Introduction to R for statistical analysis

John Bastiaansen, Gerrit Gort, Albart Coster

Introduction to R

What is R

- R is a programming environment
  - Statistics
  - Graphics
- Why R ?
  - free and runs on Windows, Linux, and MacOS
  - excellent built-in help system
  - easy to learn syntax with many built-in statistical functions
- R is a command-line system, not a point-and-click system

Downloading and Installation

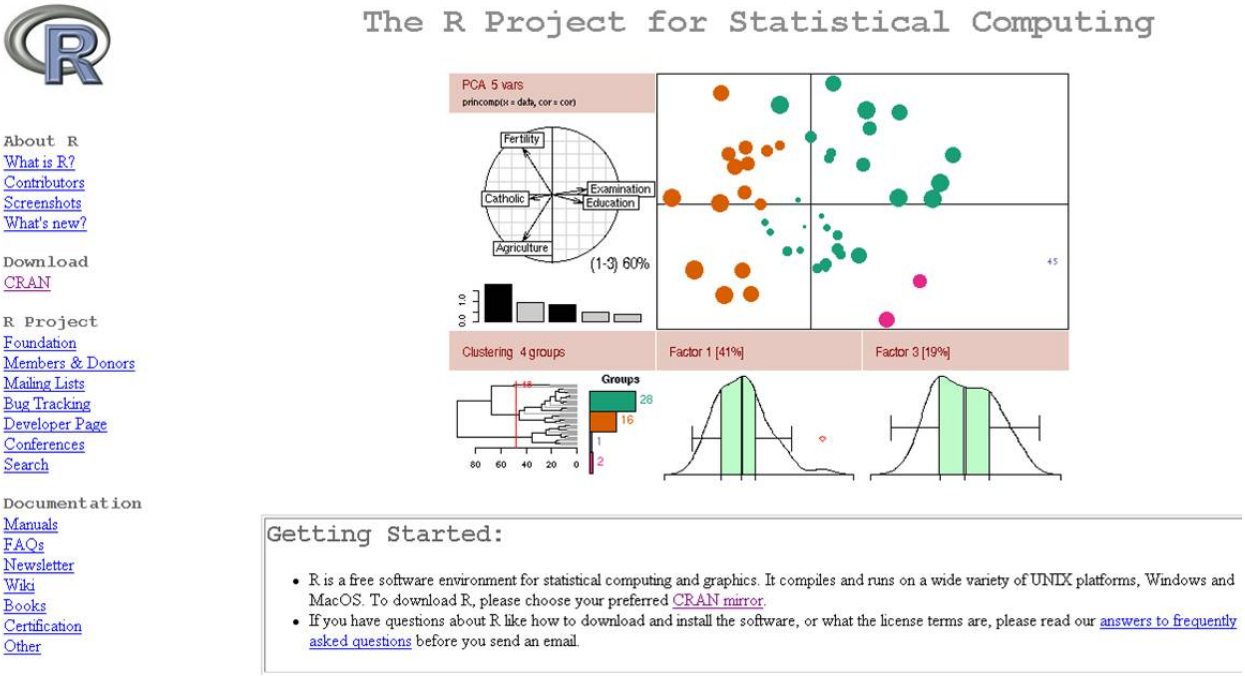


Figure 1: CRAN

Main website of the R project : http://www.r-project.org/

Download and Installation

For this course, R has been installed on your computers

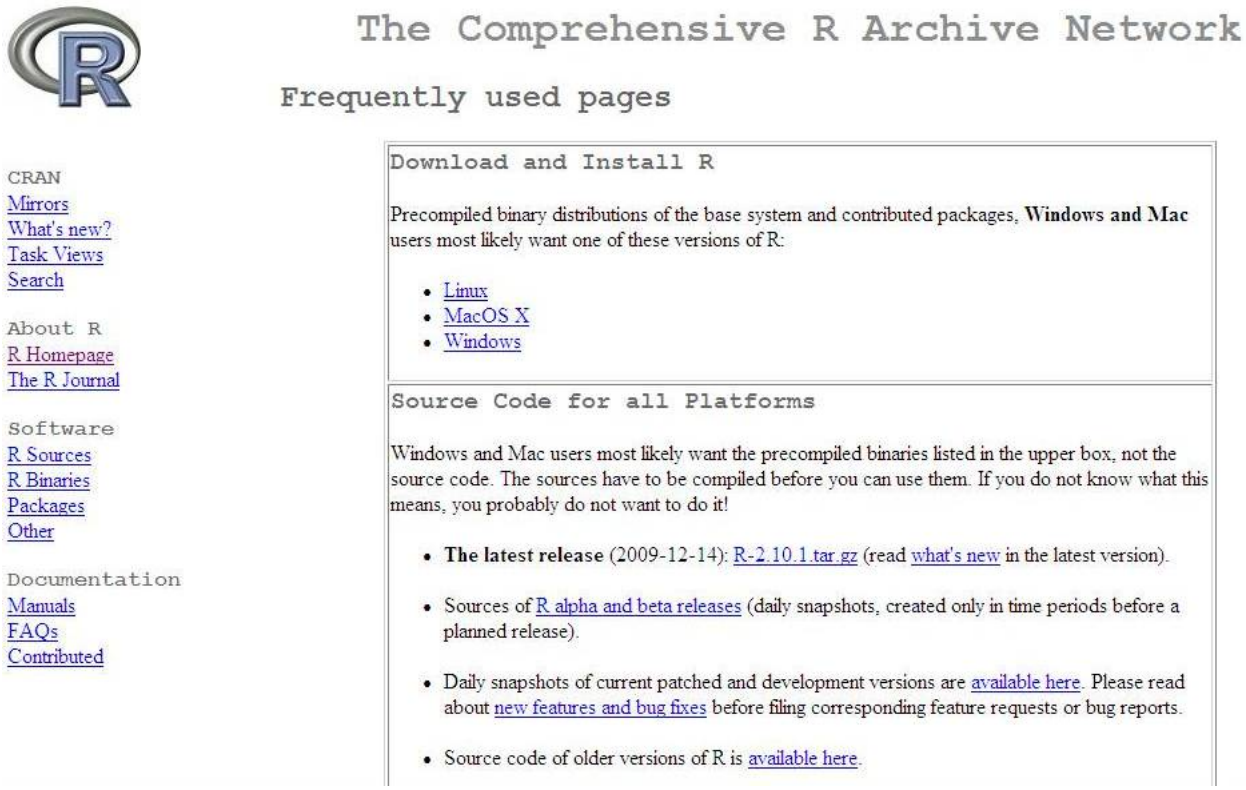


Figure 2: Downloading R

Start R

When R is installed it comes with the program's GUI (Graphical User Interface) which looks like this:

We don't use this GUI in our course, instead we use RStudio

RStudio : an Integrated Development Environment

- For this course we use RStudio
- Advantages of RStudio include
  - easy management of your workspace and files
  - highlighted code avoids mistakes
  - integrated access to R help
- Rstudio can be downloaded from : http://rstudio.org/

Using R

You can use R as a simple calculator:

```
(1+3)* (7-3)
```

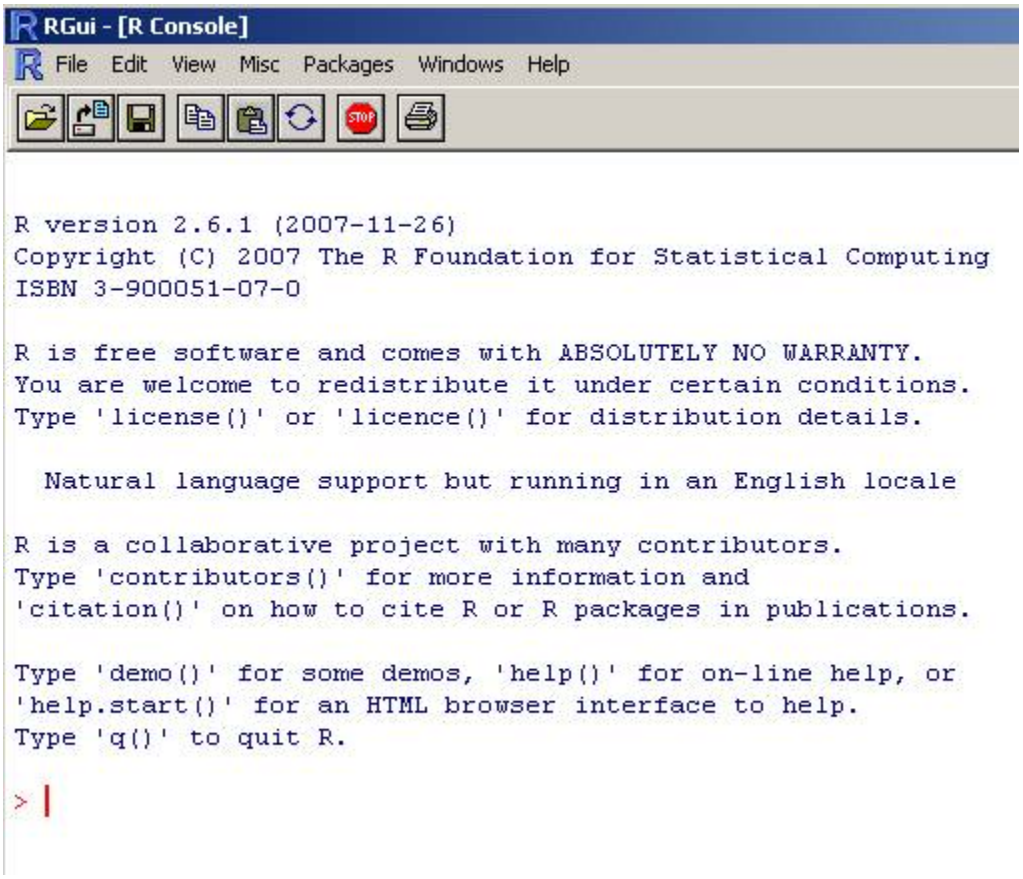


Figure 3: R GUI

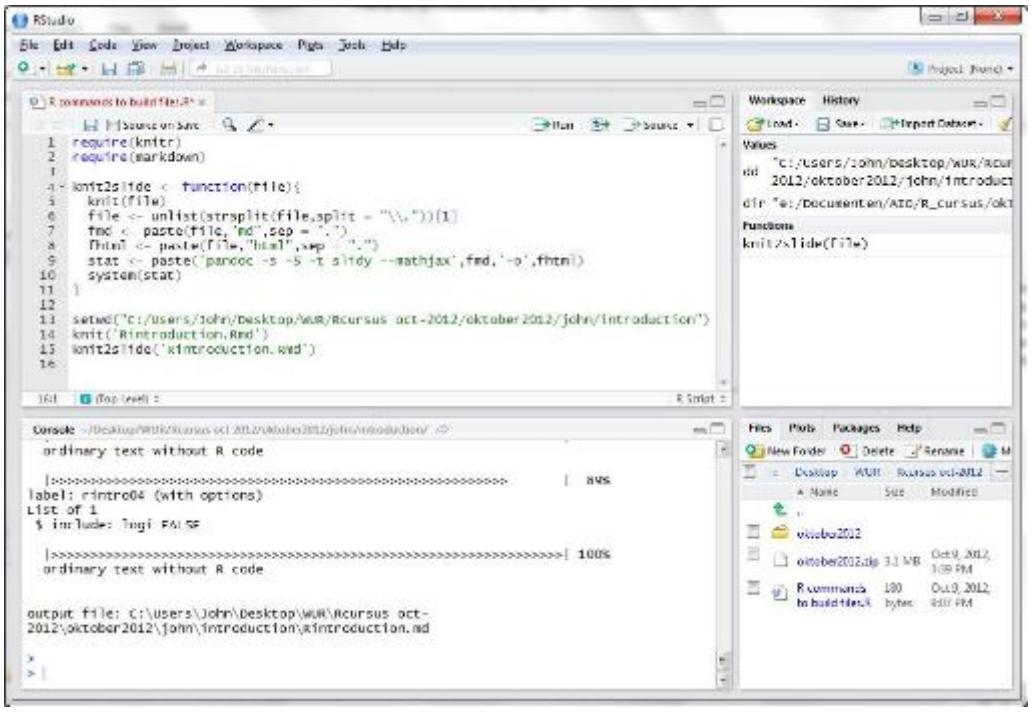


Figure 4: Rstudio interface

```
## [1] 16
```

```
1024/256
```

```
## [1] 4
```

```
5^2
```

```
## [1] 25
```

```
exp(pi)-pi
```

```
## [1] 20
```

The working directory of R

The working directory is R's default place to look for files, find it using function `getwd`:

```
getwd()
dir <- "~/Dropbox/Rcursus/mei2015/john/introduction/"
setwd(dir)
getwd()
```

```
## [1] "~/Dropbox/Rcursus/mei2015/john/introduction/"
```

The help system in R: general

Use the function `help.start()` to open a help browser page:

In RStudio the help pages open in the lower right panel

The help system in R: specific

Use the `?` operator to open a help page on an object or function:

`?print` to open help on the `print` function:

In RStudio the help pages open in the lower right panel and can be expanded to full screen view

References for learning R

- Within R
  - An introduction to R : http://cran.r-project.org/doc/manuals/R-intro.html
  - Help functions: `help.start()` , `help(object)` , `help.search('topic')`
- Free documentation and books

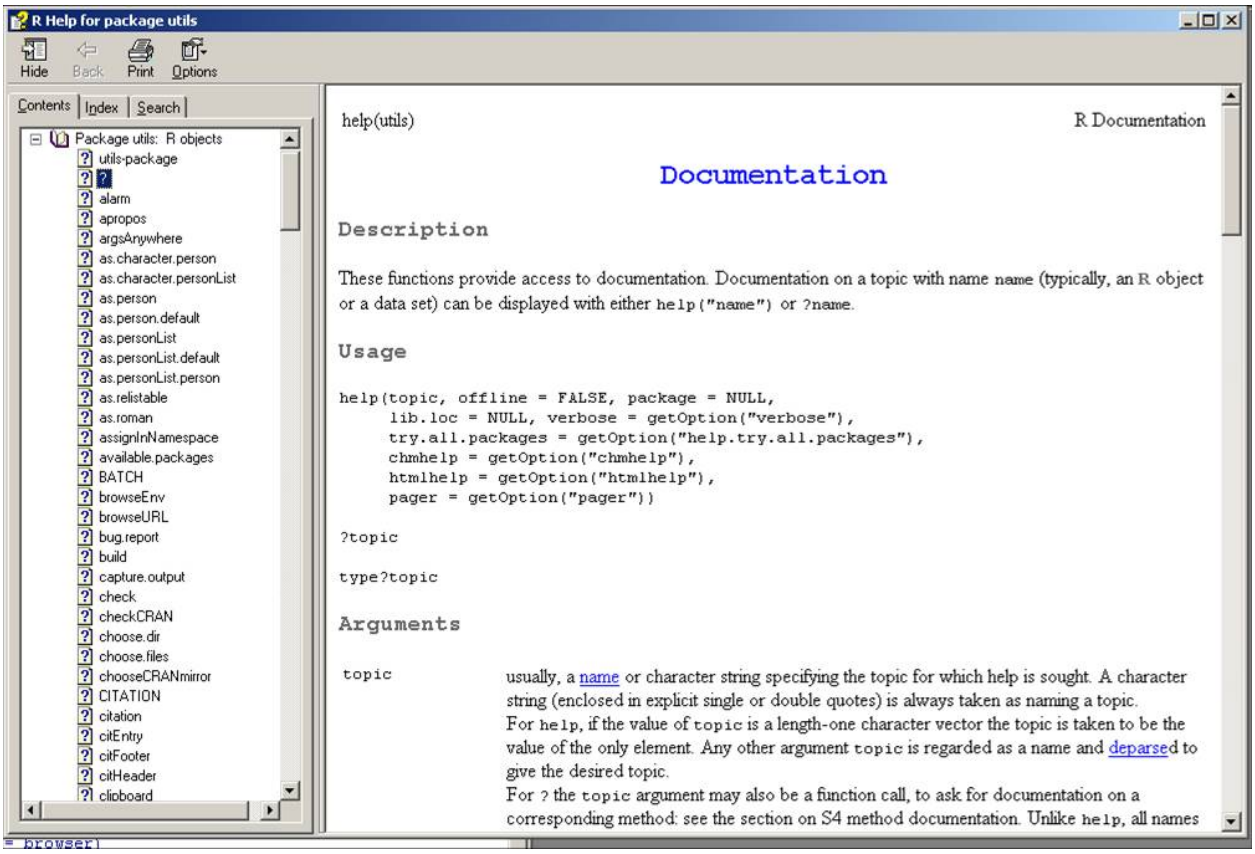


Figure 5: The help page

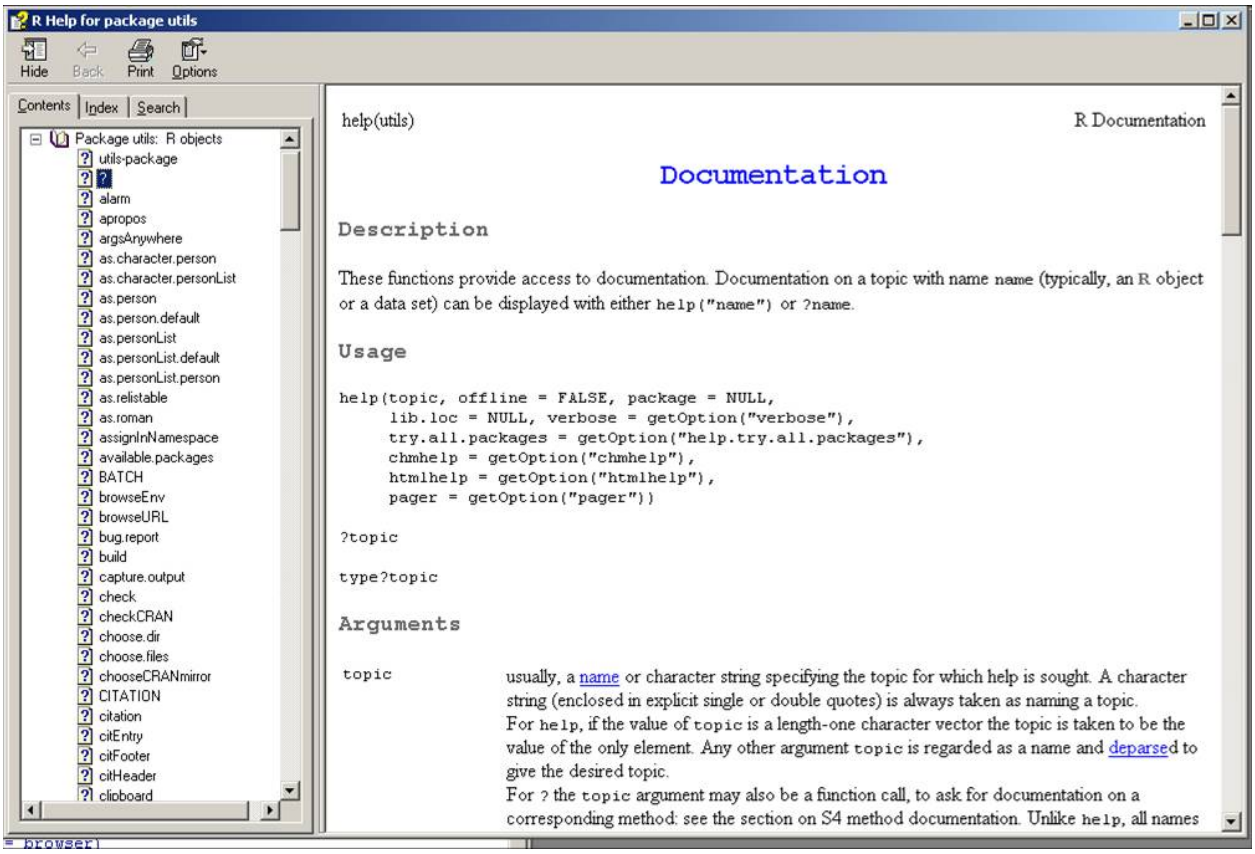


Figure 6: R help

  - Using R for data analysis and graphics, John Maindonald
  - R for beginners, Emmanuel Paradis
  - An Introduction to R, Venables
  - Software for Data Analysis: Programming with R (Statistics and Computing), John Chambers
  - Data Manipulation with R, Phil Spector
  - Introductory Statistics with R, Peter Dalgaard
  - Linear Models with R, Julian Faraway
  - Lattice: Multivariate Data Visualization with R, Deepayan Sarkar
- R mailing list : http://r.789695.n4.nabble.com
- Collection of R blogs : http://www.r-bloggers.com

Quitting R

Type `q()` and R will quit:

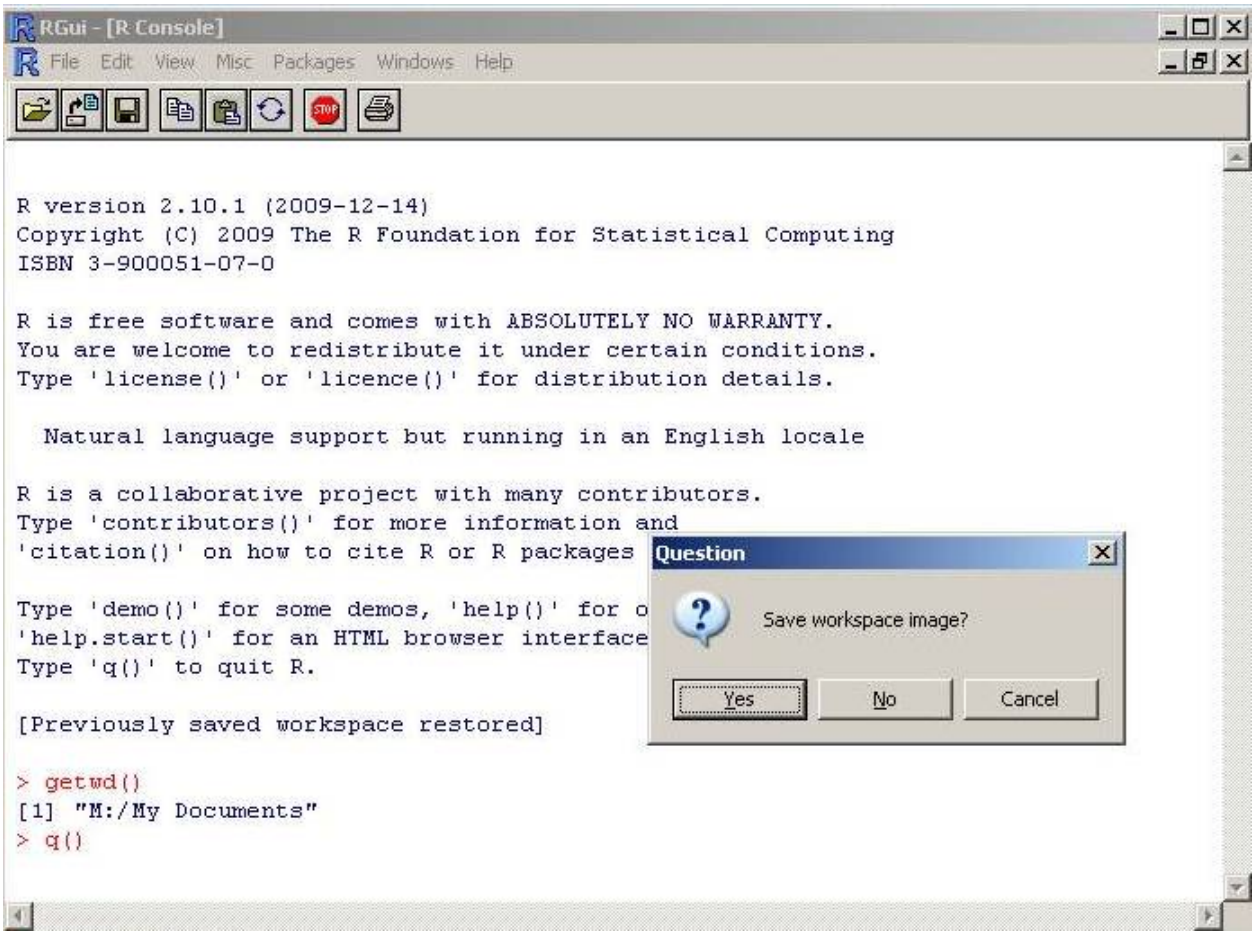


Figure 7: Quitting R

Exercises

- Work on the first page of exercises

Return to main page

- Back to the main page

R basics

R Packages

- R is organized in packages
  - A package is a collection of functions, data and other information that are somehow related
  - R packages can be downloaded from the CRAN http://cran.r-project.org/
  - R packages can be downloaded through RStudio
  - There are many R packages, and the number continues to increase
- Use function `install.packages` to install a package and subsequently function `require` to load a package:

```
install.packages("reshape")
```

Note the use of quotes around the name of the package

- After installation you need to load the package

```
require(reshape)
```

R Packages

- Installing R packages in RStudio
  - Select the tab `Packages` in the lower right panel
  - Click `Install Packages`

R Packages

- Start typing the name of the package and select the required package

R Packages

- The Console will show what is happening

Objects

- In R everything is an object:
- `class(object)` shows the class, or type, of an object
  - For simple objects, such as vectors, this is just the mode, for example `numeric`, `logical`, `character`
  - Other classes are `matrix`, `factor` and `data.frame`
- The call `ls()` shows all the objects in the workspace.

