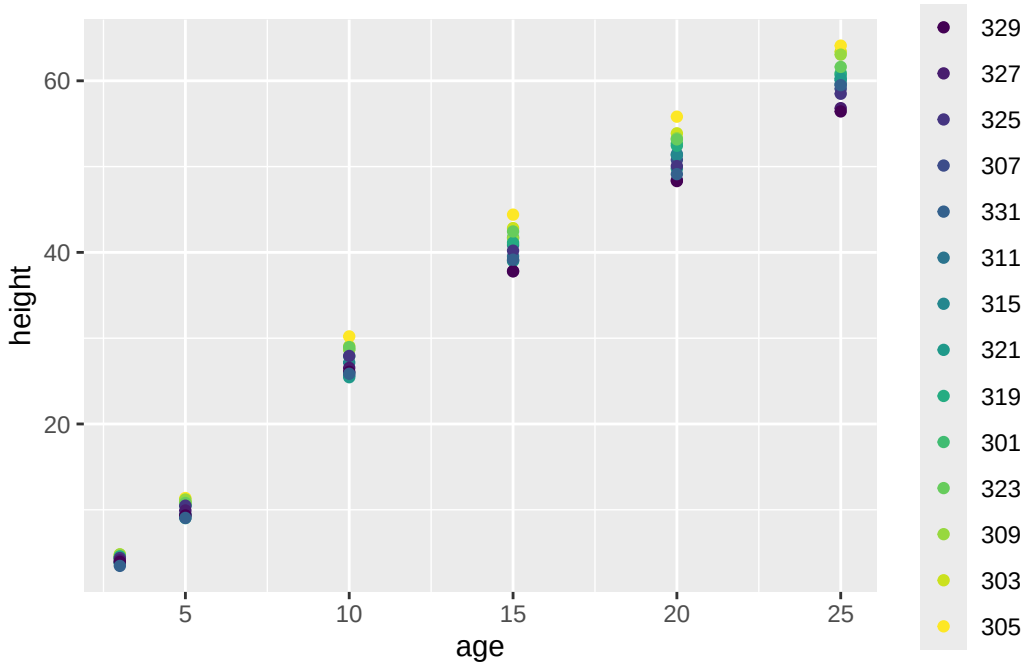
```
ggplot(
    data = Loblolly,
    mapping = aes(
        y = height,
        x = age
    )
) +
geom_point() +
facet_wrap(~ Seed)
```



Colour

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

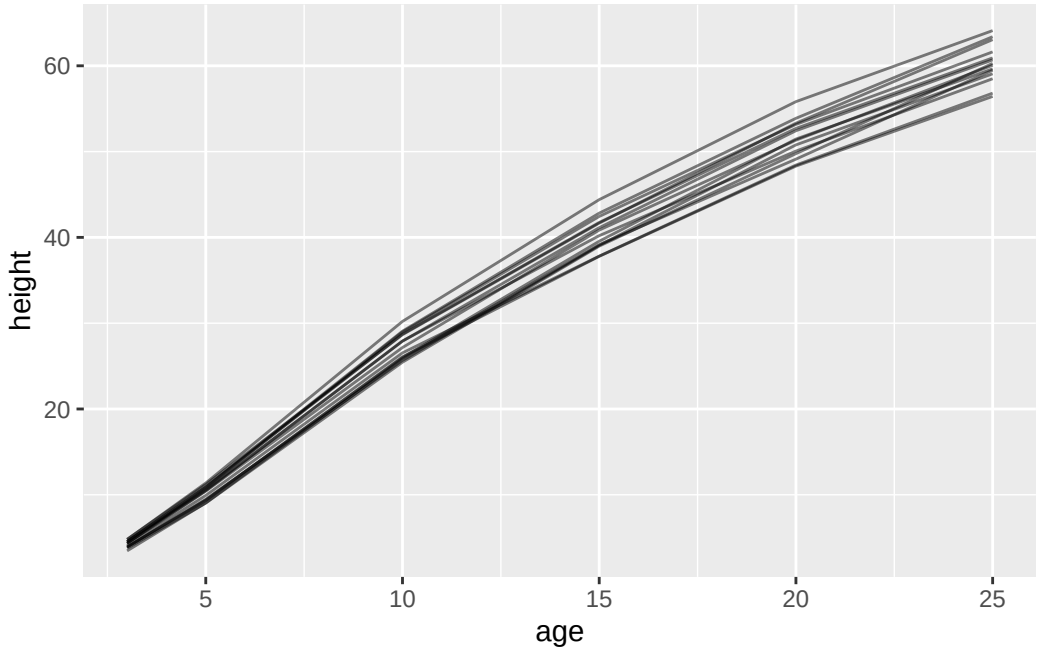
```
ggplot(
    data = Loblolly,
    mapping = aes(
        y = height,
        x = age,
        colour = Seed
    )
) +
geom_point()
```



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)
```

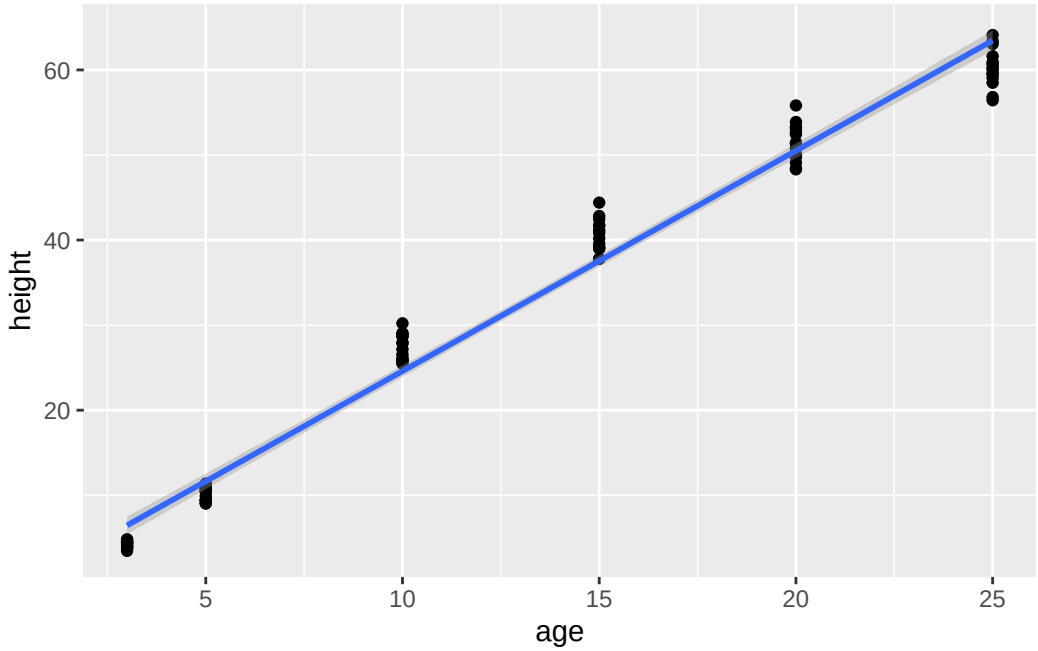


Smoothing Methods: Linear Regression

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.