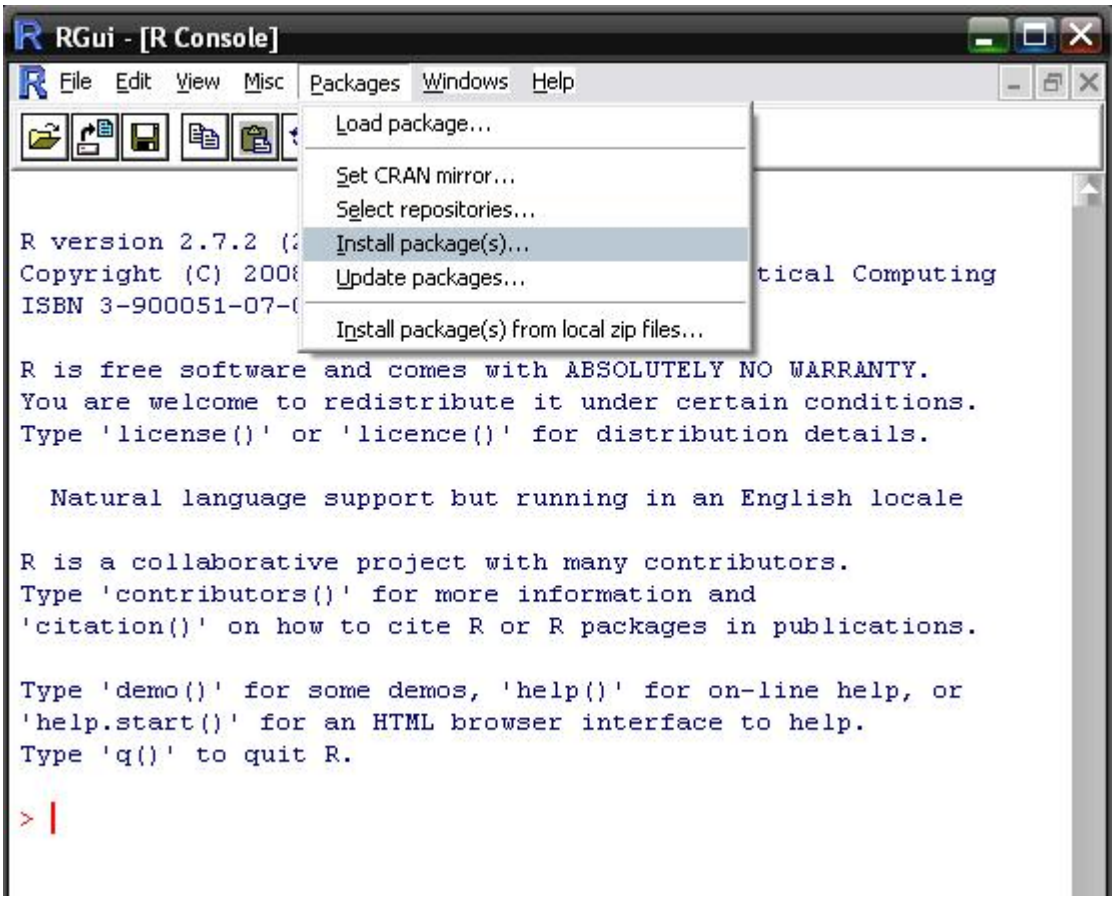


Figure 8: Packages tab

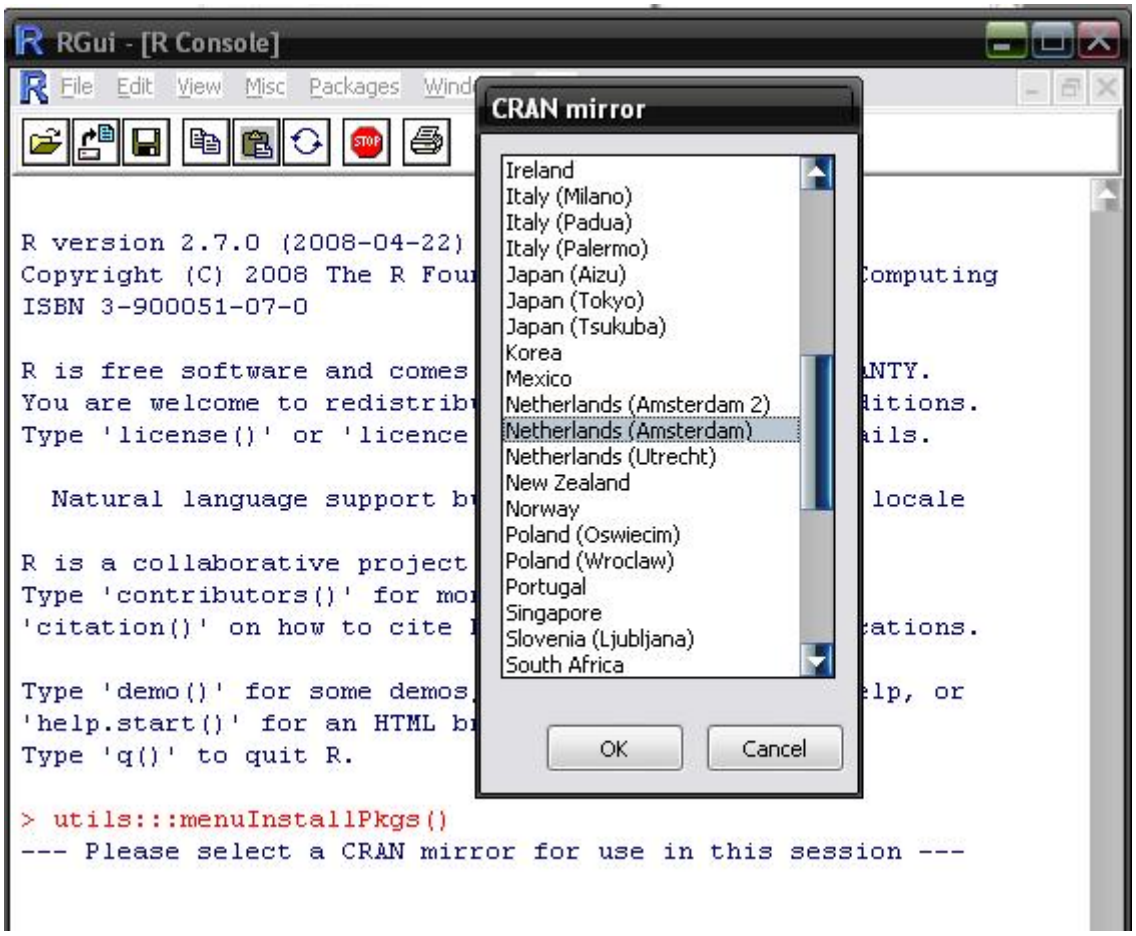


Figure 9: Select CRAN mirror

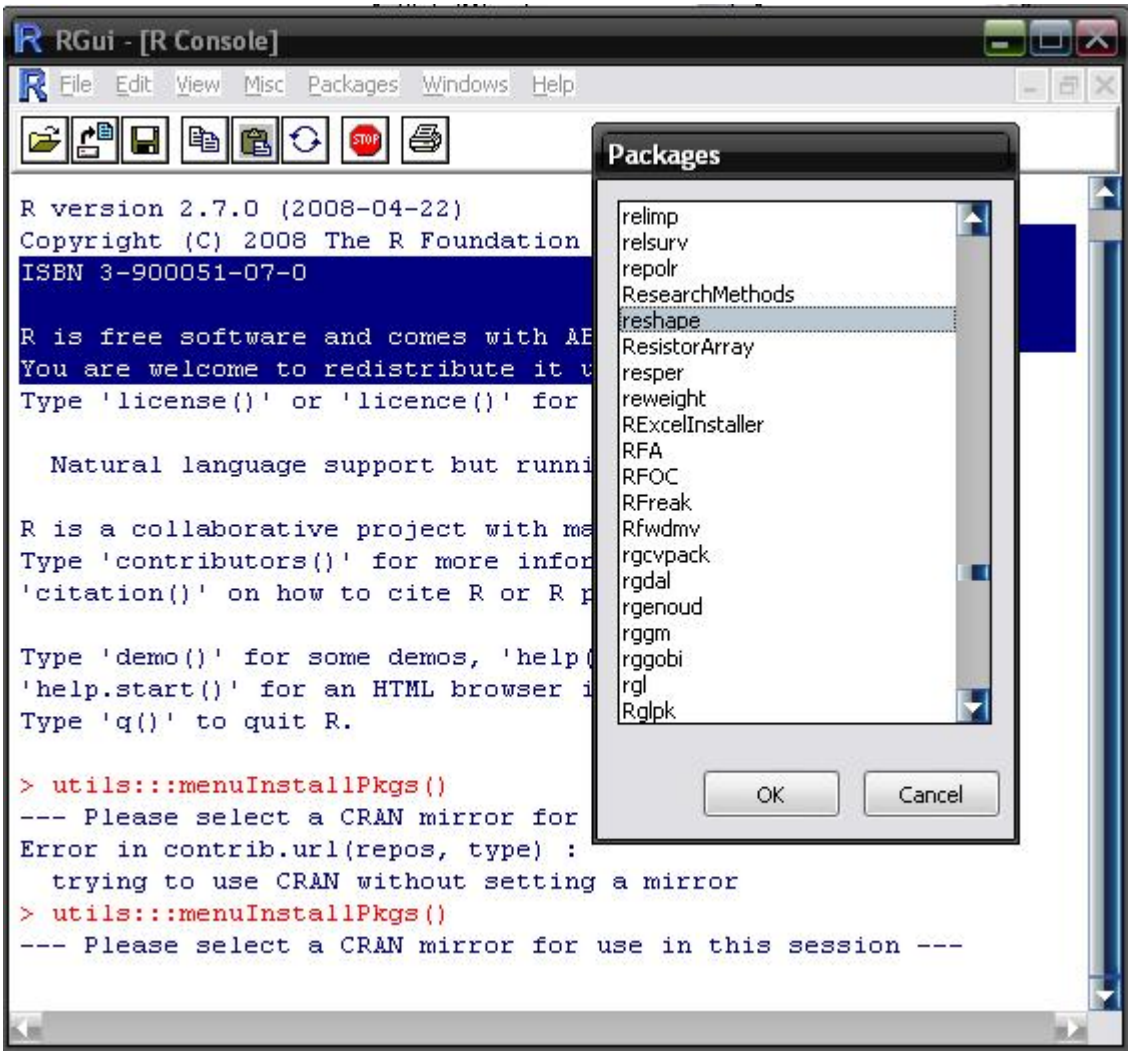


Figure 10: Select package

```
ls()
```

```
## [1] "dd"            "document.type"
## [3] "tidy.opts"
```

Object class types

We introduce these classes:

- vector
- factor
- matrix
- list
- data.frame
- function

Some or all of these will sound familiar because they are not specific to R

Vector

A vector is an ordered collection of data of the same type. We make 2 vectors with the function `c()` :

```
vector1 <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
vector1
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
vector2 <- c("a", "b", "c", "d", "e", "b", "a", "c", "b", "d")
vector2
```

```
##  [1] "a" "b" "c" "d" "e" "b" "a" "c" "b" "d"
```

Vector

Several functions exist to obtain information about vectors that were made :

```
class(vector1)
```

```
## [1] "numeric"
```

```
class(vector2)
```

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

```
length(vector1)
```

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

```
length(vector2)
```

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

Syntax notes

- The assignment operator `<-` points to the object receiving the `value` of the expression
- Often the `=` operator can be used (but not always)

```
object <- c("a", "b", "c", "d")
object
```

```
## [1] "a" "b" "c" "d"
```

```
## OR equivalently
object = c("a", "b", "c", "d")
object
```

```
## [1] "a" "b" "c" "d"
```

Functions to make vectors

We may encounter several functions to create vectors:

```
x <- 1:10
assign("x", 1:10)
x <- seq(1, 10, by = 1)
x <- seq(length = 10, from = 1, by = 1)
x <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

Choose an efficient and convenient method, the results will be the same

```
x
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Factor

A factor is an efficient way to store character data where elements are repeated, for instance when they represent grouping of the data, like for instance the name of a treatment.

```r
x1 <- c("a", "b", "c", "a", "b", "c", "a", "c")
x1
```

```
## [1] "a" "b" "c" "a" "b" "c" "a" "c"
```

```r
x2 <- as.factor(x1)
x2
```

```
## [1] a b c a b c a c
## Levels: a b c
```

```r
levels(x2) <- c("low", "mid", "high")
x2
```

```
## [1] low  mid  high low  mid  high low  high
## Levels: low mid high
```

Matrix

A matrix:

- Has 2 dimensions: rows and columns
- Can only contain data of one type, `numeric` or `character`

```r
matrix1 <- matrix(1:20, nrow = 2, ncol = 10)
matrix1
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]    1    3    5    7    9   11   13   15   17
## [2,]    2    4    6    8   10   12   14   16   18
##      [,10]
## [1,]    19
## [2,]    20
```

```r
matrix2 <- matrix(1:20, nrow = 2, ncol = 10, byrow = TRUE)
matrix2
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]    1    2    3    4    5    6    7    8    9
## [2,]   11   12   13   14   15   16   17   18   19
##      [,10]
## [1,]    10
## [2,]    20
```

- Note the effect of the extra argument `byrow = TRUE` for `matrix2`

Matrix elements

Elements in matrix can be accessed as `matrix[x,y]`:

```
matrix2
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,]    1    2    3    4    5    6    7    8    9
## [2,]   11   12   13   14   15   16   17   18   19
##      [,10]
## [1,]    10
## [2,]    20
```

```
matrix2[2, 3]
```

```
## [1] 13
```

```
matrix2[2, ]
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

```
matrix1[, 2]
```

```
## [1] 3 4
```

Accessing elements by their index will be used in the section on programming.

list

- A list is an ordered collection of objects called the components of the list
- The components of a list can be of any class:

```
record <- list(name = "Fred", spouse = "Mary", no.children = 3, child.ages = c(4,7, 9))
record
```

```
## $name
## [1] "Fred"
##
## $spouse
## [1] "Mary"
##
## $no.children
## [1] 3
##
## $child.ages
## [1] 4 7 9
```

list

- List components are accessed with the dollar operator `$` or the double square brackets operator `[[]]`
- The single square brackets operator `[]` will give a subset of the list

```
record$name
```

```
## [1] "Fred"
```

```
record$child.ages
```

```
## [1] 4 7 9
```

```
record[1:2]
```

```
## $name
## [1] "Fred"
##
## $spouse
## [1] "Mary"
```

```
record[[1]]
```

```
## [1] "Fred"
```

data.frame

- A `data.frame` is a matrix where the columns can be of different classes numeric, character, factor, ...
- The `[` and `]` operators are used as in a matrix
- Columns in a `data.frame` can be accessed with the `$`

```
vector3 <- c("a", "a", "c", "d")
matrix3 <- matrix(1:20, nrow = 4, ncol = 5)
data1 <- data.frame(vector3, matrix3)
class(data1)
```

```
## [1] "data.frame"
```

function

- Functions are the tools in R
- Every action is performed by calling a function, we have used several already
- A function is also an object
- Functions can have **arguments** within the parentheses
  - `functionName(argument1, argument2, argument3, ...)`

```
ls()
search()
matrix(1:20, nrow = 4, ncol = 5, byrow = FALSE)
```

Syntax notes

- When a function is incomplete: a forgotten parentheses or by hitting **Enter** while an expression is incomplete
- You get a `+` sign instead of the normal `>` prompt sign to indicate a continuing statement
- Either
  - Continue the function call with the remaining arguments OR
  - Press `ESC` on Windows, or `Ctrl+C` on Linux to quit inputting the command

You can write commands in one line or in more lines, as long as it is clear that they belong together:

```
matrix4 <- matrix(1:20, nrow = 4, ncol = 5)
matrix4 <- matrix(1:20, nrow = 4,
                  ncol = 5)
```

End

- Exercises R basics
  - Solutions exercises R basics
- Return to main page

Data manipulation in R

Data Manipulation and Basic Statistics in R

In this session, we will cover the following topics:

- Import data
- Subset data
- Basic statistics
- Explore data
- Computations
- Order data
- Merge data
- Save & Export results

Import data

We have seen how to create new data

- Create vectors: `c()`, `seq()`, `:`, ...
- Create matrices: `matrix()`
- Create data.frame: `data.frame()`
- ...

But: in most of cases we already have the data:

- in a file, .csv or .txt, ...
- in tables, or other files such as SPSS/SAS results

Import data

We want to get the external data into R

- We can use function `read.table`:

```
args(read.table)
```

```
## function (file, header = FALSE, sep = "", quote = "\"'", dec = ".",
##     numerals = c("allow.loss", "warn.loss", "no.loss"), row.names,
##     col.names, as.is = !stringsAsFactors, na.strings = "NA",
##     colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
##     fill = !blank.lines.skip, strip.white = FALSE, blank.lines.skip = TRUE,
##     comment.char = "#", allowEscapes = FALSE, flush = FALSE,
##     stringsAsFactors = default.stringsAsFactors(), fileEncoding = "",
##     encoding = "unknown", text, skipNul = FALSE)
## NULL
```

- You have to set the working directory to where the file (filename.txt) is stored:

```
df <- read.table("filename.txt", header = TRUE)
head(df)
```

```
##   Va V2 V3 V4 Vee
## 1  1  5  9 13  17
## 2  2  6 10 14  18
## 3  3  7 11 15  19
## 4  4  8 12 16  20
```

Import data

Importing comma separated data (.csv files)

```
df <- read.csv("filename.csv", header = TRUE)
head(df)
```

```
##   X V1 V2 V3 V4 V5
## 1 1  1  5  9 13 17
## 2 2  2  6 10 14 18
## 3 3  3  7 11 15 19
## 4 4  4  8 12 16 20
```

Reading data from the internet:

```
df <- read.table("http://rcursus.dairyconsult.nl/lime.prn",header = TRUE)
head(df)
```

```
##   type rate   pH
## 1   AL    0 5.74
## 2   AL    0 5.73
## 3   AL    0 5.75
## 4   AL    1 5.84
## 5   AL    0 5.74
## 6   AL    0 5.77
```

These functions will result in objects of class `data.frame`

Syntax notes

- Spaces in commands do NOT matter (except for readability), but CAPITALIZATION does matter
- `TRUE` and `FALSE` are logical constants, you can also use abbreviations `T` and `F`
- Both single- `''` and double-quotes `""` symbols are valid as long as the left quote is the same as the right quote
- `args(functionName)` will show the arguments for functionName with default values, try `args(read.table)`
- Argument matching
    1. match on position: `read.table("filename.txt" , TRUE , "\t" )`
    2. match by name: `read.table(file="filename.txt" , header=T , sep="\t" , ...)`
    3. Both argument matching types can be mixed in the same function `read.table("filename.txt" , header=T, "\t" , ...)`

Brackets in R

Bracket	Use
`()`	For function calls (`print(1)`) and to set priorities (`a = 3*(1+2)`)
`[]`	For indexing vectors (`a[1]`), matrices (`m[1,1]`), or lists (`l[[1]]`)
`{}`	Block delimiter for grouping commands: (`if(!exists(a)){a = 1; print(a)}`)

Build in data

- In the following sessions, we will use the `ChickWeight` dataset from R:
  - Data from an experiment on the effect of diet on early growth of chickens
  - Object `ChickWeight` is a data.frame
  - Check `?ChickWeight` for a description
- The following functions will help you to find data that comes with the R package datasets:

```
data()
```

```
data(ChickWeight)
names(ChickWeight)
```

```
## [1] "weight" "Time"   "Chick"  "Diet"
```

The `ChickWeight` data

An overview of the complete dataset:

- The dimensions of the `data.frame` :

```
dim(ChickWeight)
```

```
## [1] 578   4
```

- The names of the columns:

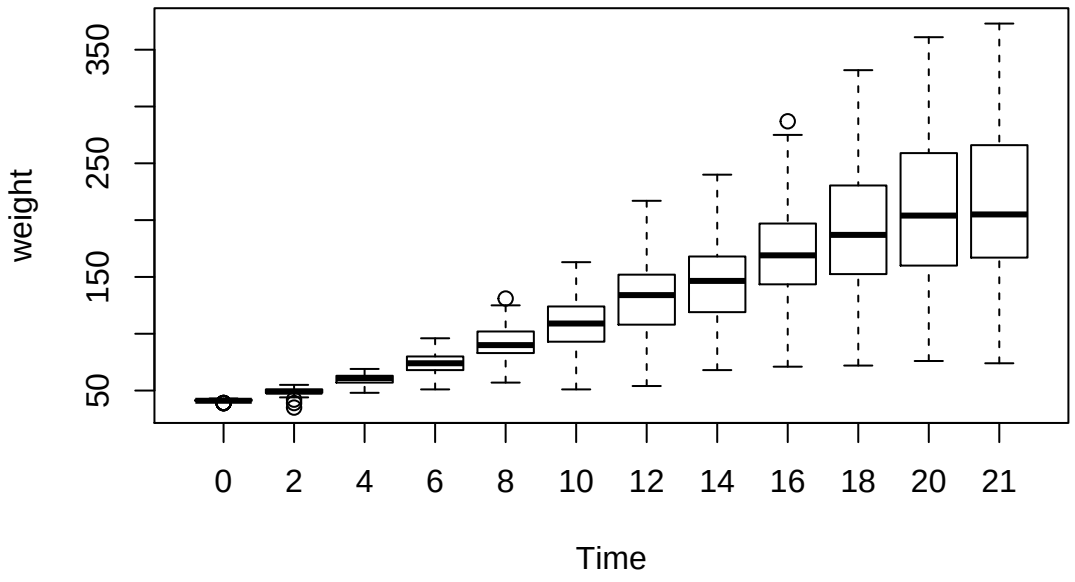
```
names(ChickWeight)
```

```
## [1] "weight" "Time"   "Chick"  "Diet"
```

The `ChickWeight` data

A graphical overview of the dataset:

```
boxplot(weight~Time,data = ChickWeight, xlab="Time", ylab="weight")
```



How to make plots like this will be explained in the course section Graphics

Subset data

First we use the function `summary()` to get a quick overview of the data:

```
summary(ChickWeight)
```

```
##      weight          Time           Chick
##  Min.   : 35   Min.   : 0.0   13     : 12
##  1st Qu.: 63   1st Qu.: 4.0   9      : 12
##  Median :103   Median :10.0   20     : 12
##  Mean   :122   Mean   :10.7   10     : 12
##  3rd Qu.:164   3rd Qu.:16.0   17     : 12
##  Max.   :373   Max.   :21.0   19     : 12
##                               (Other):506
##  Diet
##  1:220
##  2:120
##  3:120
##  4:118
##
##
##
```

Later on we will use all the data in `ChickWeight`, but for now we only want the weights of each chick on day 21:

```
CW21 <- subset(ChickWeight, Time == 21)
summary(CW21)
```

```
##      weight          Time        Chick    Diet
##  Min.   : 74   Min.   :21   13     : 1   1:16
##  1st Qu.:167   1st Qu.:21   9      : 1   2:10
##  Median :205   Median :21   20     : 1   3:10
##  Mean   :219   Mean   :21   10     : 1   4: 9
##  3rd Qu.:266   3rd Qu.:21   17     : 1
##  Max.   :373   Max.   :21   19     : 1
##                             (Other):39
```

Subset data

To make `CW21` we used conditional subsetting (`Time == 21`).

- Available comparison operators are:
  - `<` less than
  - `>` greater than
  - `==` equal to
  - `<=` less than or equal to
  - `>=` greater than or equal to
  - `!=` NOT equal to (`!` symbol indicates negation)
  - `is.na(x)` tests if `x` has missing values
- Logical operators to combine expressions are:
  - `&` logical AND
  - `|` logical OR
  - `!` logical NOT (negation)

Subset data

Often, the use of indexes will be more convenient than using the function `subset()`:

```
CW21[CW21$weight >= 250, ]
```

```
## Grouped Data: weight ~ Time | Chick
##     weight Time Chick Diet
## 84     305   21     7    1
## 167    266   21    14    1
## 232    331   21    21    2
## 280    265   21    25    2
## 292    251   21    26    2
## 328    309   21    29    2
## 352    256   21    31    3
```

```
## 364    305   21    32    3
## 388    341   21    34    3
## 400    373   21    35    3
## 436    290   21    38    3
## 448    272   21    39    3
## 460    321   21    40    3
## 484    281   21    42    4
## 554    322   21    48    4
## 578    264   21    50    4
```

Subset data

To understand how subsetting with indexes works we break it down in two steps

```
sel <- CW21$Diet == "1"
sel
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [8]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [15]  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
## [22] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [29] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [36] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [43] FALSE FALSE FALSE
```

`sel` is a logical vector that is TRUE for all the rows of CW21 that match our criteria

```
CW21.subset1 <- CW21[sel, ]
dim(CW21.subset1)
```

```
## [1] 16  4
```

Subsetting on multiple variables

Sometimes, we want to subset based on multiple variables:

```
CW21.subset2 <- CW21[(CW21$Diet == "1" & CW21$weight >= 250), ]
CW21.subset2
```

```
## Grouped Data: weight ~ Time | Chick
##     weight Time Chick Diet
## 84     305   21     7    1
## 167    266   21    14    1
```

Explore data

- What type of object are we dealing with

```r
is.data.frame(CW21)
```

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

- How much data do we have

```r
dim(CW21)
```

```
## [1] 45  4
```

- See the first 6 rows of the data

```r
head(CW21)
```

```
## Grouped Data: weight ~ Time | Chick
##    weight Time Chick Diet
## 12    205   21     1    1
## 24    215   21     2    1
## 36    202   21     3    1
## 48    157   21     4    1
## 60    223   21     5    1
## 72    157   21     6    1
```

- Get the names of the columns of our data

```r
names(CW21)
```

```
## [1] "weight" "Time"   "Chick"  "Diet"
```

- How is Diet distributed:

```r
table(CW21$Diet)
```

```
##
##  1  2  3  4
## 16 10 10  9
```

Variance and other summary statistics

The sample variance is defined as

$\sigma^2 = \frac{1}{n-1} \sum_{i=1}^{n} \left(x_i - \frac{1}{n} \sum_{i=1}^{n} x_i\right)^2$

So we can calculate

```r
x <- 1:100
sum((x - mean(x))^2)/(length(x) - 1)
```

```
## [1] 842
```

But of course a function exists to calculate the sample variance:

```r
var(x)
```

```
## [1] 842
```

The same is true for many other functions: `mean`, `sd`, `median`, `quantile`, `max`, `min`, `range`, ... We will try some of these functions in the exercises.

Missing values

- In R, a missing value is represented as `NA`
  - `NA`'s will be carried through in computations
  - **Operations on `NA` will always return `NA` as the result**
- Working with `NA` can give unexpected results:

```r
aa <- 1:10
bb <- c(1:10, NA)
mean(aa)
```

```
## [1] 5.5
```

```r
mean(bb)
```

```
## [1] NA
```

```r
mean(bb, na.rm = TRUE)
```

```
## [1] 5.5
```

Many functions have an argument `na.rm`

Distributions

R functions are available for many distributions. For most distributions, four functions are available, the density, probability, quantile, and random function. These functions are identified by the starting letter being `d`, `p`, `q`, `r`:

```r
dnorm(x, mean=0, sd=1, log = FALSE)
pnorm(q, mean=0, sd=1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean=0, sd=1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean=0, sd=1)
```

```r
plot(dnorm(seq(-4,4,by=0.1))~seq(-4,4,by=0.1), ylab = 'density')
```

The normal distribution

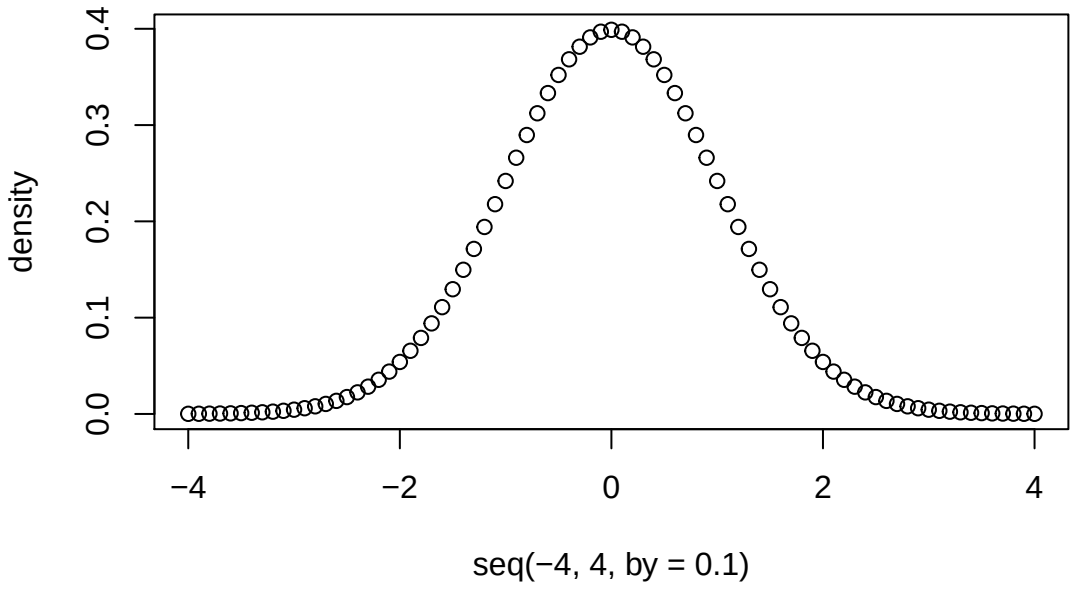


Figure 11:

```
x1 <- seq(-4, 4, 0.1)
x2 <- c(1e-04, 0.001, 0.01, 0.05, 0.1, 0.3, 0.5,0.7, 0.9, 0.95, 0.99, 0.999, 0.9999)
par(mfrow = c(1, 4), mar = c(2, 2, 2, 2))
plot(x1, dnorm(x1), main = "density")
plot(x1, pnorm(x1), main = "distribution function")
plot(x1, rnorm(x1), main = "random numbers")
plot(x2, qnorm(x2), main = "quantile function")
```

Other distributions

A large number of distributions are available:

Distribution	Function
beta	dbeta(x, shape1, shape2, ncp=0, log=F)
binomial	dbinom}(x, size, prob, log=F)
chi-squared	dchisq(x, df, ncp, log=F)
exponential	dexp(x, rate, log=F)
F	df(x, df1, df2, ncp, log=F)
gamma	dgamma(x, shape, rate, scale, log=F)
log-normal	dlnorm(x, meanlog, sdlog, log=F)
logistic	dlogis(x, location, scale, log=F)
normal	dnorm(x, mean, sd, log=F)

Distribution	Function
Poisson	dpois(x, lambda, log=F)
t	dt(x, df, ncp, log=F)
uniform	dunif(x, min, max, log=F)
weibull	dweibull(x, shape, scale, log=F)

Using distributions

Using the distribution functions as statistical tables for:

- quantile fuction to obtain significance thresholds:

```
qnorm(0.05)
```

```
## [1] -1.64
```

```
qt(0.05, df = 50)
```

```
## [1] -1.68
```

- probability function to perform testing:

```
pnorm(-1.645)
```

```
## [1] 0.05
```

```
pt(-1.676, df = 50)
```

```
## [1] 0.05
```

t-test

Hypothesis testing of problems with one variable: the t-test:

```
t.test(x, y = NULL,
       alternative = c("two.sided", "less", "greater"),
       mu = 0, paired = FALSE, var.equal = FALSE,
       conf.level = 0.95, ...)
```

t-test

Applying a t-test to the weights on days 21:

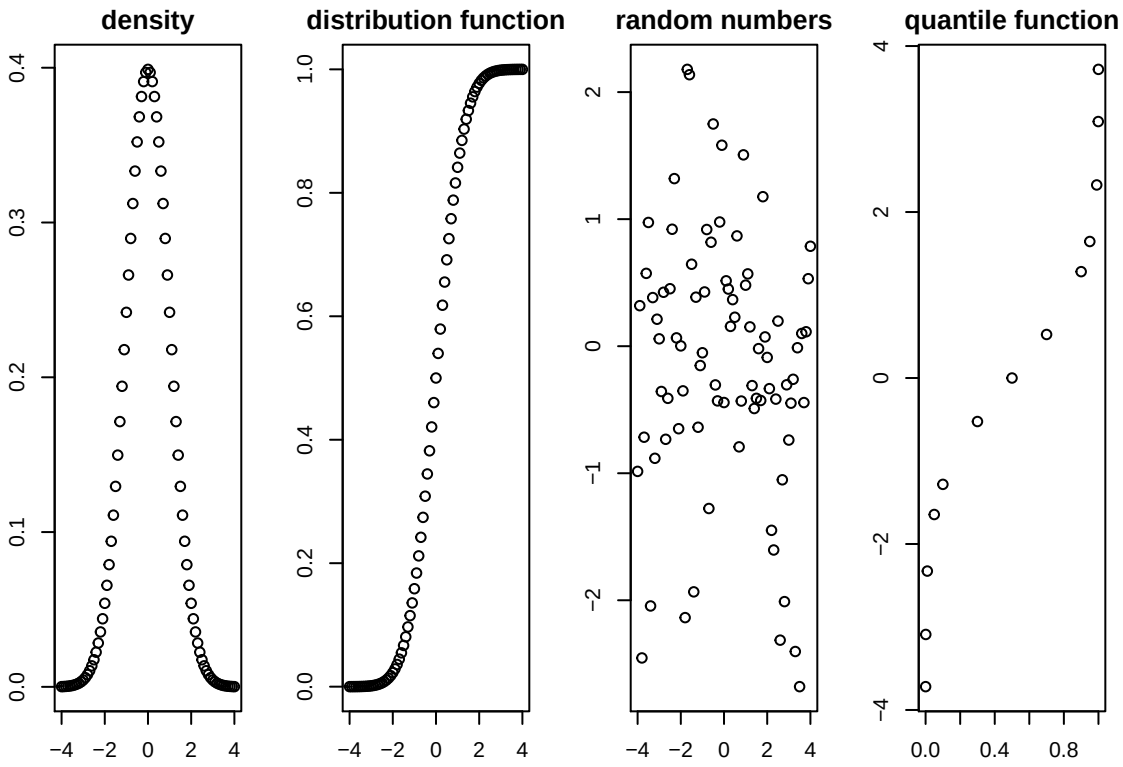


Figure 12:

```
t.test(CW21$weight)
```

```
##
##  One Sample t-test
##
## data:  CW21$weight
## t = 20, df = 40, p-value <2e-16
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  197 240
## sample estimates:
## mean of x
##       219
```

This way we test if weights are different from zero, which they obviously are.

t-test

We can also compare the mean of different groups: Are Chicks on Diet 3 heavier than Chicks on Diet 1? For this question we compare 2 samples in the t.test:

```
t.test(CW21$weight[CW21$Diet == "3"], CW21$weight[CW21$Diet =="1"], var.equal = T)
```

```
##
##  Two Sample t-test
##
## data:  CW21$weight[CW21$Diet == "3"] and CW21$weight[CW21$Diet == "1"]
## t = 4, df = 20, p-value = 0.001
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##   39.4 145.7
## sample estimates:
## mean of x mean of y
##       270       178
```

Order data

Use function `order()`

```
ord <- order(CW21$weight)
CW21.ordered <- CW21[ord, ]
head(CW21.ordered)
```

```
## Grouped Data: weight ~ Time | Chick
##     weight Time Chick Diet
## 268     74   21    24    2
## 155     96   21    13    1
## 107     98   21     9    1
## 220    117   21    20    1
## 119    124   21    10    1
## 194    142   21    17    1
```

Merging data

First we get the weights at day 10 from `ChickWeight` and copy them to another data.frame `CW10`:

```
CW10 <- ChickWeight[ChickWeight$Time == 10, c(1,3)]
names(CW10) <- c("weight10", "Chick")
head(CW10)
```

```
##    weight10 Chick
## 6        93     1
## 18      103     2
## 30       99     3
## 42       87     4
## 54      106     5
## 66      124     6
```

Merging data

Now, we want to merge the two datasets, to have two colums for each Chick with the weights at day 10 and day 21:

```
names(CW21)
```

```
## [1] "weight" "Time"   "Chick"  "Diet"
```

```
names(CW10)
```

```
## [1] "weight10" "Chick"
```

The only overlapping variable is "Chick"

```
CW <- merge(CW21, CW10, by = "Chick", all = TRUE)
head(CW)
```

```
##   Chick weight Time Diet weight10
## 1    13     96   21    1       67
## 2     9     98   21    1       96
## 3    20    117   21    1       73
## 4    10    124   21    1       81
## 5    17    142   21    1       89
## 6    19    157   21    1       71
```

Save & export data

When you have finished the analysis, the complete R workspace can be saved:

- `save.image("myR.RData")`
- `load("myR.RData")` to load in a later R session

Or you may want to export the results in a single object as a table:

```
write.table(CW, "ChickWeight21and10.txt", sep = "\t")
```

The resulting file can be read into your R session using `read.table` function, or used outside of R

Additional topics: some tips

- Use the **UP** and **DOWN** arrow keys to retrieve what you have typed in the R console (simplifies repeating and customizing your script)
- Type `history()` to see what you have entered
- Use the function `rm()` to remove objects

Return to main page

- Exercises data manipulation
  - Solutions exercises data manipulation
- Return to main page

Programming in R

Introduction

Programming: writing commands in a language understandable for a computer. An example:

```
text <- "Today is"
today <- date()
print(text)
```

```
## [1] "Today is"
```

```
print(today)
```

```
## [1] "Tue May 03 21:49:02 2016"
```

```
(Today <- paste(text,today))
```

```
## [1] "Today is Tue May 03 21:49:02 2016"
```

Make use of effort of others: *e.g.* using the `paste` function.

Important components of programming in R

- Objects
- Conditional statements
- Loops
- Indexing
- Output

Objects

Everything in R is an *object*. An object is defined by its *class*:

```
a <- 1
class(a)
```

```
## [1] "numeric"
```

```
b <- "R course"
class(b)
```

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

```
paste
```

```
## function (..., sep = " ", collapse = NULL)
## .Internal(paste(list(...), sep, collapse))
## <bytecode: 0x000000000f1aad90>
## <environment: namespace:base>
```

```
class(paste)
```

```
## [1] "function"
```

Functions

A function:

```
functionname(argument1 = a1, argument2 = a2, ..., argumentn = an)
```

- Write name of the function without parentheses: see its definition
- Run a function: provide name, parentheses and arguments. Function generally returns some result

Declare and remove objects

- The assignment operator `<-` assigns objects
- Function `rm` to remove objects;
- Or overwrite them with a new object

```
a <- 1
a <- 2
print(a)
```

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

```
rm(a)
exists('a')
```

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

```
a <- "object a"
b <- a
b == a
```

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

Note: the '==' operator. Returns `TRUE` or `FALSE`

Conditional operators

Suppose that we want to *know* if a certain condition is true. We use *conditional operators*:

```
a <- 1; b <- 2
a < b
```

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

```
a <= b
```

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

```
a > b
```

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

```
a >= b
```

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

```
a == b
```

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

```
a != b
```

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

Conditional operations

Suppose that we want to *do* something depending on a certain outcome:

```
{
    if(a < b)
        print(" a is smaller than b")
    else if (a > b)
        print(" a is bigger than b")
    else
        print("a and b are equal")
}
```

```
## [1] " a is smaller than b"
```

Logical **OR** and **AND** operators:

```
{
  if(a == 1 & b == 10)
    print("A is equal to 1 AND B is equal to 10")
  else if(a == 1 | b == 10)
    print("A is equal to 1 OR B is equal to 10")
  else
    print("A is not equal to 1 NOR B equal to 10")
}
```

```
## [1] "A is equal to 1 OR B is equal to 10"
```

Vectors

- `numeric` vector is a series of numbers
- `character` vector is a series of characters
- `factor` vector a series of factors

Advantage of `factor` over `character` is related to storage.

Basic functions to make vectors

Function `c` to *concatenate*, required to join numbers or characters into a vector:

```
(v1 <- c(1,2))
```

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

```
(v2 <- c("a","b","cc"))
```

```
## [1] "a"  "b"  "cc"
```

```
(v3 <- c(v1,v1))
```

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

```
(v4 <- c(v1,v2))
```

```
## [1] "1"  "2"  "a"  "b"  "cc"
```

```
(v5 <- c(first = 1,second = 2,last = 1000))
```

```
##  first second   last
##      1      2   1000
```

Function `seq` to *make a sequence*:

```r
(seq1 <- seq(from = 0,to = 10,by = 1))
```

```
##  [1]  0  1  2  3  4  5  6  7  8  9 10
```

```r
(seq2 <- seq(from = 1,to = 2,length.out = 3))
```

```
## [1] 1.0 1.5 2.0
```

Indexing a vector

The '[' operator to access specific elements of a vector:

```r
v1 <- seq(0,10)
v1[1]
```

```
## [1] 0
```

```r
v1[10]
```

```
## [1] 9
```

```r
ii <- seq(1,3)
v1[ii]
```

```
## [1] 0 1 2
```

Note: the last option; access various indices 1,2,3 directly. This is called *vectorized computing*.

Indexing by name

Give name to elements of a vector and use the names to access elements:

```r
v1
```

```
##  [1]  0  1  2  3  4  5  6  7  8  9 10
```

```r
names(v1) <- letters[1:length(v1)]
v1
```

```
##  a  b  c  d  e  f  g  h  i  j  k
##  0  1  2  3  4  5  6  7  8  9 10
```

```r
v1["b"]
```

```
## b
## 1
```

```r
v5["first"]
```

```
## first
##     1
```

Note: the `':'` operator is similar to the `seq` function. For for help: `?':'`.

```r
1:3
```

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

Character vectors

Character vectors are vectors of character strings:

```r
letters
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k"
## [12] "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v"
## [23] "w" "x" "y" "z"
```

```r
class(letters)
```

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

```r
(action <- c("programming","in","R"))
```

```
## [1] "programming" "in"          "R"
```

```r
length(action)
```

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

```r
action[1]
```

```
## [1] "programming"
```

```r
action[1:3]
```

```
## [1] "programming" "in"          "R"
```

Function `paste`

Function `paste` to combine characters into one character string:

```
(p1 <- paste(1,letters[1:10],sep = "_"))
```

```
##  [1] "1_a" "1_b" "1_c" "1_d" "1_e" "1_f" "1_g"
##  [8] "1_h" "1_i" "1_j"
```

```
(p2 <- paste(1:10,letters[1:10],sep = "."))
```

```
##  [1] "1.a"  "2.b"  "3.c"  "4.d"  "5.e"  "6.f"
##  [7] "7.g"  "8.h"  "9.i"  "10.j"
```

```
(p3 <- paste(1:3,letters[1:10],sep = "."))
```

```
##  [1] "1.a" "2.b" "3.c" "1.d" "2.e" "3.f" "1.g"
##  [8] "2.h" "3.i" "1.j"
```

Note p3: first argument is *recycled*.

To concatenate result into one string:

```
(p4 <- paste(p3,collapse = "/"))
```

```
## [1] "1.a/2.b/3.c/1.d/2.e/3.f/1.g/2.h/3.i/1.j"
```

Function `strsplit`

Suppose that we have a string and want to split it at certain positions:

```
(p5 <- strsplit(p4,split = "/"))
```

```
## [[1]]
##  [1] "1.a" "2.b" "3.c" "1.d" "2.e" "3.f" "1.g"
##  [8] "2.h" "3.i" "1.j"
```

```
p6 <- strsplit(unlist(p5),split = "\\.")
p6[1:2]
```

```
## [[1]]
## [1] "1" "a"
##
## [[2]]
## [1] "2" "b"
```

```
(todaysplitted <- unlist(strsplit(today,split = " ")))
```

```
## [1] "Tue"      "May"      "03"       "21:49:02"
## [5] "2016"
```

Note: `strsplit` returns a `list`. Use `unlist` to transform the list to a `character` object.

Functions `substr` and `gsub`

Suppose that we want a section of the string:

```
string <- "this is a string"
(sstring <- substr(x = string,start = 1,stop = 4))
```

```
## [1] "this"
```

Suppose that we want to substitute a part of a string:

```
(nstring <- gsub(pattern = "is",replacement = "was",
                 x = string))
```

```
## [1] "thwas was a string"
```

```
(nnstring <- gsub(pattern = "[[:space:]]is[[:space:]]"," was ",string))
```

```
## [1] "this was a string"
```

This also introduces the regular expressions. Have a look at `?regexp` or google for this topic.

Operator `'%in%'`

Suppose that we want to know if elements of one vector are present in another vector: operator `'%in%'`:

```
v1 <- letters[1:10]
v2 <- letters[5:15]
v1%in%v2
```

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

It does not tells us *where* each element of `v1` appears in `v2`.

Function `match`

Suppose that we want to know `where` each element of `v1` appears in `v2`: function `match`:

```
v1
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

```
v2
```

```
##  [1] "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o"
```

```r
match(v1,v2)
```

```
##  [1] NA NA NA NA  1  2  3  4  5  6
```

```r
match(v1,c(v2,v2))
```

```
##  [1] NA NA NA NA  1  2  3  4  5  6
```

Note: `match` only returns the first match.

Function `which`

Function `which` to obtain the index where a condition is `TRUE`:

```r
which(v1 == "a")
```

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

```r
vv <- c(v1,v1,v1)
which(vv == "a")
```

```
## [1]  1 11 21
```

Now, we use `which` to substitute the elements of `vv` which are a with A:

```r
vv[which(vv=="a")] <- "A"
```

Extending the `match` function:

Suppose that we want **all** the matches. We need to program:

```r
v22 <- c(v2,v2)
indices <- lapply(v1,function(i)
   which(v22%in%i))
```

A more elegant way: write a new function called `matchall`, first check if it is not already defined (`help.search('matchall')`:

```r
matchall <- function(x,y){
  lapply(x,function(i){
    which(y%in%i)})
}
identical(indices,matchall(v1,v22))
```

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

This introduces how to program functions.

Time for exercises

- Go to exercises programming 1
- Go to solutions programming 1
- Return to main page

Matrices

Function `matrix` to create a matrix:

```
(mat1 <- matrix(1:4,nrow = 2,ncol = 2,byrow = T))
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
```

Access the elements of the matrix with `'['` operator:

```
mat1[1,1]
```

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

```
mat1[1,]
```

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

Matrices have two dimensions: the `','` separates the dimensions.

Replace elements of the matrix with the `[ <-` operator:

```
mat1[,2] <- -10
mat1
```

```
##      [,1] [,2]
## [1,]    1  -10
## [2,]    3  -10
```

Combining matrices

Function `rbind` to combine two matrices under each other (make sure that the number of columns coincides):

```
mat2 <- matrix(0,nrow = 2,ncol = ncol(mat1))
mat3 <- rbind(mat1,mat2)
dim(mat3)
```

```
## [1] 4 2
```

Function `cbind` to combine matrices by column (make sure that the number of rows coincides):

```r
mat4 <- cbind(mat1,mat2)
dim(mat4)
```

```
## [1] 2 4
```

Other functions for matrices

- Transpose of a matrix: function `t`
- The inverse of a (square, invertable) matrix: function solve
- The matrix product: `'%*%'` operator
- Obtain diagonal elements of (square) matrix: function `diag`
- Replace diagonal elements of a (square) matrix: function `diag <-`

Application of matrix functions

Suppose that we have observations $\mathbf{x}$ and dependent observations $\mathbf{y}$ and we want to know the linear relationship between $\mathbf{x}$ and $\mathbf{y}$:

$\mathbf{y} = \mathbf{x}b$,

where b is an *unknown* regression coefficient. The *best*, least squares, solution is:

$\hat{b} = (\mathbf{x}'\mathbf{x})^{-1}\mathbf{x}'\mathbf{y}$.

Now: program it in R:

```r
(x <- rep(1:3,length.out = 10))
```

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

```r
b <- 0.5
(y <- b*x + rnorm(10))
```

```
##  [1] -0.871 -0.385  1.255  0.585  2.301  2.356
##  [7]  0.767  1.181  1.720 -0.814
```

```r
(bhat <- solve(t(x)%*%x)%*%t(x)%*%y)
```

```
##       [,1]
## [1,] 0.508
```

Compare the original b to the estimate of b, *bhat*.

Note: more efficient way to solve linear models with function `lm`; see slides on regression.

Objects of class `data.frame`

The columns of a `matrix` are of one single class. A `data.frame` is *generalized* `matrix` where each column can be of a different class.

Create an object of class `data.frame`:

```
animals <- data.frame(id = letters[1:10],
                      weight = rnorm(mean = 100,10),
                      sex = sample(c("F","M"),10,replace = T))
head(animals)
```

```
##   id weight sex
## 1  a   99.7   M
## 2  b  100.5   F
## 3  c  100.4   F
## 4  d   99.0   F
## 5  e   99.2   M
## 6  f   99.1   M
```

Manipulating data.frames

We want to extract the weights from `animals` into a vector `weigths`:

```
weights <- animals$weight
```

We want the subset of the the `animals` where `weight` is more than 100 in a new data.frame `animals100p`:

```
animals100p <- animals[animals$weight>100,]
```

We we want to substitute `weight` when `sex == 'M'` by `NA`:

```
animals$weight[animals$sex == 'M'] <- NA
```

We want to add a column to `animals` with the squared `weight`:

```
animals$weight2 <- animals$weight^2
```

Reading and writing data

Function `write.csv` write a `data.frame` as a `*.csv` file:

```
write.csv(animals,file = 'animals.csv',row.names= FALSE,quote = FALSE)
```

Function `read.csv` to read a csv file into a `data.frame`:

```
animals1 <- read.csv(file = 'animals.csv',header = TRUE)
```

Note: we made a file called `animals.csv` somewhere on the HD and then read this file back in the memory. Test if a file exists:

```
if(!file.exists('nonexistentfile.csv'))
    print("File does not exists.")
```

```
## [1] "File does not exists."
```

Time for exercises

- Go to exercises programming 2
- Go to solutions programming 2
- Return to main page

Repeating

Suppose that we want to repeat an operation many times:

Sum up random numbers until the sum reaches 10:

```
s <- 0
n <- 0
while(s<10){
  n <- n + 1
  s <- s + runif(1,0,1)
}
print(s)
```

```
## [1] 10.1
```

```
print(n)
```

```
## [1] 20
```

We needed 20 steps to get a sum higher than 10.

Loops: `for`

Suppose that we want to *follow* a certain sequence of values and do something at each value, **looping**:

```
vec1 <- rnorm(10)
vec2 <- numeric(10)
for(i in 1:length(vec1))
{
    vec2[i] <- abs(vec1[i])
}
```

Note: try to avoid loops. Make code slow and ugly. Much better and faster solution:

```
vec2 <- abs(vec1)
```

Loops: `while`

Suppose that we want to do something until a certain criterion is met:

```
i <- 0
while(i<10)
    i <- i + 1
```

Be careful: do not make endless loops:

```
i <- 1
while(i>0)
    i <- i + 1
```

Function `apply`

Suppose that we have a matrix and want to calculate the sums of each column (columns are the 2^{nd} dimension of a matrix).

Ugly method:

```
mat <- matrix(rnorm(20),nrow = 2,ncol = 10)
uglyCS <- numeric(ncol(mat))
for(col in 1:ncol(mat)){
    uglyCS[col] <- sum(mat[,col])
}
```

Elegant method:

```
elegantCS <- apply(mat,2,sum)
identical(uglyCS,elegantCS)
```

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

Function `apply` *applies* a function to a matrix. Here the function `sum` is applied to each column of `mat` (2^{nd} dimension), second argument of `apply` is 2. Try to calculate the sums of the rows of `mat`.

Intermezzo: the `ChickWeight` dataset

The `ChickWeight` dataset is from an experiment of the effect of diet on the weight of chicken. The variables are:

- `weight`: the weight of the chicken expressed in grams;
- `Time`: time since hatching when the measurement was done;

- `Chick`: identifies each chick;
- `Diet`: identifies the diet of each chick.

```
summary(ChickWeight)
```

```
##      weight          Time          Chick
##  Min.   : 35   Min.   : 0.0   13     : 12
##  1st Qu.: 63   1st Qu.: 4.0   9      : 12
##  Median :103   Median :10.0   20     : 12
##  Mean   :122   Mean   :10.7   10     : 12
##  3rd Qu.:164   3rd Qu.:16.0   17     : 12
##  Max.   :373   Max.   :21.0   19     : 12
##                               (Other):506
##  Diet
##  1:220
##  2:120
##  3:120
##  4:118
##
##
##
```

Function `tapply`

Suppose that we want to obtain mean weight per Diet and per Time period from `ChickWeight`. We need:

- the number of unique timepoints;
- the number of unique diets;
- the sum of `weight` for each `DietxTime` combination.

Ugly method:

```
mns <- list()
for(t in unique(ChickWeight$Time)){
    for(d in unique(ChickWeight$Diet)){
        ii <- with(ChickWeight,which(Diet%in%d&Time%in%t))
        mns[[paste(t,d)]] <-
            mean(ChickWeight$weight[ii])
    }}
```

Elegant method:

```
tt <- tapply(ChickWeight$weight,list(ChickWeight$Diet,
                                     ChickWeight$Time),mean)
tt[,1:4]
```

```
##      0    2    4    6
## 1 41.4 47.2 56.5 66.8
## 2 40.7 49.4 59.8 75.4
## 3 40.8 50.4 62.2 77.9
## 4 41.0 51.8 64.5 83.9
```

Here we applied the function `mean` to `Diet` at each unique `time`. Again, with less writing using function `with`:

```
tt <- with(ChickWeight,tapply(weight,list(Diet,Time),mean))
tt[,1:4]
```

```
##      0    2    4    6
## 1 41.4 47.2 56.5 66.8
## 2 40.7 49.4 59.8 75.4
## 3 40.8 50.4 62.2 77.9
## 4 41.0 51.8 64.5 83.9
```

Programming functions

Tools in R are *functions*. Defining a function is not difficult.

Example: a function to summarize data into a mean and a standard deviation, `meansd`:

```
meansd <- function (x, nDec = 2,na.rm = TRUE)
{
  mn <- mean(x, na.rm = na.rm)
  if (!is.na(mn))
    mn <- round(mn, nDec)
  sd <- sd(x, na.rm = na.rm)
  if (!is.na(sd))
    sd <- round(sd, nDec)
  return(paste(mn, " (", sd, ")", sep = ""))
}
meansd(1:10)
```

```
## [1] "5.5 (3.03)"
```

Application of `meansd`

Suppose that we want to summarize `weight` for each combination of `diet` and `time` into mean and standard deviation:

```
ttFancy <- tapply(ChickWeight$weight,list(ChickWeight$Diet,ChickWeight$Time),
                  meansd,1)
                  ttFancy[,10:12]
```

```
##   18             20             21
## 1 "158.9 (49.2)" "170.4 (55.4)" "177.8 (58.7)"
## 2 "187.7 (63.3)" "205.6 (70.3)" "214.7 (78.1)"
## 3 "233.1 (57.6)" "258.9 (65.2)" "270.3 (71.6)"
## 4 "202.9 (33.6)" "233.9 (37.6)" "238.6 (43.3)"
```

Time for exercises

- Go to exercises programming 3
- Go to solutions programming 3
- Return to main page

Graphics in R

Function `plot`

Function `plot` is the basic function for plotting:

```
plot(weight~Time,data = ChickWeight)
```

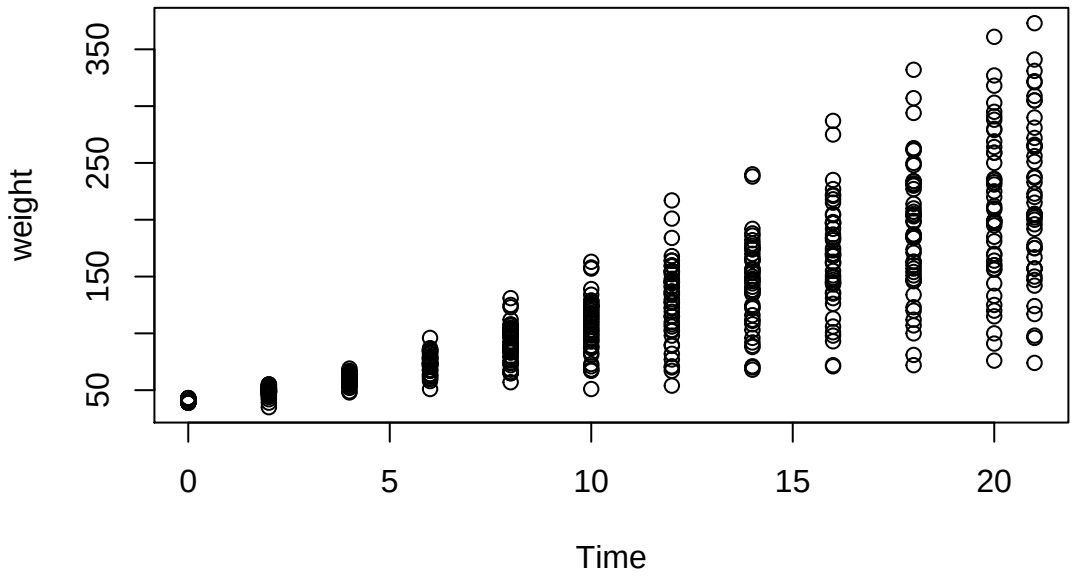


Figure 13: Plot of weight against time.

Specify the display with a formula. LHS of formula is the vertical axis, RHS is the horizontal axis of the plot (usually):

```
class(weight~Time)
```

```
## [1] "formula"
```

Function `boxplot`

Function `boxplot` for making box-and-wisker plots:

```
boxplot(weight~Time,data = ChickWeight)
```

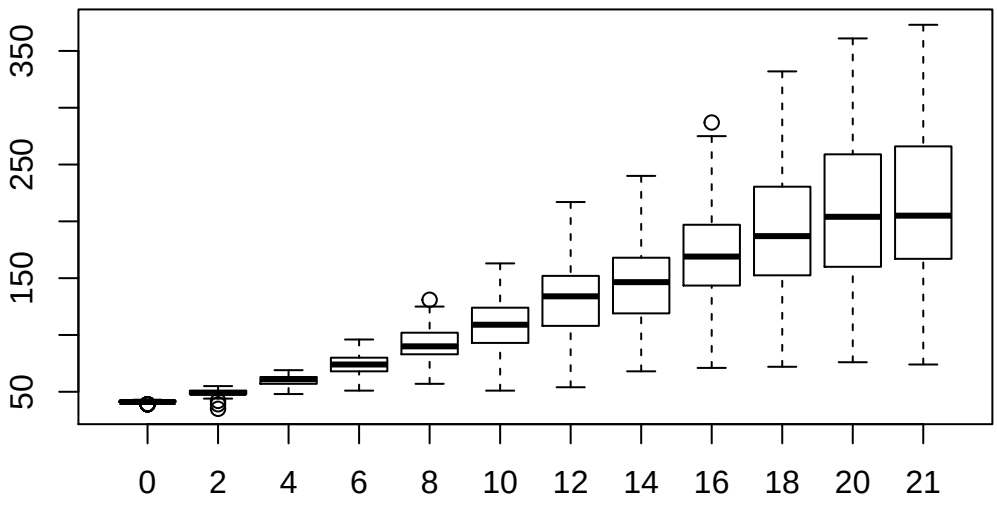


Figure 14: Boxplot of chickweights.

Function `barplot`

For a `barplot` we need to provide a matrix or a vector of data:

```
mat1 <- with(ChickWeight,tapply(weight,Diet,mean))
barplot(mat1,xlab = "Diet",ylab = " Mean weight (kg)")
```

Combining graphics

Suppose that we want to combine the three graphs into one plot. We use function `par` to set specific graphical parameter settings, in our case `mfrow`:

```
par(mfrow = c(1,3))
plot(weight~Time,data = ChickWeight)
boxplot(weight~Time,data = ChickWeight)
barplot(mat1,xlab = "Diet",ylab = " Mean weight (kg)")
```

Saving graphics

Functions `jpeg`, `png`, `bmp` and `pdf` to save graphics in specific formats:

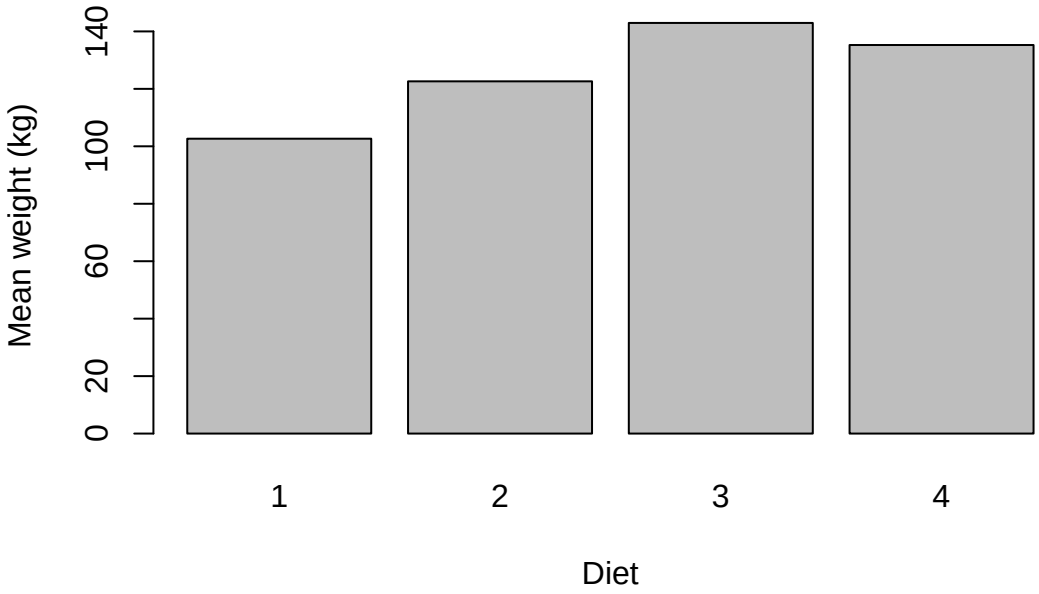


Figure 15: Barplot of chickweights.

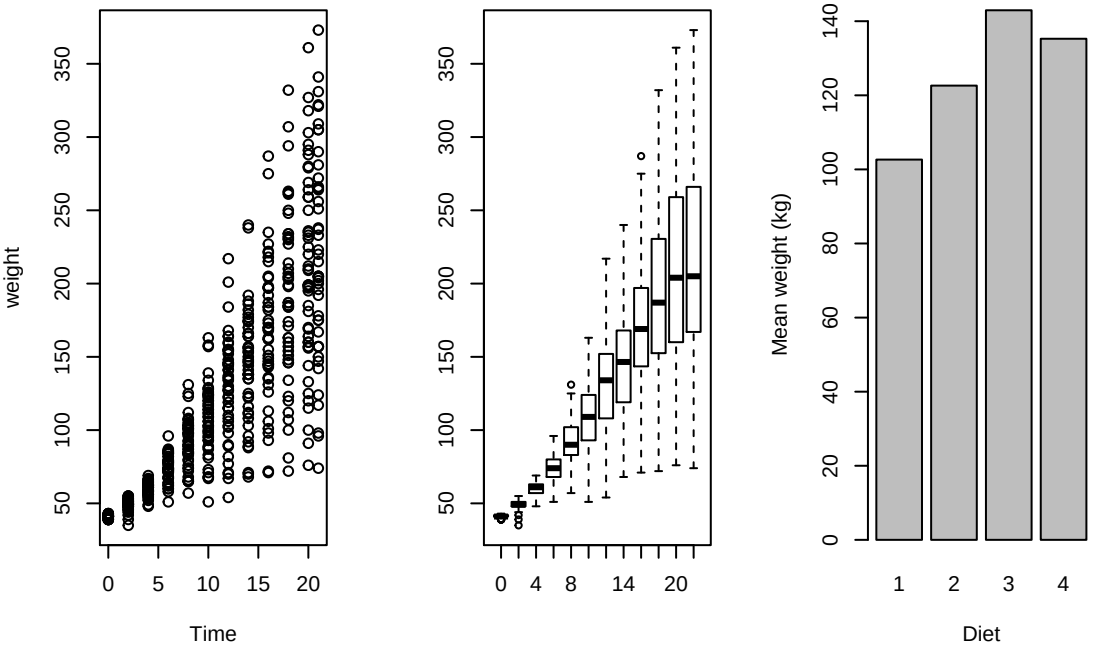


Figure 16: Plots combined with function par().

```
pdf(file = "figure/Cboxplot.pdf",width = 20,height = 10)
boxplot(weight~Time,data = ChickWeight,col = "blue")
graphics.off()
file.exists('figure/Cboxplot.pdf')
```

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

Note the call `graphics.off()` to close the *graphical printer*.

Customizing a graphic:

Suppose that we want:

- A scatterplot with the data (`ChickWeight` data);
- Add a regression line for each `Diet`;
- Add a small legend in the top-left part of the graph.

```
## convert Diet to numeric for convenience
ChickWeight$Diet <- as.numeric(ChickWeight$Diet)
nDiets <- max(ChickWeight$Diet)
colors <- rainbow(nDiets)
lineTypes <- (1:nDiets)
plotChar <- 18+(1:nDiets)
plot(weight~Time,type = "n",data = ChickWeight)

for(d in 1:nDiets)
{
  CW <- ChickWeight[ChickWeight$Diet%in%d,]
  points(weight~Time,data=CW,col = colors[d],pch=plotChar[d])
  abline(lm(weight~Time,data = CW),col=colors[d],lty=lineTypes[d],lwd=1.5)
}

legend(x = 0.05*max(ChickWeight$Time),
       y = 0.85*max(ChickWeight$weight),
       col = colors,pch = plotChar,lty = lineTypes,
       legend = paste("Diet",unique(ChickWeight$Diet)),
       text.col = colors)
```

- We need to take care of the correspondence between the plotting characters, colors and line types of the plot and the legend.
- We need to specify everything, usually with a loop.
- Very difficult to do for general situations.
- Even more difficults in a *matrix* of plots, us shown previously with the `par(mfrow=c(1,3))` specification.

There are special packages designed to plot multivariate data, see lattice and ggplot2. In this course, we will only use package ggplot2.

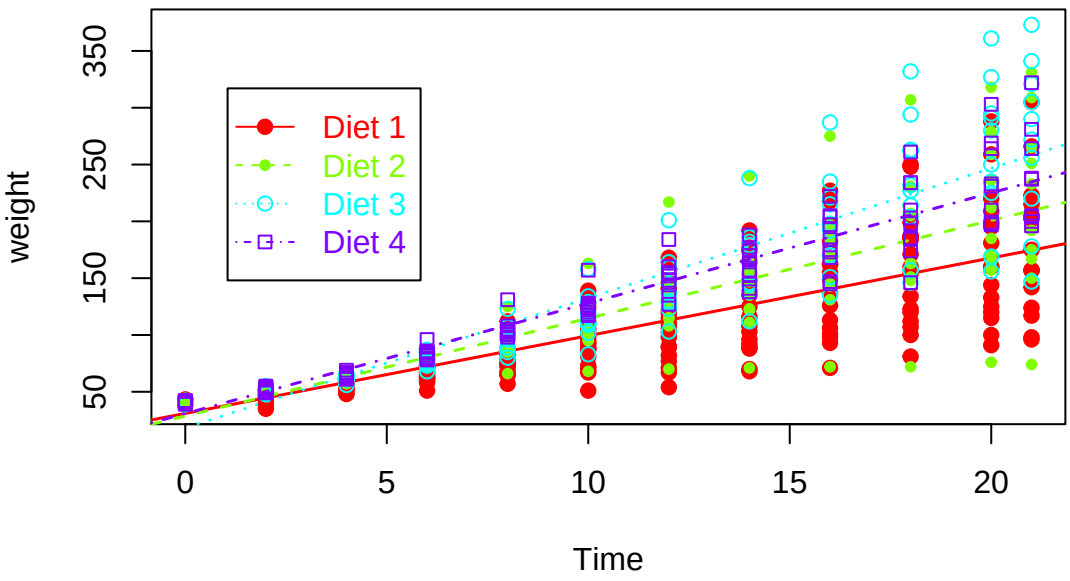


Figure 17: Plot with extras.

Time for exercises

- Go to exercises basic graphics
- Go to solutions basic graphics
- Return to main page

The ggplot2 package

The ggplot2 package is based on the *grammar of graphics*. It is important to mention because it has become very popular among R-users and because it is very useful to create a wide variety of graphics in R.

Users who wish to create a graphic using the ggplot2 package need to think about the following aspects:

- The *data* and the *aestetic mapping* of the data, which describes how the variables are displayed.
- Geometic objects, *geoms*, represent what you see on the plot (points, lines, polygons).
- Statistical transformations of the data, *stats*, summarize and transform the data before displaying them in the plot.
- The *scales* map the (transformed) values in the data to an aestetic space. The scales also provide the legend.
- A coordinate system, *coord*.
- A *facet*ting specification describes how to break up the data into subsets.

Creating a plot

We use function `ggplot` to create a plot object. Function `ggplot` has two arguments: *data* and aestetic mapping, which set up the defaults of the plot object and can also be specified in each layer. The aestetic mapping of the data can be specified with function *aes*.

```
p <- ggplot(data = ChickWeight,aes(x = Time,y = weight))
```

This plot can not be displayed until we add a layer, since there is nothing to see:

Adding layers to the plot

To display something, we need to add a layer to the plot. The layers can be added with function `layer`:

```
layer(geom, geom_params, stat, stat_paramss, data, mapping, position)
```

Or we can use specific functions to add specific layers:

- `geom_bar` to add bars to the plot
- `geom_errorbar` to add errorbars
- `geom_point` to add points to the plot
- `geom_line` to add lines
- `geom_histogram` to add a histogram

We can also perform some analyses before and subsequently add a layer to the plot:

- `stat_summary` to summarize the data at each x-value
- `stat_smooth` to draw a smoothed line through the data

Adding points

Now, we add points representing the individual weights to the plot:

```
p + geom_point()
```

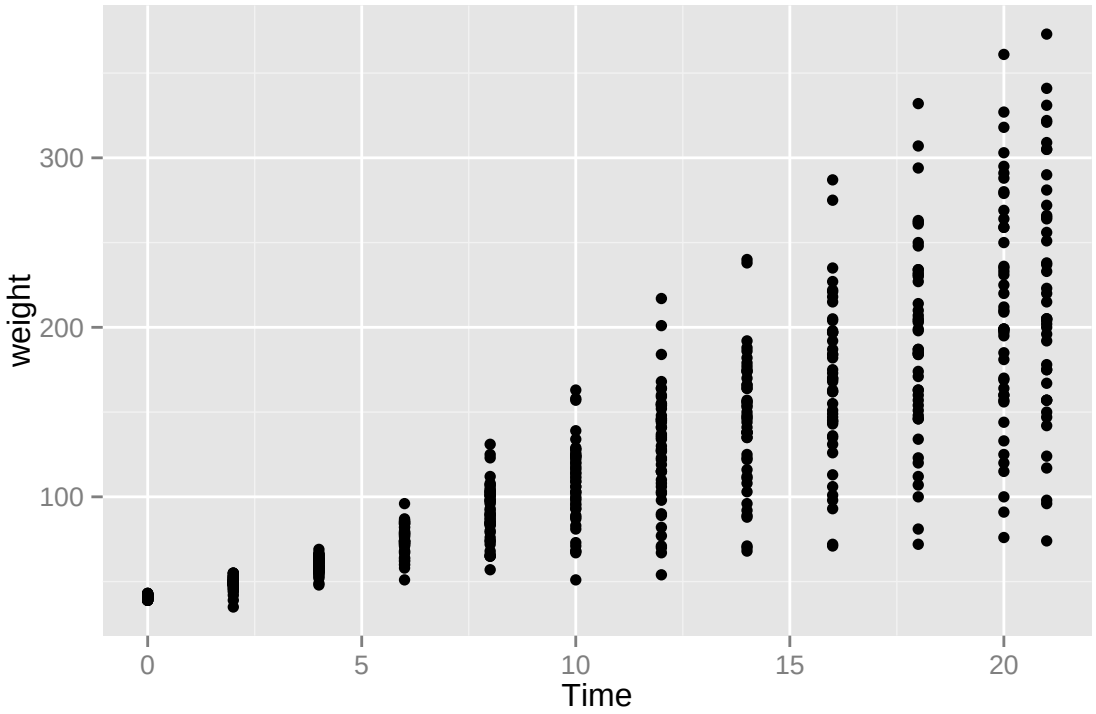


Figure 18: Scatterplot with ggplot.

the + operation returns a ggplot object to the console which is not stored into a new object, we can store the resulting object as a new object, but now we need to specifically call this object to print it to the screen:

```
p1 <- p + geom_point()
p1
```

Differentiating between chicks:

We can differentiate between the chick using distinct colours for each chick:

```
p + geom_point(aes(colour = Chick))
```

since the number of chicks is large, the legend is useless. We use function `theme` to remove the legend:

```
p + geom_point(aes(colour = Chick)) + theme(legend.position = 'none')
```

We can also use a distinct shape for each chick:

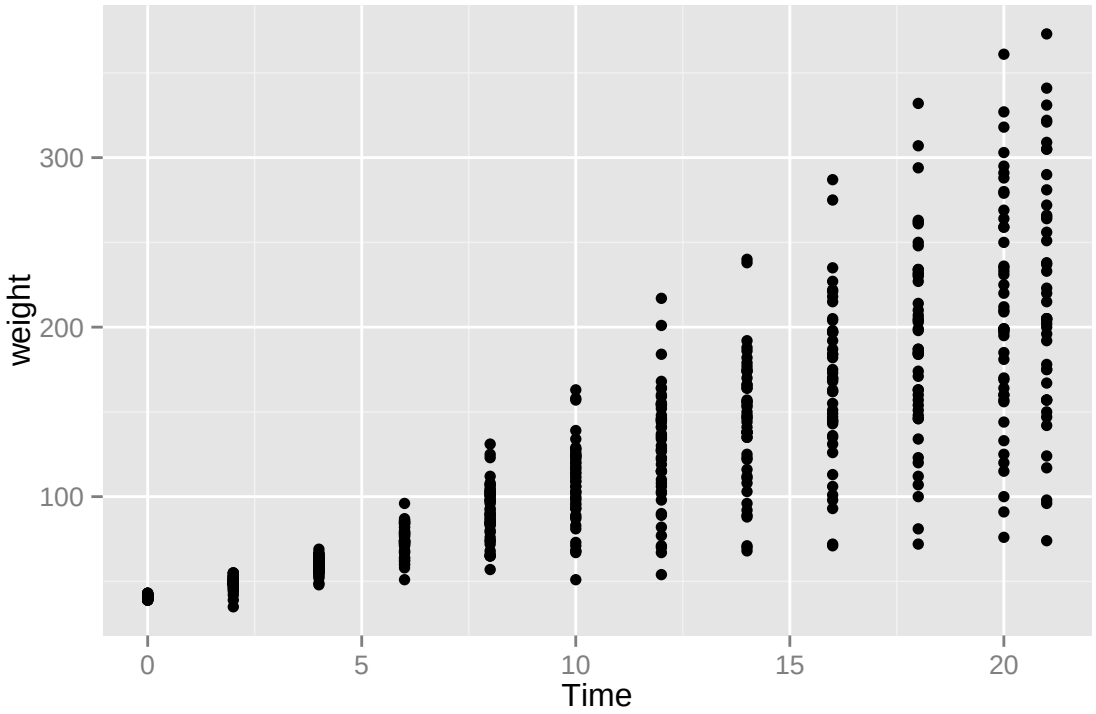


Figure 19: Another scatterplot with ggplot.

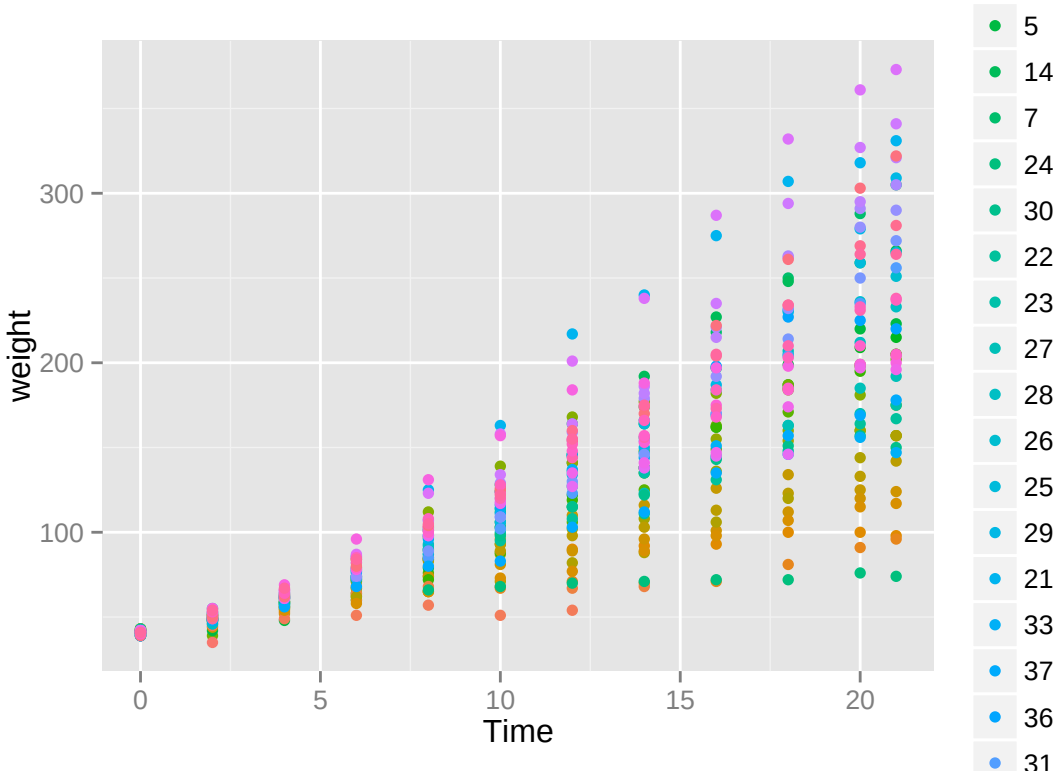


Figure 20: Scatterplot with distinct colour for each chick.

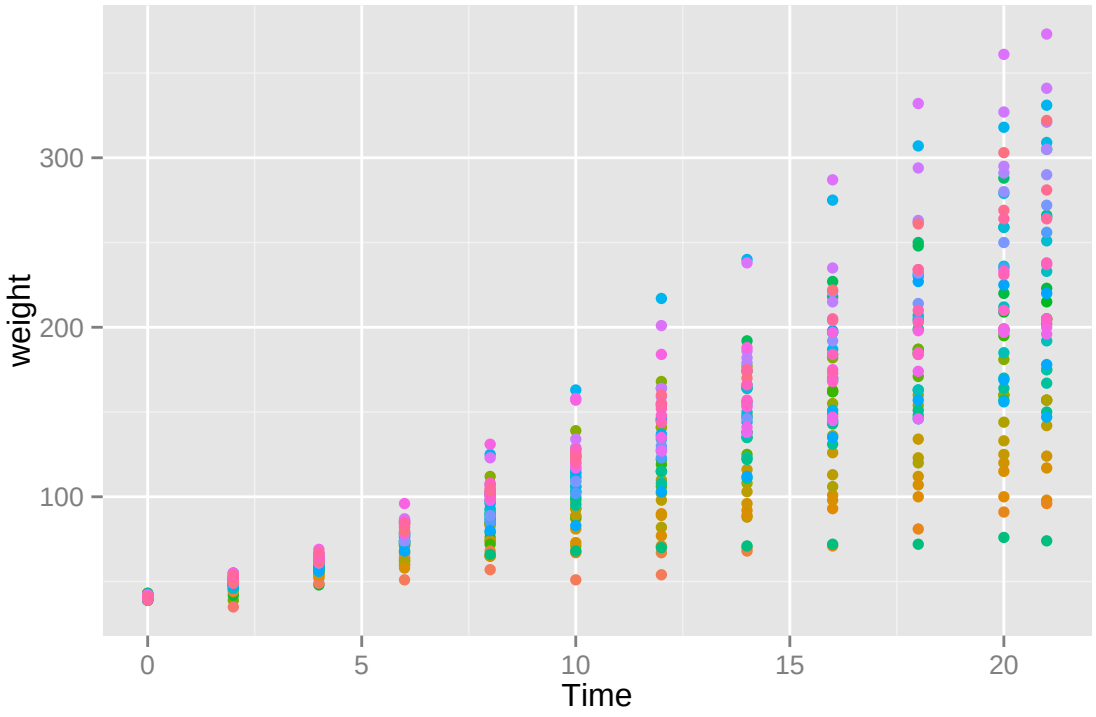


Figure 21: Scatterplot with distinct colour for each chick without legend.

```
p + geom_point(aes(shape = Chick,colour = Chick)) + theme(legend.position = 'none')
```

```
## Warning: The shape palette can deal with a maximum of
## 6 discrete values because more than 6
## becomes difficult to discriminate; you have
## 50. Consider specifying shapes manually if
## you must have them.

## Warning: Removed 525 rows containing missing
## values (geom_point).
```

Since we have more than 6 Chicks, we should specify the shapes manually. First, let's have a look at the different point shapes available in `ggplot2`:

```
df <- data.frame(x=  1:12,y = 1,sh = as.character(1:6),colour = as.character(1:6))
ggplot(data = df,aes(x= x,y = y,colour = colour,shape = colour)) + geom_point()
```

```
ChickWeight$shape <- factor(as.numeric(ChickWeight$Chick)%%6)
p$data <- ChickWeight
p + geom_point(aes(shape = shape,colour = Chick)) + theme(legend.position = 'none')
```

I used the modulus operator '%%' of R (the modulo operation finds the remainder of the division of one number by anonther, see wikipedia).

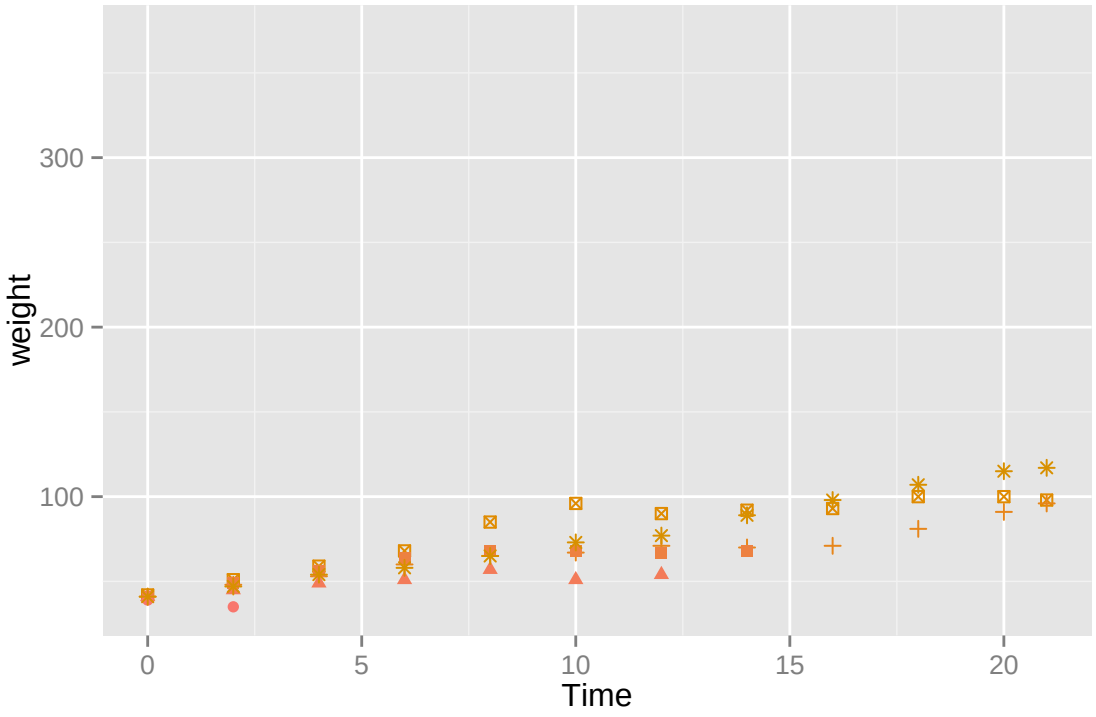


Figure 22: Scatterplot with distinct colour and shape for each chick

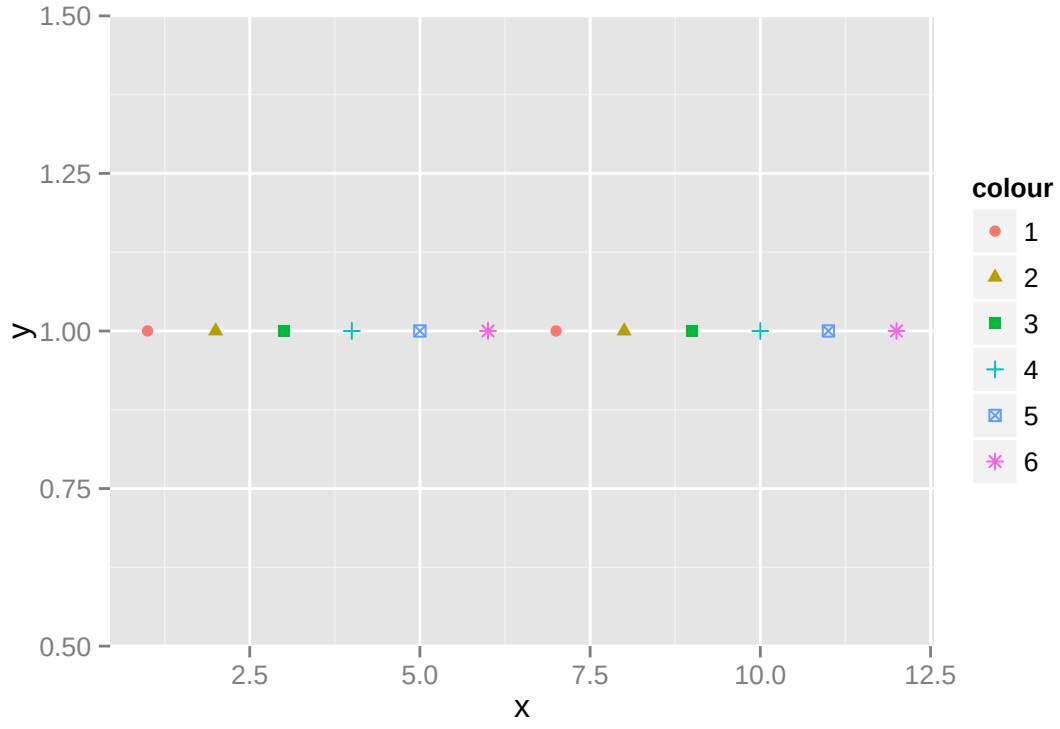


Figure 23: Scatterplot with 6 distinct colours and shapes for chicks

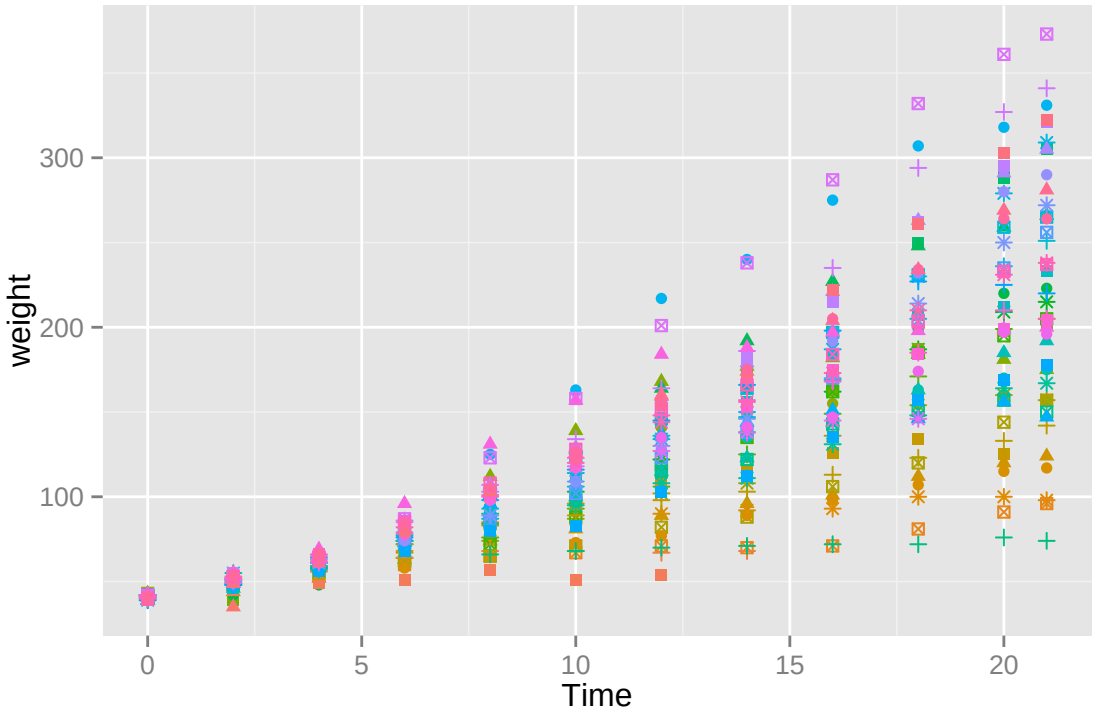


Figure 24: Scatterplot with 6 distinct colours and shapes for chicks

Connecting the chicks by a line

Since there a many observations on each chick, we can connect the observations by individual lines, for this, we need to add another layer using function `geom_lines`. Simply adding `+ geom_line()` does not work correctly, since `ggplot2` does not know which observations should be connected to each other:

```
p + geom_line()
```

Hence, we need to tell function `ggplot` that each chick is a group. Furthermore, we can specify the type of the lines and the colour of the lines as we did previously for the points:

```
p <- ggplot(data = ChickWeight,aes(x = Time,y = weight,
     colour = shape,group = Chick,linetype = shape))
p + geom_line() + theme(legend.position = 'none')
```

Summarizing data

We might want to summarize the observations in a graphic

```
ChickWeight$Diet <- factor(ChickWeight$Diet)
p <- ggplot(data = ChickWeight,aes(x = Time,y = weight))
p + stat_summary(fun.y = 'mean',geom = 'line',aes(group = Diet,colour = Diet)) +
    stat_summary(fun.y = 'sd',geom ='point',aes (group = Diet,colour = Diet))
```

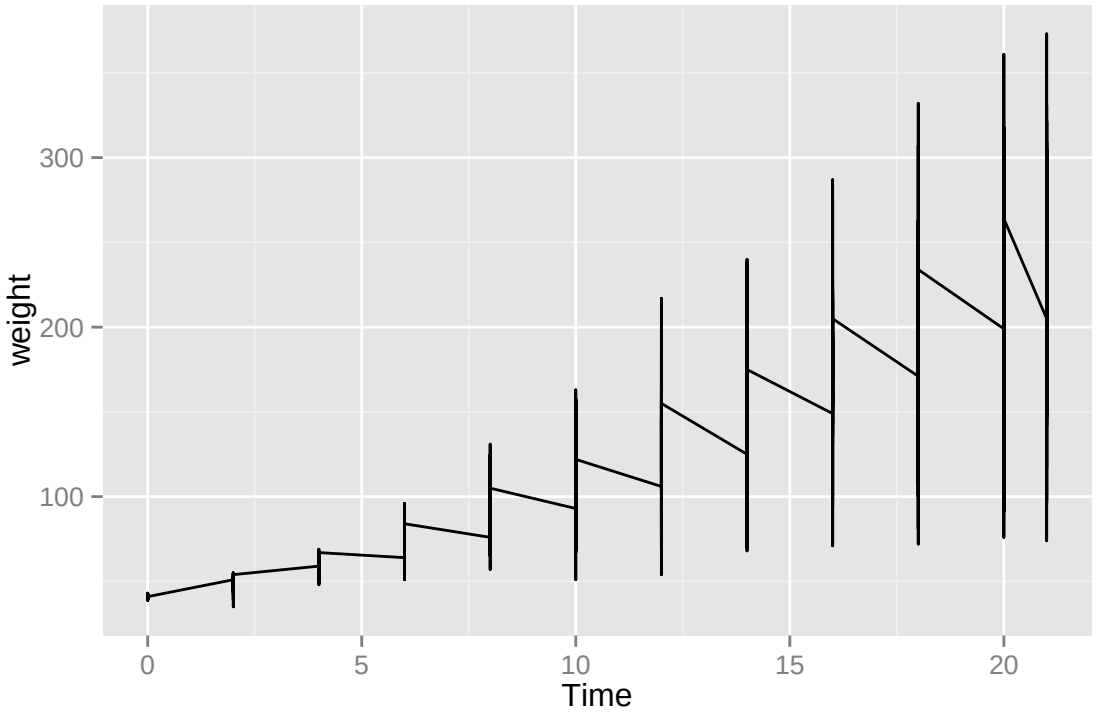


Figure 25: Plot with lines.

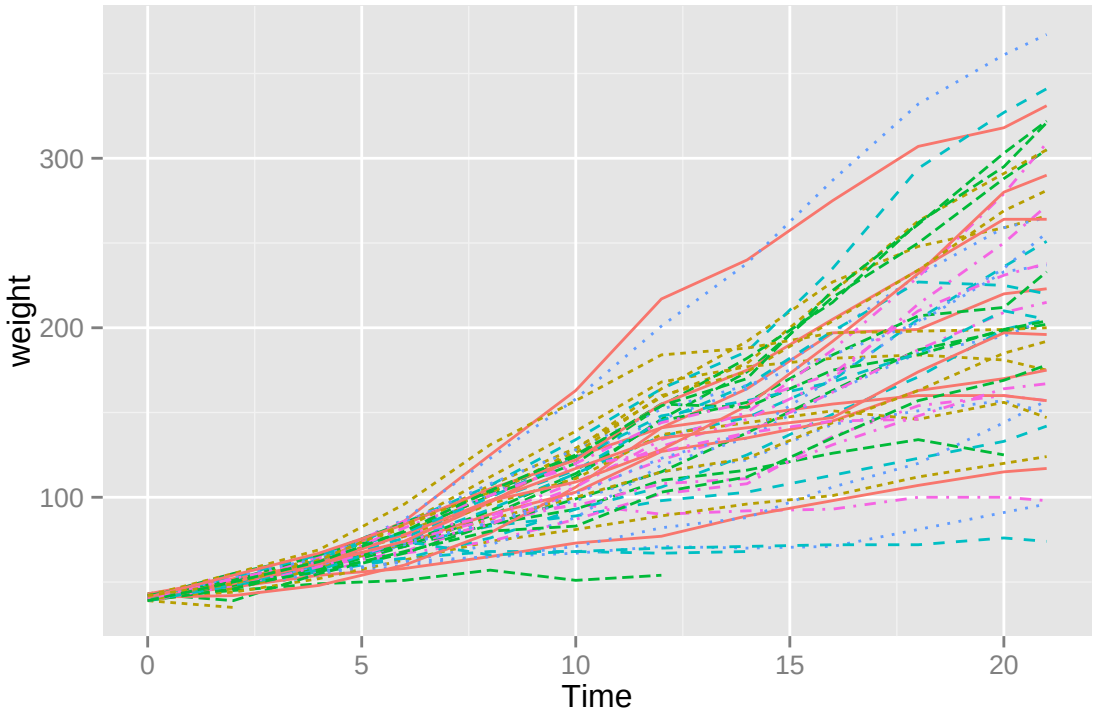


Figure 26: Plot with better lines.

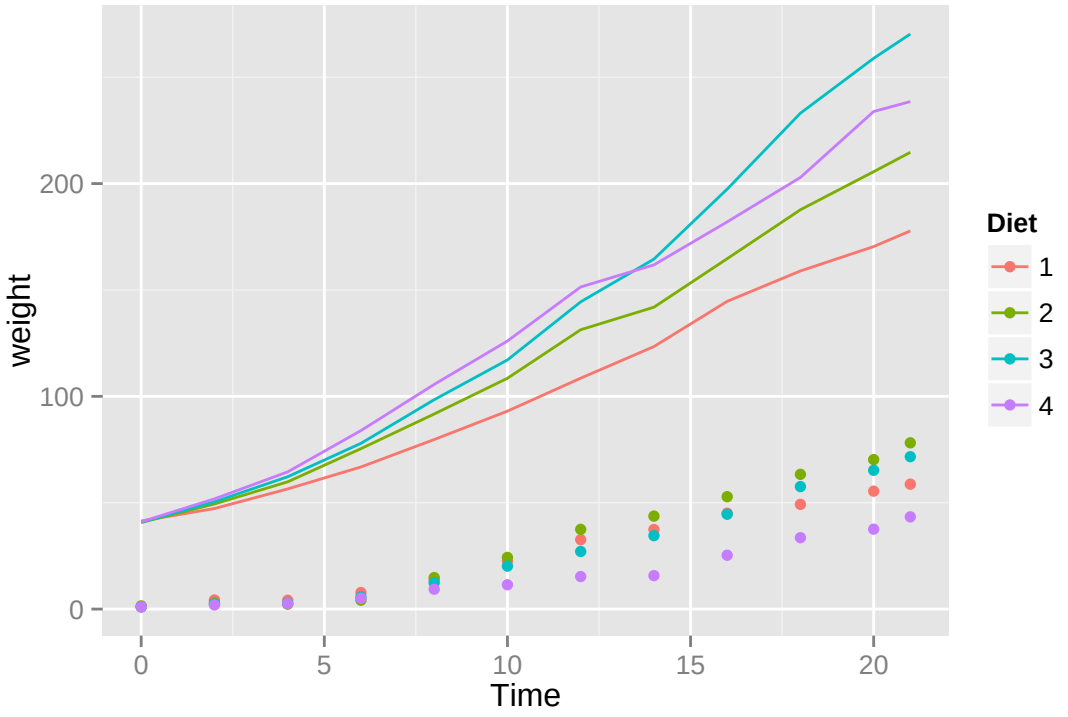


Figure 27: Plot summarizing data with function stat_summary().

We might also want to show the range of the data at each point (line show the 95% quantiles, point showing the median):

```
q5 <- function(x)
   quantile(x,0.05)
q95 <- function(x)
    quantile(x,0.95)

p + stat_summary(fun.y = median,fun.ymin = q5,fun.ymax = q95,aes (group = Diet,colour = Diet))
```

This is not very useful since the lines overlap each other. We have several alternatives. The first is to **jitter** the points to avoid overlapping:

```
p + stat_summary(fun.y = median,fun.ymin = q5,fun.ymax = q95,
    position = 'jitter',aes (group = Diet,colour = Diet))
```

The second option is to make a distinct plot for each diet, using a technique called **facetting**. Now, the colours are no longer needed to identify the diets and we can use a single colour:

```
levels(p$data$Diet) <- paste('Diet',levels(p$data$Diet))
p + stat_summary(fun.y = median,fun.ymin = q5,fun.ymax = q95,colour = 'red',aes (group = Diet)) +
    facet_grid(.~Diet)
```

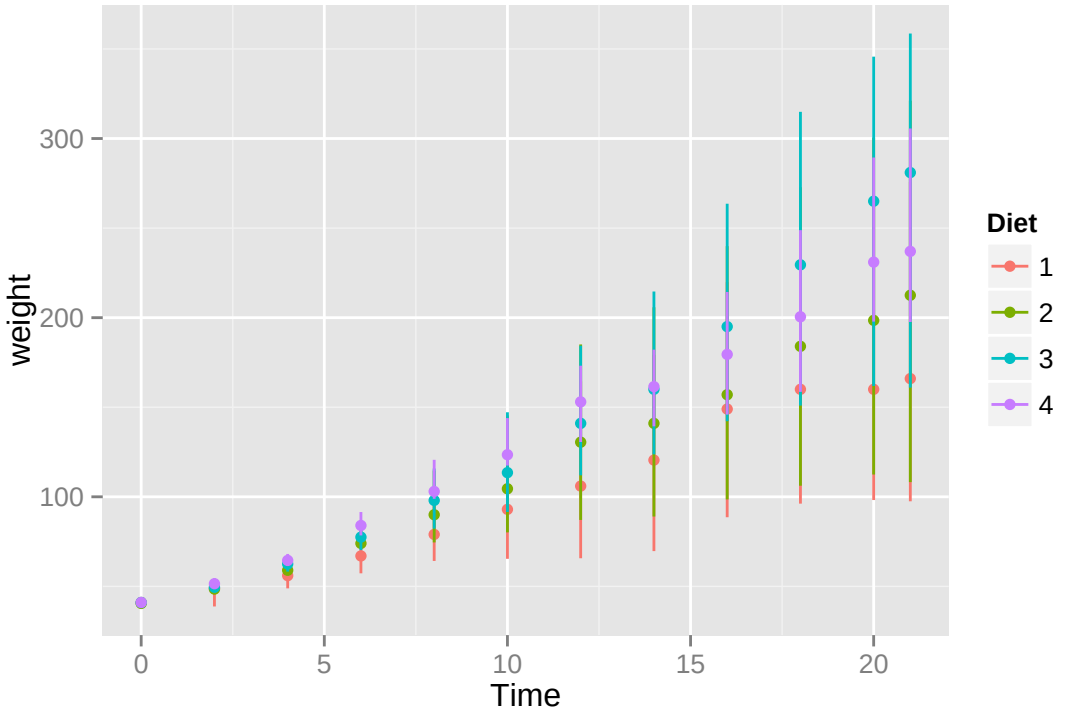


Figure 28: Custom summary of the data with function stat_summary().

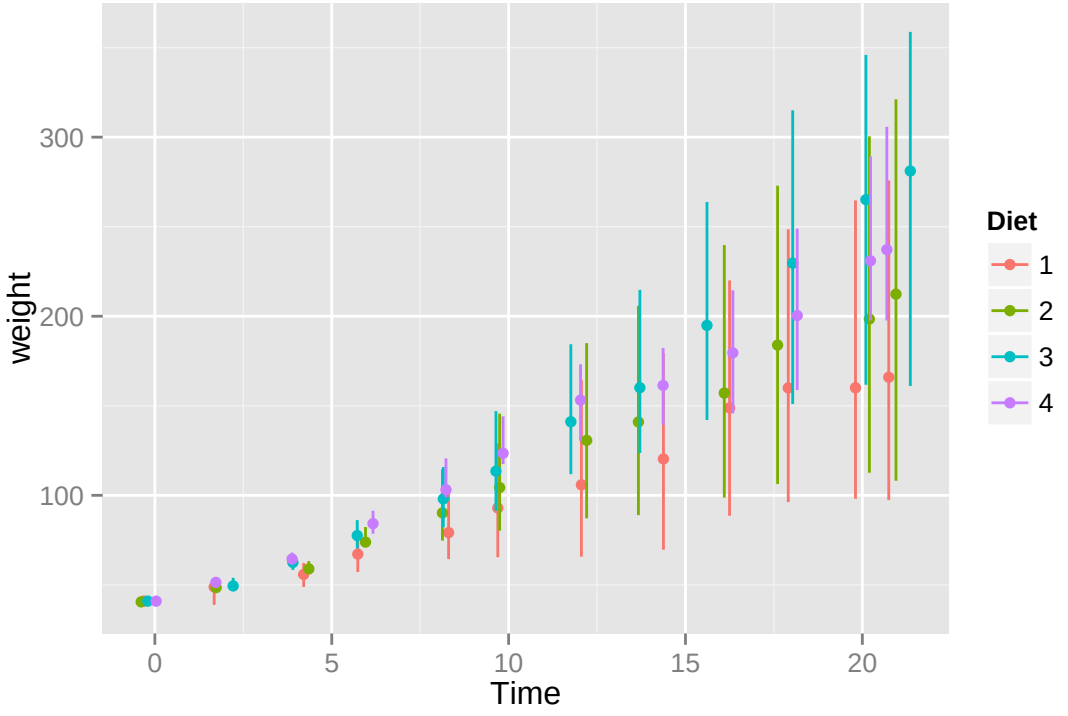


Figure 29: Summary with jittered points.

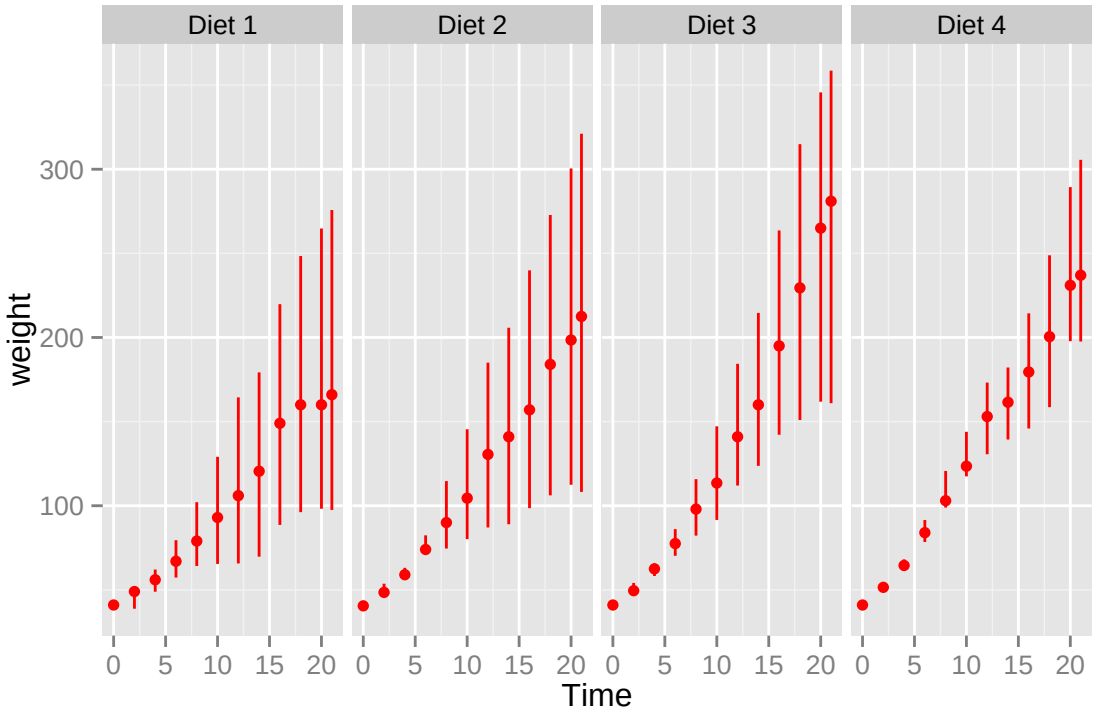


Figure 30: Plot with facets.

Barplots

To make barplots, we have two alternatives. First, we can provide summarized data, for example the average weight per diet and per point in time:

```
sw <- with(ChickWeight,tapply(weight,list(Time,Diet),mean,na.rm = T))
head(sw)
```

```
##       1     2     3     4
## 0  41.4  40.7  40.8  41.0
## 2  47.2  49.4  50.4  51.8
## 4  56.5  59.8  62.2  64.5
## 6  66.8  75.4  77.9  83.9
## 8  79.7  91.7  98.4 105.6
## 10 93.1 108.5 117.1 126.0
```

Object `sw` has a format which is not usable for `ggplot2`; therefore we need to have the data in *long* format using function `melt` package `reshape`:

```
install.packages('reshape')
require(reshape)
```

```
swl <- melt(sw)
head(swl)
```

```
##   X1 X2 value
## 1  0  1  41.4
## 2  2  1  47.2
## 3  4  1  56.5
## 4  6  1  66.8
## 5  8  1  79.7
## 6 10  1  93.1
```

```
colnames(swl) <- c('Time','Diet','weight')
swl$Diet <- factor(swl$Diet)
```

```
bp <- ggplot(data = swl,aes(y = weight,x = Time,fill = Diet))
bp + geom_bar(stat = 'identity')
```

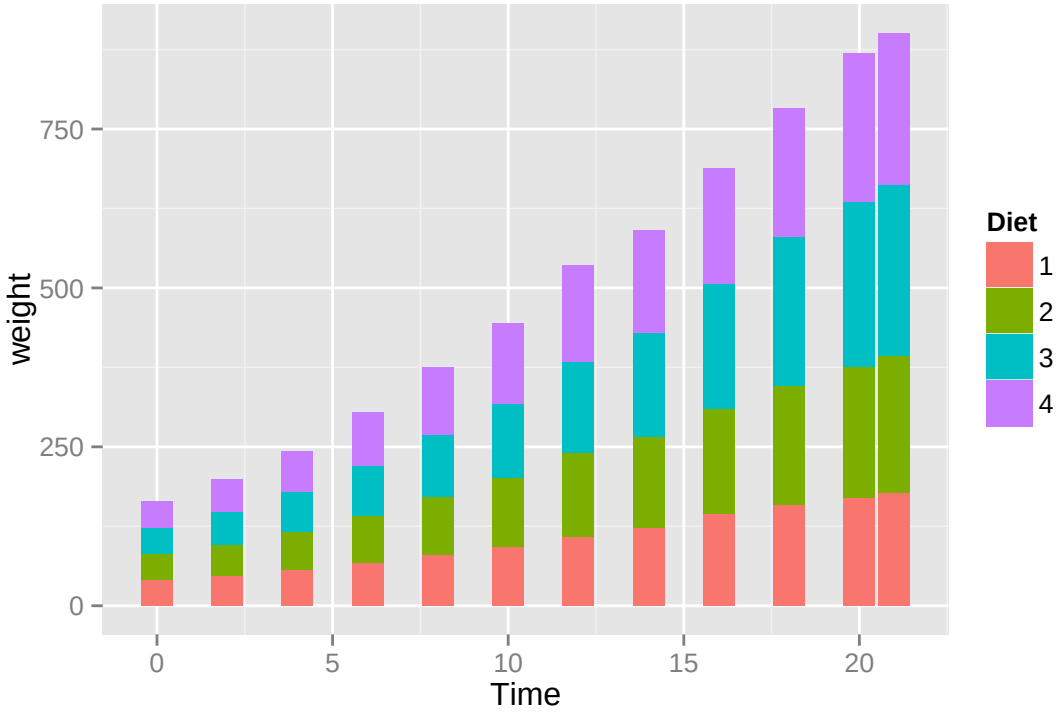


Figure 31: Barplot

By default, the bars are stacked upon each other, we can also place them next to each other:

```
bp + geom_bar(stat = 'identity',position = 'dodge')
```

But we can also use function `stat_summary` to calculate the average weight per point in time, as shown previously. Argument `position = 'dogde'` specifies that the bars are located next to each other, with `position = 'stack'`, they are placed on top of each other:

```
bp1 <- ggplot(data = ChickWeight,aes(y = weight,x = Time,fill = Diet))
bp1 + stat_summary(fun.y = mean,geom = 'bar',position = 'dodge')
```

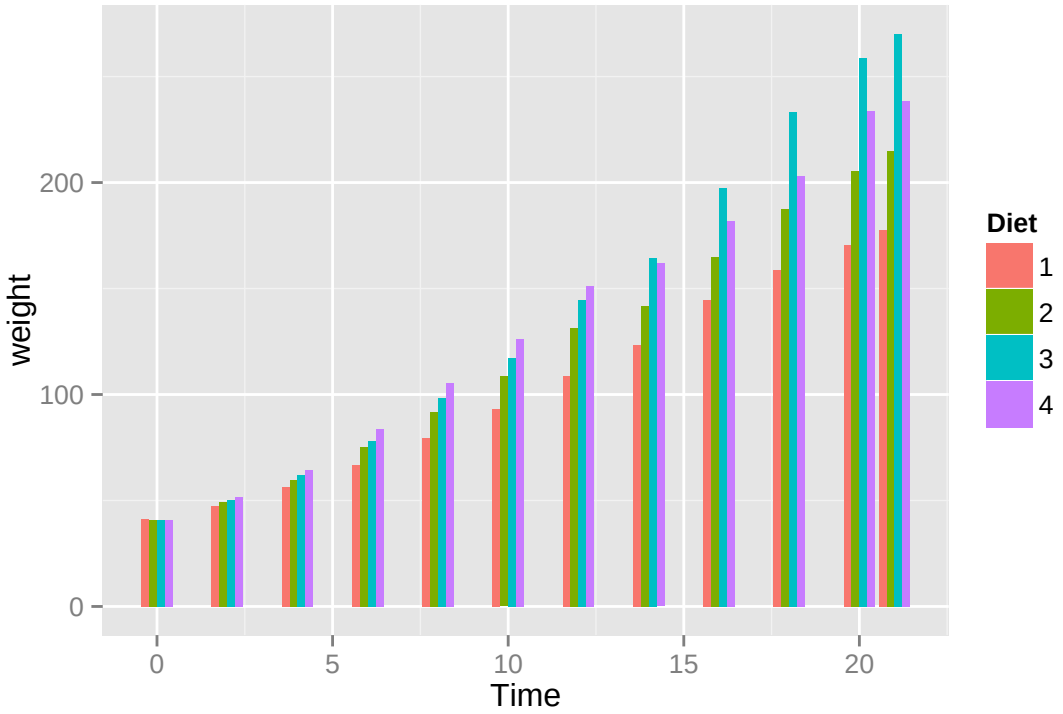


Figure 32: Dodged barplot.

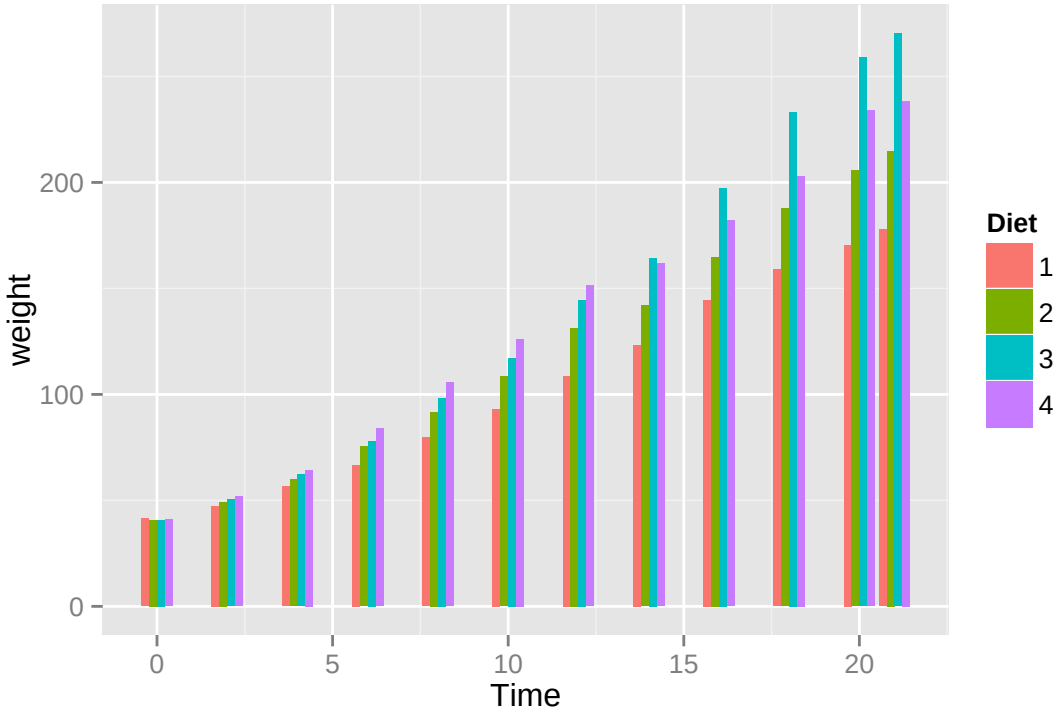


Figure 33: Barplots with stat_summary().

```
bp1 + stat_summary(fun.y = mean,geom = 'bar',position = 'stack')
```

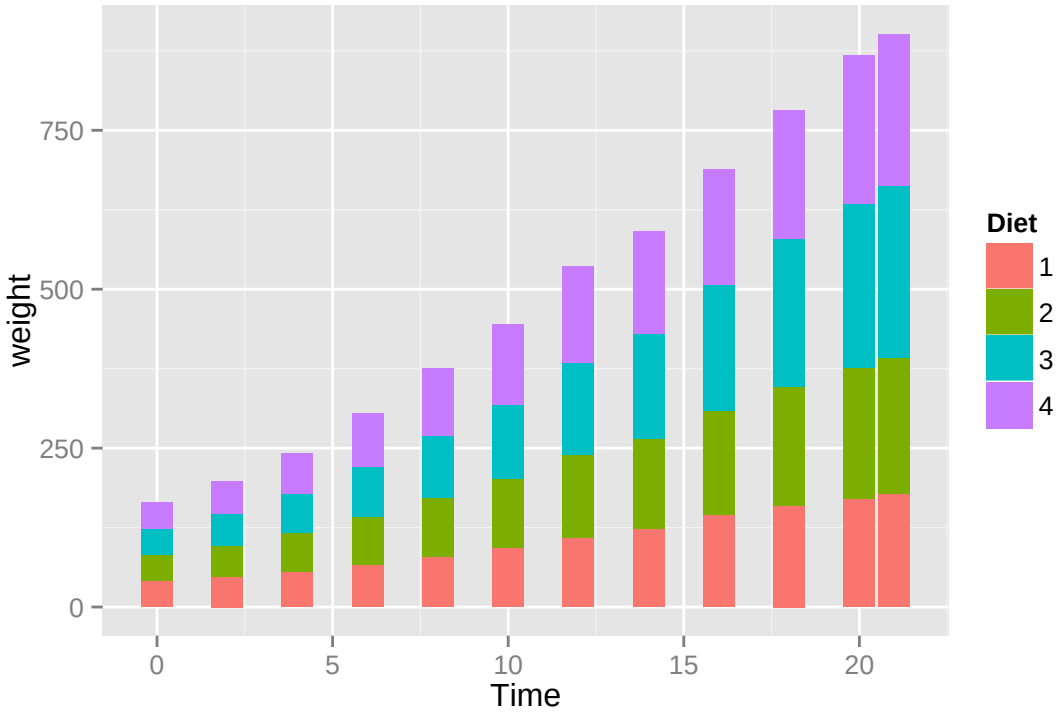


Figure 34: Barplots with stat_summary().

Since we use the raw data, it should be possible to add bars showing the range of the data. For this, we use function `geom_errorbar`:

```
bp1 + stat_summary(fun.y = mean,geom = 'bar',position = 'dodge') +
   stat_summary(fun.ymax = q95,fun.ymin=q5,geom="errorbar", width=0.25,
   position = position_dodge(width = 0.9),size = 1.2)
```

Trendlines

Supose that we want to draw a line through the data, for this we use function `stat_smooth`:

```
p <- ggplot(data = ChickWeight,aes(x = Time,y = weight))
p + geom_point(aes(colour = Diet)) + stat_smooth(se = FALSE)
```

```
## geom_smooth: method="auto" and size of largest group is <1000, so using loess. Use 'method = x' to cl
```

We can also add new lines for each `Diet`

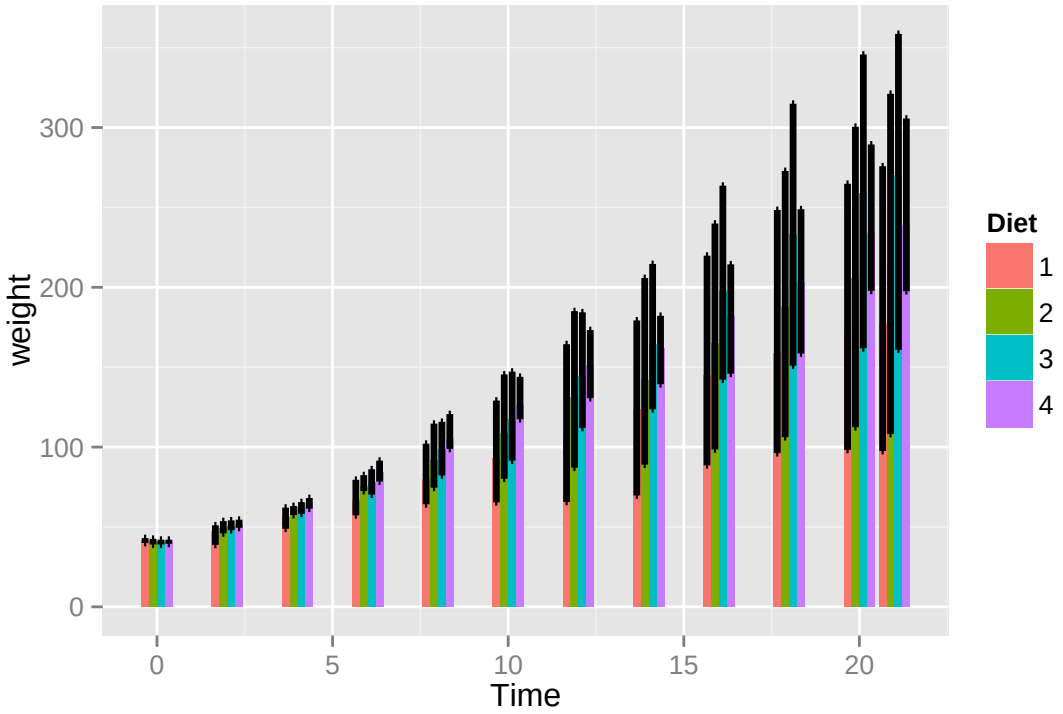


Figure 35: Barplot showing the range of the data.

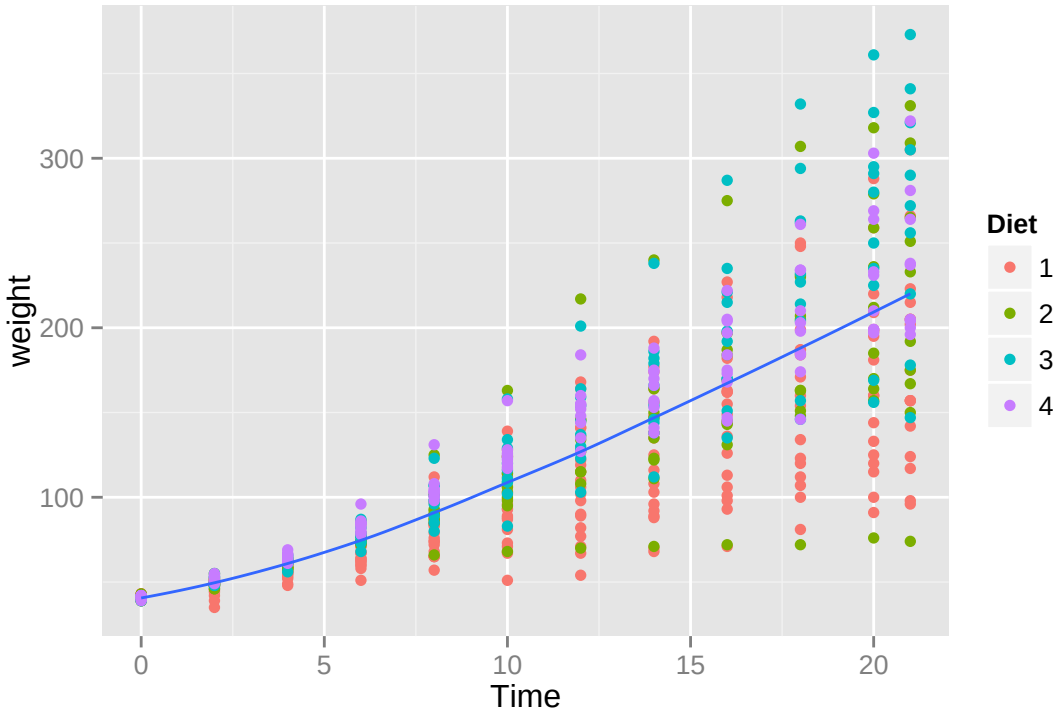


Figure 36: Scatterplot with trendlines

```
p + geom_point(aes(colour = Diet)) +
  stat_smooth(se = FALSE,colour = 'black',size = 1.2) +
  stat_smooth(aes(colour = Diet),size = 1,se = FALSE)
```

```
## geom_smooth: method="auto" and size of largest group is <1000, so using loess. Use 'method = x' to ch
## geom_smooth: method="auto" and size of largest group is <1000, so using loess. Use 'method = x' to ch
```

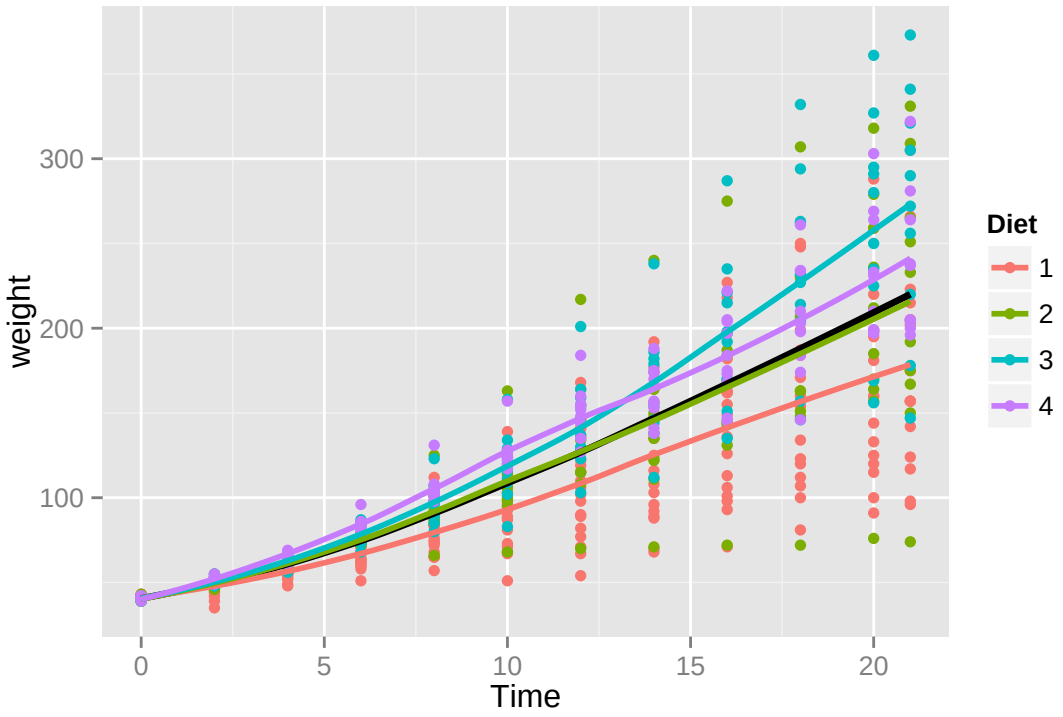


Figure 37: Trendline for each diet.

In the previous examples, the function used a nonparametric method called LOESS. We can also use linear regression to fit the data:

```
p + geom_point(aes(colour = Diet)) + stat_smooth(aes(colour = Diet),
    method = 'lm',size = 1.2,se = FALSE)
```

And we can also specifiy the equation:

```
p + geom_point(aes(colour = Diet)) + stat_smooth(aes(colour = Diet),
    method = 'lm',formula = y~x + I(x^2),size = 1.2,se = FALSE)
```

Time for exercises

- Go to exercises ggplot2 graphics
- Go to solutions ggplot2 graphics
- Return to main page

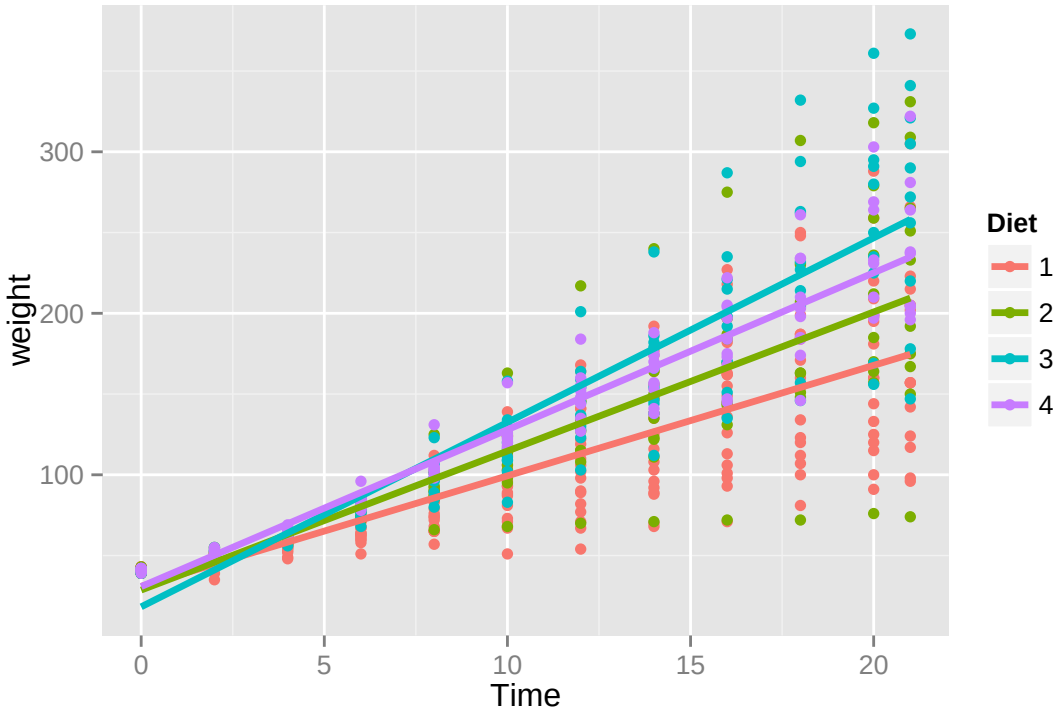


Figure 38: Linear trendline.

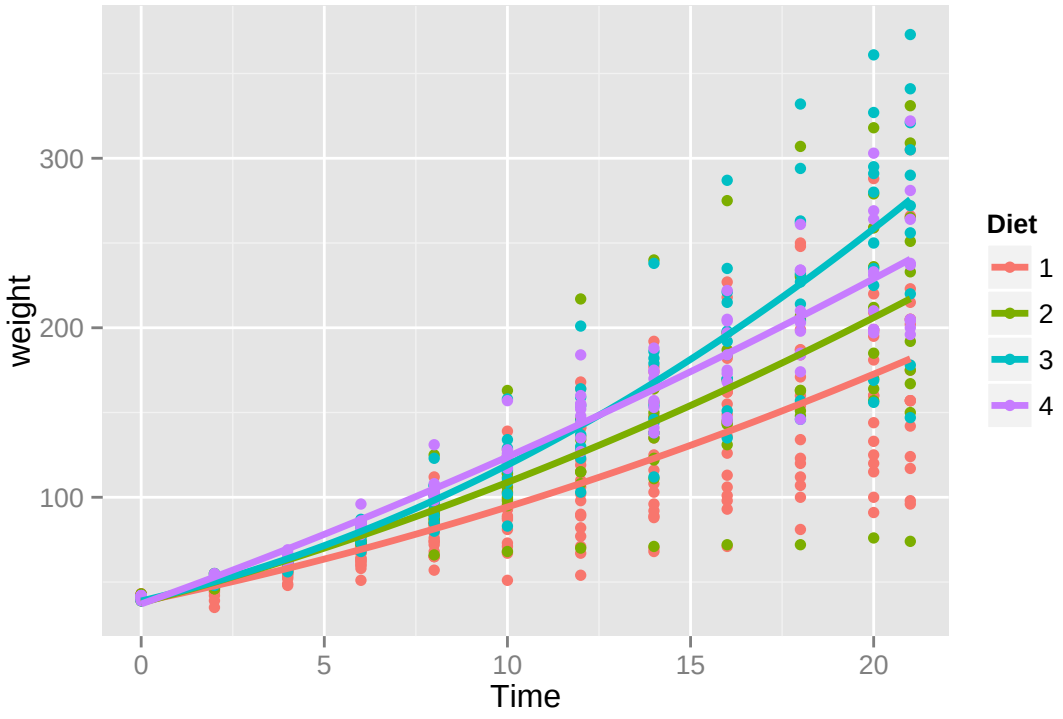


Figure 39: Quadratic trendline.

Linear regression

Data

We will use the `ChickWeight` data:

```
summary(ChickWeight)
```

```
##      weight         Time           Chick
##  Min.   : 35   Min.   : 0.0   13     : 12
##  1st Qu.: 63   1st Qu.: 4.0   9      : 12
##  Median :103   Median :10.0   20     : 12
##  Mean   :122   Mean   :10.7   10     : 12
##  3rd Qu.:164   3rd Qu.:16.0   17     : 12
##  Max.   :373   Max.   :21.0   19     : 12
##                               (Other):506
##  Diet    shape
##  1:220   0: 96
##  2:120   1: 98
##  3:120   2:102
##  4:118   3: 92
##          4: 96
##          5: 94
##
```

Plotting the data

Plot weight versus time with a color for each diet (using package ggplot2):

```
ggplot(data = ChickWeight,aes(x = Time,y = weight,colour = Diet,shape = Diet)) +
   geom_point() + theme(legend.position = c(0.1,0.8))
```

Linear models

In R, we fit a linear model using the function `lm`. Arguments of `lm` are:

```
args(lm)
```

```
## function (formula, data, subset, weights, na.action, method = "qr",
##     model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
##     contrasts = NULL, offset, ...)
## NULL
```

Remember type `?functionName` for help and examples

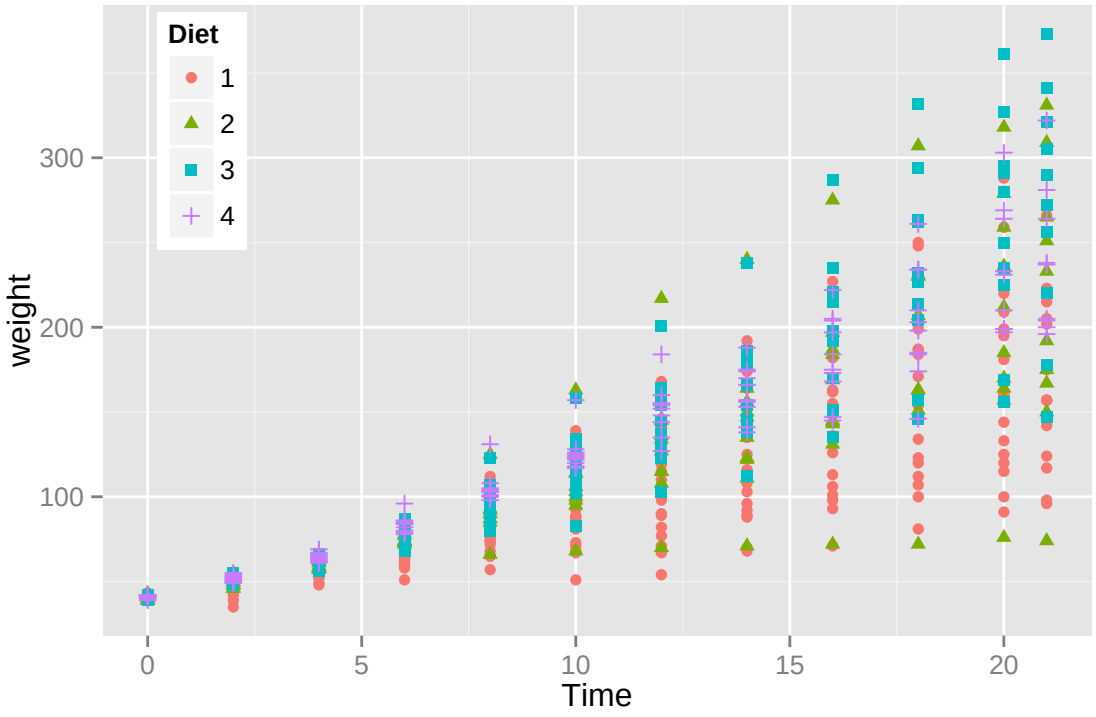


Figure 40: Plot of ChickWeight data

Simple linear regression

From the scatterplot it was clear that there is a relationship between Time and Chick weight. We may wonder whether the relationship is linear in time, and whether all further needed assumptions, like independent errors, are fulfilled, but as a starter we just fit the simple linear regression model to `ChickWeight`:

$weight_{ij} = \beta_0 + \beta_1 \cdot Time_i + \epsilon_{ij}$

```
(lmW1 <- lm(weight~Time,ChickWeight))
```

```
##
## Call:
## lm(formula = weight ~ Time, data = ChickWeight)
##
## Coefficients:
## (Intercept)         Time
##        27.5          8.8
```

The lm class

Remember *classes* and *objects*; the R language is object-oriented:

- Everything in R is an **object**
- Every *object* is defined by its **class**

- Complex *objects* have **attributes**

lmW1 is an object with two attributes (names and class). Its class is `lm`:

```
attributes(lmW1)
```

```
## $names
##  [1] "coefficients"  "residuals"
##  [3] "effects"       "rank"
##  [5] "fitted.values" "assign"
##  [7] "qr"            "df.residual"
##  [9] "xlevels"       "call"
## [11] "terms"         "model"
##
## $class
## [1] "lm"
```

```
class(lmW1)
```

```
## [1] "lm"
```

Accessing elements of an lm object

An object of class `lm` is a list containing at the least the components mentioned under the `names` attribute. These list elements can be extracted in the usual way or using special functions.

Access the first element of `lmW1` (by order):

```
lmW1[[1]]
```

```
## (Intercept)        Time
##        27.5         8.8
```

We can also access the `coefficients` list element by name:

```
lmW1$coefficients
```

```
## (Intercept)        Time
##        27.5         8.8
```

Generic functions

A *generic function* is a function defined for different classes:

```
head(methods(summary))
```

```
## [1] "summary,ANY-method"
## [2] "summary,diagonalMatrix-method"
## [3] "summary,sparseMatrix-method"
## [4] "summary.aareg"
## [5] "summary.agnes"
## [6] "summary.aov"
```

```
args(summary.lm)
```

```
## function (object, correlation = FALSE, symbolic.cor = FALSE,
##     ...)
## NULL
```

Generic functions that can be applied to objects of class lm:

Function	Use
summary	returns summary information about regression
plot	makes diagnostic plots
coef	returns the coefficients
residuals	returns the residuals
fitted	returns fitted values
deviance	returns residual sum of squares
predict	performs predictions
anova	finds various sums of squares, produces anova table
AIC	used for model selection

Time for trying them

Exploring the fitted model

The generic function `summary` summarizes an object of class lm:

```
summary(lmW1)
```

```
##
## Call:
## lm(formula = weight ~ Time, data = ChickWeight)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -138.33  -14.54    0.93   13.53  160.67
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)    27.47       3.04    9.05   <2e-16
## Time            8.80       0.24   36.73   <2e-16
##
## (Intercept) ***
## Time        ***
```

```
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 38.9 on 576 degrees of freedom
## Multiple R-squared:  0.701,  Adjusted R-squared:   0.7
## F-statistic: 1.35e+03 on 1 and 576 DF,  p-value: <2e-16
```

Plotting the regression line

Using basic plotting functions

```
plot(weight ~ Time, data=ChickWeight)
abline(lmW1)
```

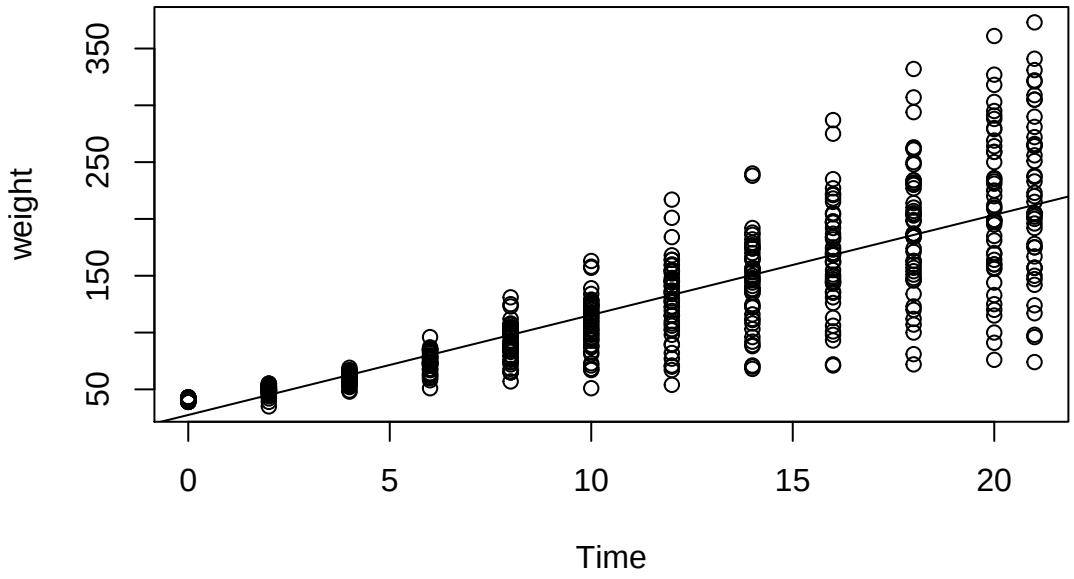


Figure 41: ChickWeight data with regression line

Package `ggplot2` also has facilities. We can use function `stat_smooth` to add a regression line to the plot:

```
ggplot(data = ChickWeight,aes(x = Time,y = weight)) +
geom_point(aes(colour = Diet,shape = Diet)) + theme(legend.position = c(0.1,0.8)) +
stat_smooth(method = 'lm',formula = y ~ x,size = 1,se = FALSE)
```

And we can also use the coefficients from the `lmW1` object to plot:

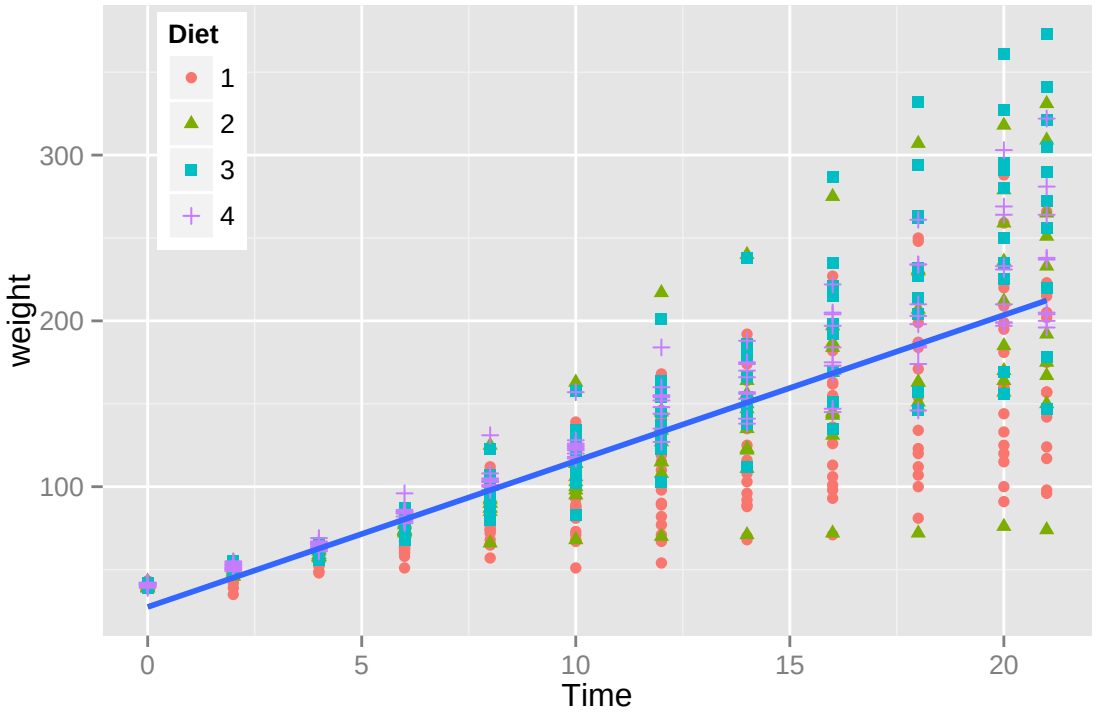


Figure 42: ChickWeight data with regression line using ggplot

```
df <- data.frame(a = coef(lmW1)[1],b = coef(lmW1)[2])
ggplot(data = ChickWeight,aes(x = Time,y = weight)) +
geom_point(aes(colour = Diet,shape = Diet)) + theme(legend.position = c(0.1,0.8)) +
geom_abline(data = df,size = 1,colour = 'blue',aes(intercept = a,slope = b))
```

t-tests for coefficients

$H_0 : \beta_i = 0$:

```
(betas <- summary(lmW1)$coefficients)
```

```
##             Estimate Std. Error t value  Pr(>|t|)
## (Intercept)     27.5       3.04    9.05  2.26e-18
## Time             8.8       0.24   36.73 5.02e-153
```

Reproduce the t-test for β_1:

```
b1 <- betas[2,1]
seb1 <- betas[2,2]
tb1 <- b1/seb1
(pval <- 2*pt(abs(tb1),lmW1$df.residual,lower.tail=FALSE))
```

```
## [1] 5.02e-153
```

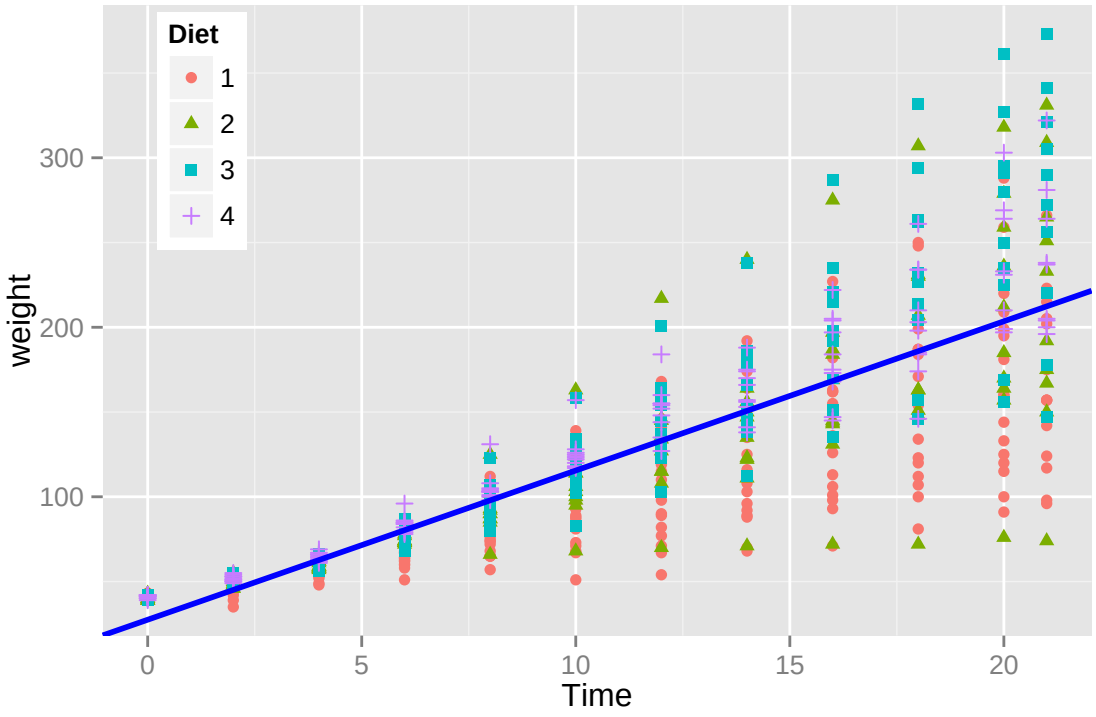


Figure 43: ChickWeight data with regression line using ggplot

ANOVA table

```
(aTab <- anova(lmW1))
```

```
## Analysis of Variance Table
##
## Response: weight
##            Df  Sum Sq Mean Sq F value Pr(>F)
## Time        1 2042344 2042344    1349 <2e-16 ***
## Residuals 576  872212    1514
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Reproduce the F-test, here testing $H_0 : \beta_1 = 0$:

```
SSreg  <- aTab[1,2]
SSres <- aTab[2,2]
dfreg  <- aTab[1,1]
dfres <- aTab[2,1]
F <- (SSreg/dfreg)/(SSres/dfres)
pf(F,dfreg, dfres, lower.tail = FALSE)
```

```
## [1] 5.02e-153
```

Confidence Interval

The confidence interval of a parameter is the interval of values which are in agreement with the data. The confidence level $1 - \alpha$ specifies the strength of this agreement. The interval has often the form:

$b_i \pm s.e.(b_i) * t_{df}(1 - \alpha)$

Use the function `confint`:

```
confint(lmW1,level = 0.95)
```

```
##             2.5 % 97.5 %
## (Intercept) 21.50  33.43
## Time         8.33   9.27
```

Reproduce them:

```
lmW1s <- summary(lmW1)
beta <- lmW1s$coefficients[,1]
se   <- lmW1s$coefficients[,2]
cbind(beta - qt(0.975,lmW1$df.residual)*se, beta + qt(0.975,lmW1$df.residual)*se)
```

```
##              [,1]  [,2]
## (Intercept) 21.50 33.43
## Time         8.33  9.27
```

Prediction

Generic function `fitted` returns the fitted values for `weight`:

```
pred <- fitted(lmW1)
pred[1:10]
```

```
##     1     2     3     4     5     6     7     8
##  27.5  45.1  62.7  80.3  97.9 115.5 133.1 150.7
##     9    10
## 168.3 185.9
```

Predict new observations with function `predict`, we need to make a `data.frame` of the new data. Then, we can predict `weight` for the new observations.

```
newDat <- data.frame(Time = 2:4)
(pred <- predict(lmW1,newDat,se = T))
```

```
## $fit
##    1    2    3
## 45.1 53.9 62.7
##
## $se.fit
##    1    2    3
```

```
## 2.64 2.46 2.28
##
## $df
## [1] 576
##
## $residual.scale
## [1] 38.9
```

Residuals and diagnostic plots

Assumptions in regression / linear models are:

- constant variance of error (or of y) = homoscedasticity;
- error is normally distributed;
- errors are independent

Residuals are used to check the assumptions, often graphically:

```
df <- data.frame(res = resid(lmW1),fitted = fitted(lmW1),Chick = ChickWeight$Chick)
ggplot(data = df,aes(x= fitted,y = res,colour = Chick)) +
   geom_point() + geom_hline(aes(yintercept = 0)) + theme(legend.position = 'none')
```

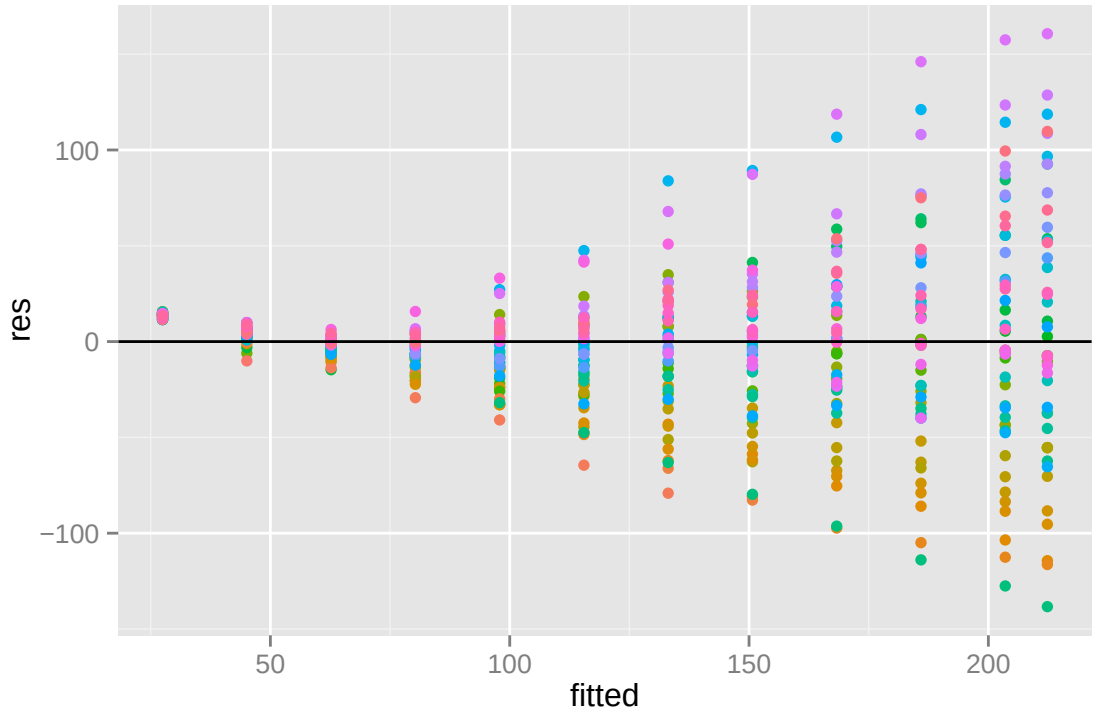


Figure 44: Residual plot: residuals vs fitted values

Checking normality

Make a QQ (quantile-quantile) plot of the residuals to check normality, adding a line through quartiles 1 and 3:

```
qqnorm(df$res)
qqline(df$res)
```

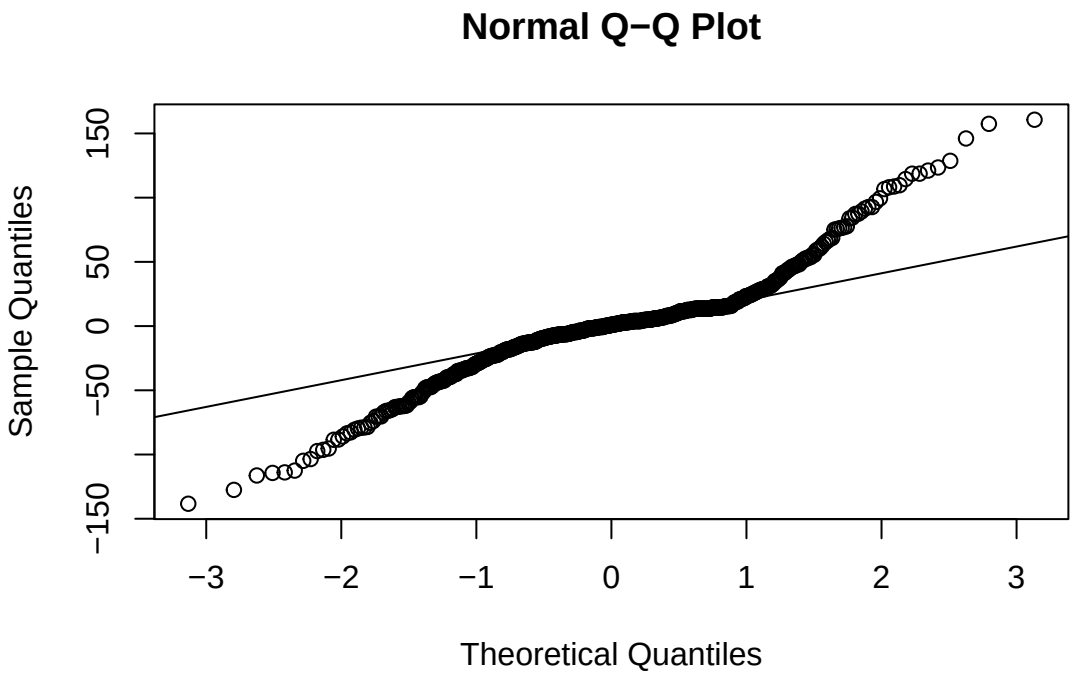


Figure 45: Basic QQ-plot

Or with a function from `ggplot2`:

```
ggplot(data = df,colour = df$Chick,aes(sample = res)) + geom_point(stat = 'qq')
```

The standard diagnostic plots

Standard diagnosis plots for objects of class lm using generic function `plot`:

```
oldPar <- par(mfrow = c(2,2),mar=c(3,3,1.5,1), cex=0.9,  mgp=c(1.5,0.3,0), tck=-0.015, cex.main=1)
plot(lmW1)
```

```
par(oldPar)
```

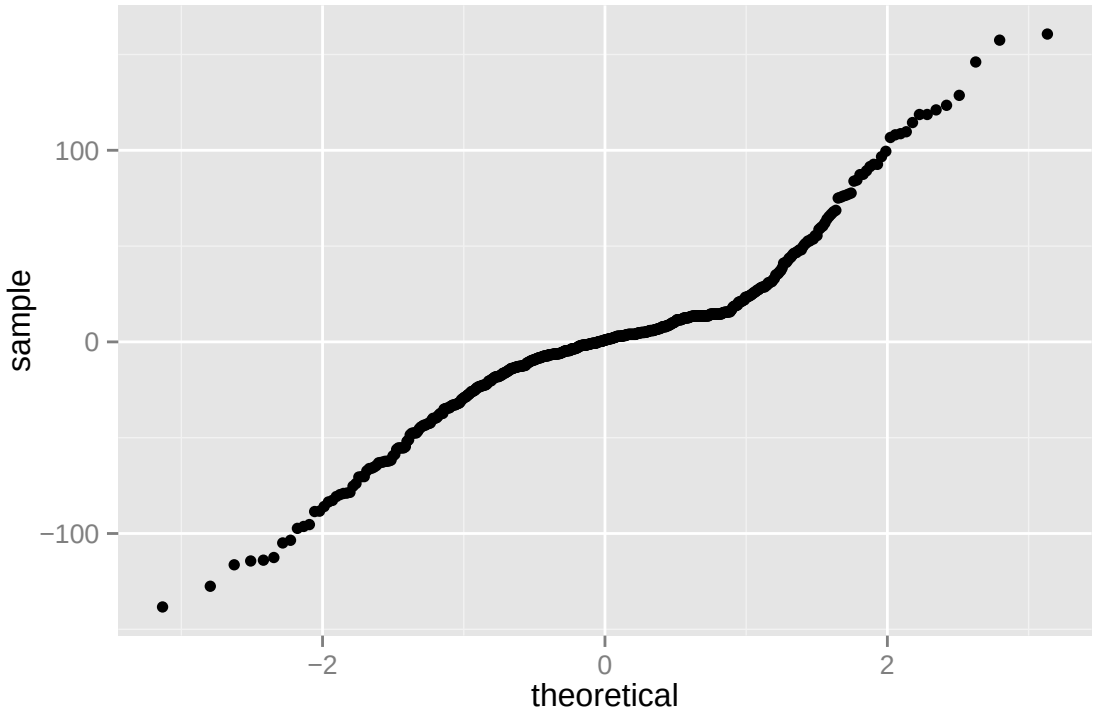


Figure 46: QQ-plot using ggplot

Leverage

$Var(residual) = \sigma^2[\mathbf{I} - \mathbf{H}]$

- $\mathbf{H} = \mathbf{X}(\mathbf{X^T X})^{-1}\mathbf{X^T}$ is the *hat* matrix (package MASS);
- As $\mathbf{H}$ is not diagonal, residuals are not independent;
- Diagonal elements of $\mathbf{H}$, h_{ii}, are called *leverages*, measuring excentricity in X-space;
- If h_{ii} large, observation i is potentially influential;
  - large $h_{ii} > 2p/n$ (MASS)

Obtain the leverages h_{ii} by `lm.influence(lmobject)$hat` or by `hatvalues()`

```
require(MASS)
lev <- lm.influence(lmW1)$hat
head(lev)
```

```
##       1       2       3       4       5       6
## 0.00609 0.00461 0.00344 0.00257 0.00201 0.00175
```

Studentized residuals and leverages

Variances of the residuals are not constant. To make residuals better comparable, it may be better to standardize them:

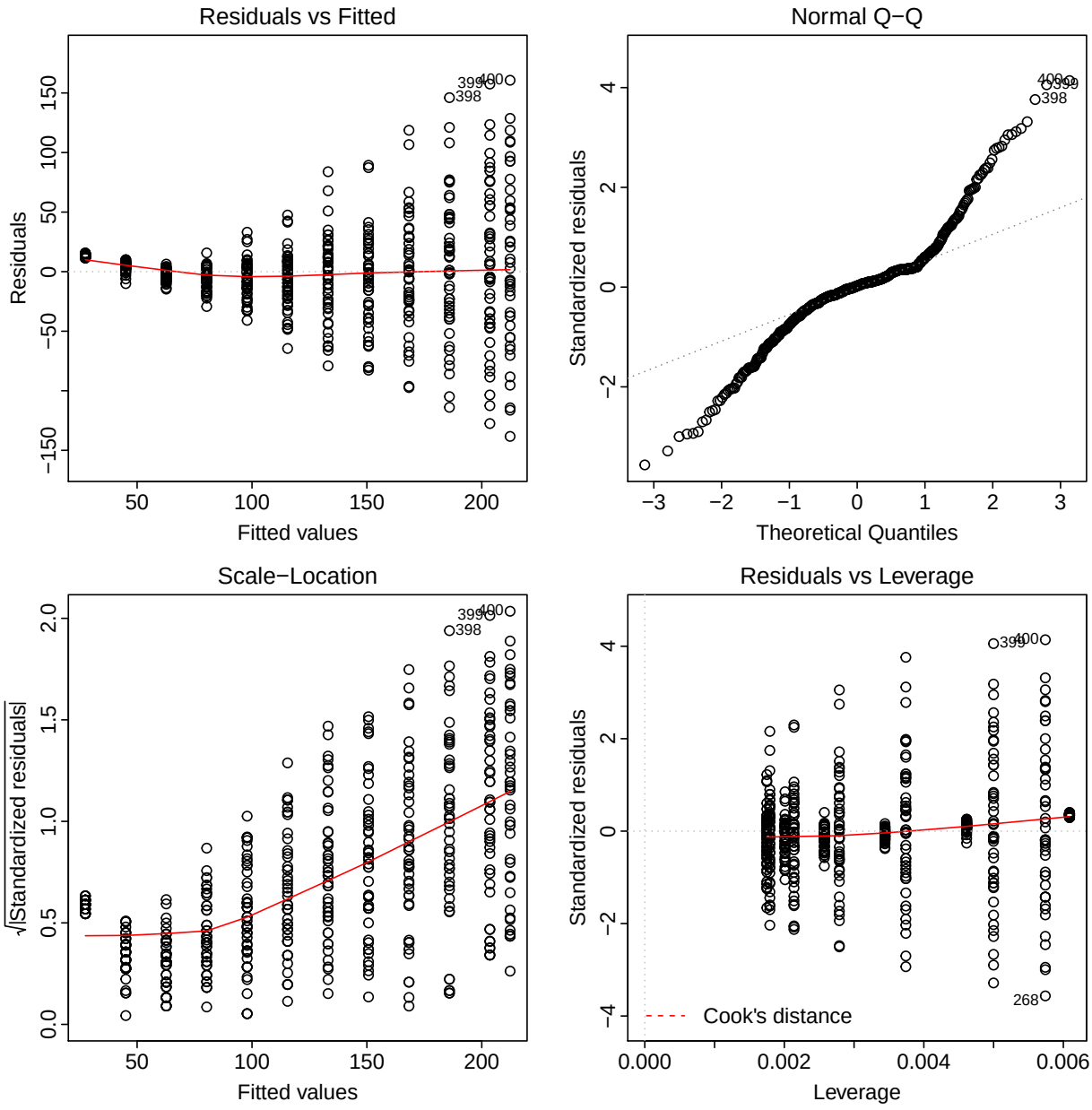


Figure 47: Diagnostic panel

$e'_i = \frac{e_i}{s\sqrt{1-h_{ii}}}$

In R: `stdres(lmobject` (from the MASS package)

Observations with larger leverage have smaller variance. It may be insightful to calculate the (standardized) residual of an observation without using the observation itself: jackknifed residuals or deleted residuals. In R these are called studentized residuals:

$e^*_i = \frac{y_i - \hat{y}_{-i}}{\sqrt{var(y_i - \hat{y}_{-i})}}$

In R: `studres(lmobject)` (from the MASS package)

```
stds <- stdres(lmW1)
studs <- studres(lmW1)
oldPar <- par(mfrow = c(1,2))
plot(studs~predict(lmW1))
plot(stds~lev)
```

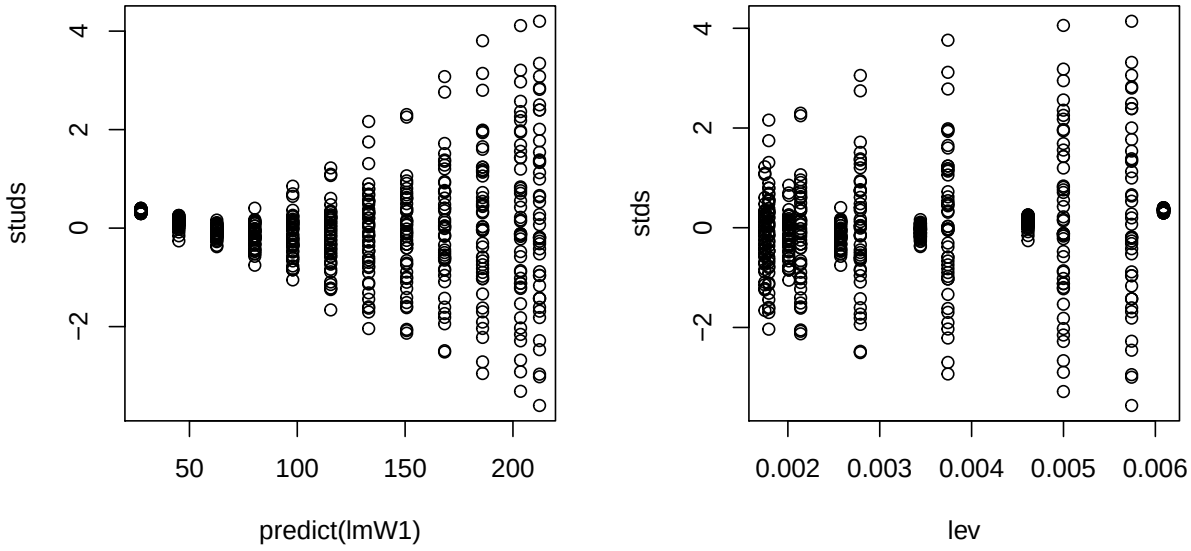


Figure 48: More diagnostic plots

```
par(oldPar)
```

With or without intercept

If not specified, by default the intercept is included in a fitted model. Intercept can be left out the model in different ways:

```
lmW21 <- lm(weight ~ 0 + Time, ChickWeight)
lmW22 <- lm(weight ~ -1 + Time, ChickWeight)
coef(lmW21)
```

```
## Time
## 10.6
```

Linear model in matrix notation

The linear model in matrix notation is

$\mathbf{y} = \mathbf{X}\beta + \mathbf{e}$

$\mathbf{e} \sim \mathbf{N}(0, \sigma_e^2\mathbf{I})$

The matrix $\mathbf{X}$ is called the *design matrix*. Here it has 2 columns: a column of ones and column `Time`. The least squares solution is:

$\mathbf{b} = (\mathbf{X^T X})^{-1}\mathbf{X^T y}$

This is easily programmed in R

```
X <- model.matrix(~Time,ChickWeight)
y <- ChickWeight$weight
b <- solve(crossprod(X),crossprod(X,y))
b
```

```
##             [,1]
## (Intercept) 27.5
## Time         8.8
```

End

- Exercises regression
  - Solutions exercises regression
- Return to main page

ANOVA

One-way ANOVA

Do different diets result in different final weights? First we select the final weights by reverse ordering the data frame and then selecting the first observation per chick. Next, treat `Diet` as a factor (**qualitative** regressor):

```
CWf <- ChickWeight[with(ChickWeight,order(Chick,-Time)),]
CWf <- CWf[!duplicated(CWf$Chick),]
if(!is.factor(CWf$Diet))
    CWf$Diet <- factor(CWf$Diet)
ggplot(CWf, aes(x = Diet, weight)) + geom_boxplot(aes(fill=Diet))
```

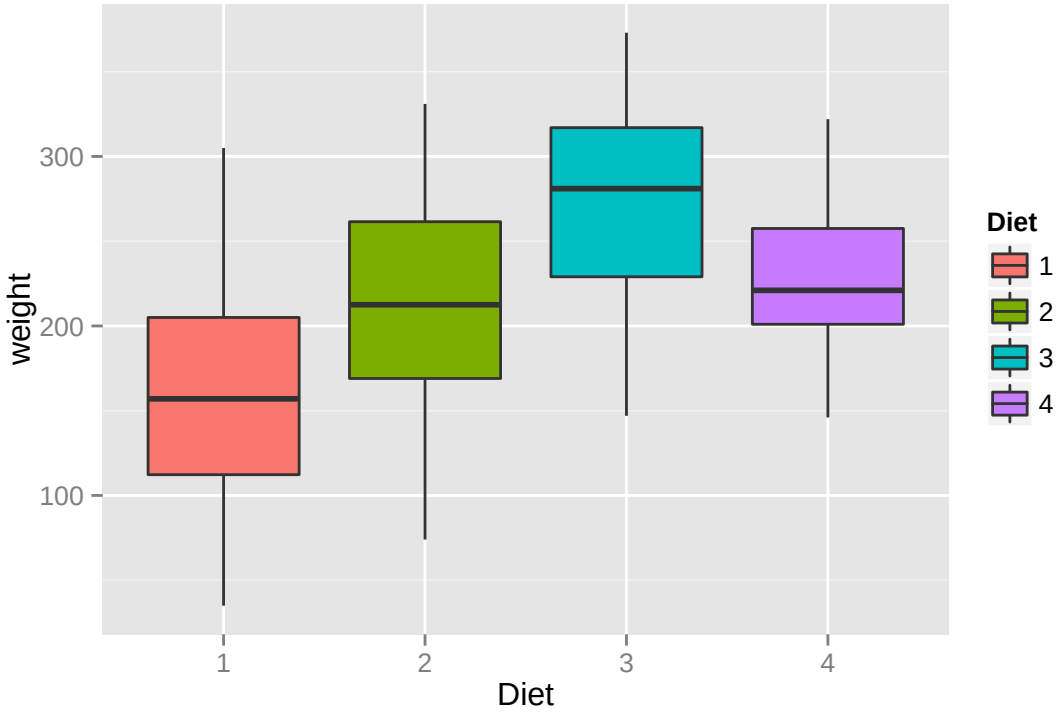


Figure 49: Boxplots of weight per diet

One-way ANOVA model as linear model

One-way ANOVA model: $weight_{ij} = \mu + Diet_i + \epsilon_{ij}$

ANOVA = regression on indicator variables for the levels of the factor(s). To see this, write the model as:

$weight_{ij} = \mu + \alpha_1 x_1 + \alpha_2 x_2 + \alpha_3 x_3 + \alpha_4 x_4 + \epsilon_{ij}$

where x_i is an indicator variable for i-th level of `Diet`

Function `lm`

ANOVA is another example of a linear model (linear in the parameters); we can use `lm`:

```
(anW1 <- lm(weight ~ Diet, CWf))
```

```
##
## Call:
## lm(formula = weight ~ Diet, data = CWf)
##
## Coefficients:
## (Intercept)        Diet2        Diet3
##       156.3         58.4        114.0
##       Diet4
##        73.0
```

Parameterization

The design matrix for ANOVA is (first 7 rows):

```
model.matrix(anW1)[1:7,]
```

```
##     (Intercept) Diet2 Diet3 Diet4
## 196           1     0     0     0
## 182           1     0     0     0
## 175           1     0     0     0
## 155           1     0     0     0
## 107           1     0     0     0
## 220           1     0     0     0
## 119           1     0     0     0
```

Hence: no indicator variable for the first level of `Diet`

How does R build this model.matrix? For models with a factor, first a matrix $\mathbf{X}$ is formed as $\mathbf{X} = [\mathbf{1}\ \mathbf{X_a}]$ where $\mathbf{X_a}$ is an incidence matrix with a separate column for each level of the factor. The model.matrix follows as $\mathbf{X}^* = [\mathbf{1}\ \mathbf{X_a}\mathbf{C_a}]$, where $\mathbf{C}_a$ is a contrast matrix.

Default parameterization

The default contrast matrix $\mathbf{C}_a$ for factor `Diet` is:

```
contrasts(CWf$Diet)
```

```
##   2 3 4
## 1 0 0 0
## 2 1 0 0
## 3 0 1 0
## 4 0 0 1
```

Which gives contrasts between all levels except the first with the first level. This is called the corner-stone parameterization. The estimated coefficients are:

```
coef(anW1)
```

```
## (Intercept)       Diet2       Diet3       Diet4
##       156.3        58.4       114.0        73.0
```

With these coefficients, we test if the weight of diet 1 differs from the other diets:

```
coefficients(summary(anW1))
```

```
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)    156.3       15.4   10.18 2.30e-13
## Diet2           58.4       26.6    2.20 3.32e-02
## Diet3          114.0       26.6    4.29 9.18e-05
## Diet4           73.0       26.6    2.75 8.60e-03
```

Other parameterization

We can define custom contrast matrices, or use other options for the contrast. *E.g.* another corner-stone parameterization takes the last level of `Diet` as the reference. Specify the contrast matrix for it and use this contrast matrix in `lm` as follows:

```
(Ca <- contr.treatment(levels(CWf$Diet),base=4))
```

```
##   1 2 3
## 1 1 0 0
## 2 0 1 0
## 3 0 0 1
## 4 0 0 0
```

```
anW2 <- lm(weight ~Diet, contrasts=list(Diet=Ca), CWf)
coef(anW2)
```

```
## (Intercept)       Diet1       Diet2       Diet3
##       229.3       -73.0       -14.6        41.0
```

All pairwise differences

Significance of pairwise differences can be calculated with Tukey's Honestly Significant Differences function `TukeyHSD`. The function works only for a fitted model object from `aov`:

```
summary(anW3 <- aov(weight ~ Diet, data = CWf))
```

```
##             Df Sum Sq Mean Sq F value  Pr(>F)
## Diet         3  96913   32304    6.85 0.00065 ***
## Residuals   46 216866    4714
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```r
round(TukeyHSD(anW3)$Diet, 3)
```

```
##      diff     lwr   upr p adj
## 2-1  58.4  -12.48 129.3 0.140
## 3-1 114.0   43.12 184.9 0.001
## 4-1  73.0    2.12 143.9 0.041
## 3-2  55.6  -26.25 137.4 0.282
## 4-2  14.6  -67.25  96.4 0.964
## 4-3 -41.0 -122.85  40.8 0.546
```

Plotting the pairwise differences:

```r
plot(TukeyHSD(anW3, "Diet"))
```

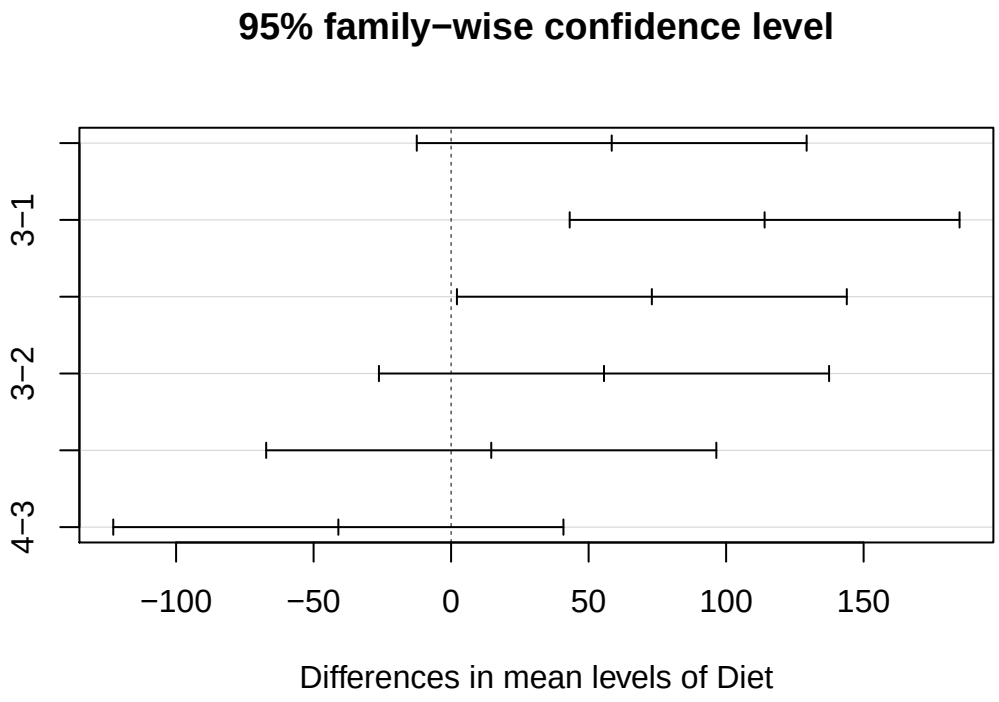


Figure 50: Tukey confidence intervals for pairwise differences

End

- Exercises anova
  - Solutions exercises anova
- Return to main page

ANCOVA

Analysis of covariance

Linear models with at least one covariate (quantitative regressor) and one factor (qualitative regressor) are historically called Analysis of Covariance models. The `ChickWeight` dataset contains the factor `Diet` and covariate `Time`.

Different models are possible:

- Model 1: common slope, common intercept: $weight_{ij} = \beta_0 + \beta_1 Time_{ij} + \epsilon_{ij}$
- Model 2: common slope, different intercepts: $weight_{ij} = \beta_{0i} + \beta_1 Time_{ij} + \epsilon_{ij}$
- Model 3: different slopes, different intercepts: $weight_{ij} = \beta_{0i} + \beta_{1i} Time_{ii} + \epsilon_{ij}$
- Model 4: different slope, common intercept: $weight_{ij} = \beta_0 + \beta_{1i} Time_{ii} + \epsilon_{ij}$

In each model we assume independent errors $\epsilon_{ij} \sim \mathbf{N}(0, \sigma^2)$.

Fitting analysis of covariance models

```
ancW1 <- lm(weight ~ Time, data = ChickWeight)
ancW2 <- lm(weight ~ Diet + Time, data = ChickWeight)
ancW3 <- lm(weight ~ Diet + Time + Diet:Time, data = ChickWeight)
ancW4 <- lm(weight ~ Diet:Time, data = ChickWeight)
```

Graphical display of the models

To display the models, we first store the intercepts and slopes and vectors `a` and `b`. We put these vectors in a new dataframe called `df`. For the first model, we have a single line, for the second model we have four different lines. Hence, the data.frame will have 1 + 3*4 = 13 rows.

```
df <- data.frame(r = 1:13,a = NA,b = NA,Diet = as.character(c(0,rep(1:4,3))),
      model = c('ancW1',rep(c('ancW2','ancW3','ancW4'),each = 4)))
df$a[1] <- coef(ancW1)[1]  # common intercept
df$b[1] <- coef(ancW1)[2]  # common slope
df$a[2:5] <- coef(ancW2)[1:4]  # 4 intercepts in 4 groups
df$b[2:5] <- rep(coef(ancW2)[5],4)  # common slope in 4 groups
df$a[6:9] <- coef(ancW3)[1:4] + c(0,rep(coef(ancW3)[1],3))  # 4 intercepts in 4 groups
df$b[6:9] <- coef(ancW3)[5:8] + c(0,rep(coef(ancW3)[5],3))  # 4 slopes in 4 groups
df$a[10:13] <- rep(coef(ancW4)[1],4)  # common intercept in 4 groups
df$b[10:13] <- coef(ancW4)[2:5]       # 4 slopes in 4 groups

CWd <- data.frame(ChickWeight,ChickWeight,ChickWeight,ChickWeight,
    model = rep(c('ancW1','ancW2','ancW3','ancW4'), each = nrow(ChickWeight)))
```

```
## Warning in data.frame(ChickWeight, ChickWeight,
## ChickWeight, ChickWeight, : row names were found
## from a short variable and have been discarded
```

```
CWd$Diet <- as.character(CWd$Diet)


ggplot(data = CWd,aes(x = Time,y = weight,colour = Diet,shape = Diet)) +
        geom_point() + facet_wrap(~model) +
        geom_abline(data = df,size = 1, aes(intercept = a,
        slope = b,colour = Diet,shape = Diet))
```

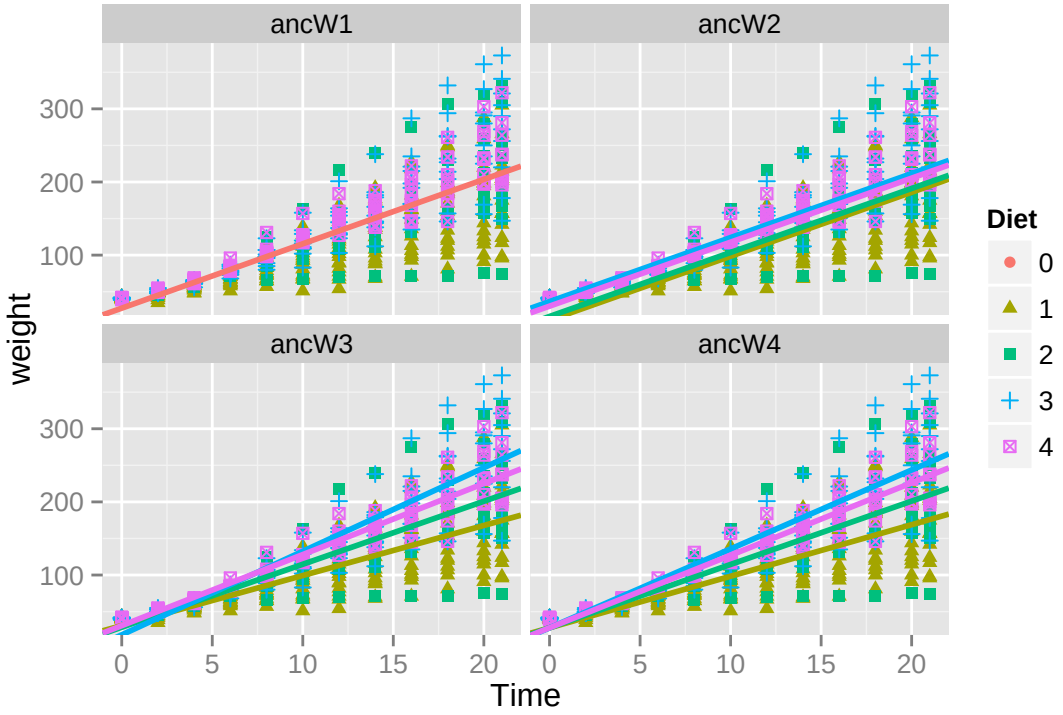


Figure 51:

Testing models

```
anova(ancW1,ancW2,ancW3,ancW4)
```

```
## Analysis of Variance Table
##
## Model 1: weight ~ Time
## Model 2: weight ~ Diet + Time
## Model 3: weight ~ Diet + Time + Diet:Time
## Model 4: weight ~ Diet:Time
##   Res.Df    RSS Df Sum of Sq     F  Pr(>F)
## 1    576 872212
## 2    573 742336  3    129876 37.30 < 2e-16 ***
## 3    570 661532  3     80804 23.21 3.5e-14 ***
```

```
## 4    573 665525 -3     -3993  1.15    0.33
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Model `ancW2` fits significantly better than model `ancW1`. Model `ancW3` fits significantly better than model `ancW2`. Model `ancW3` does not significantly fit better than model `ancW4`, though.

Finding the best model

We cannot formally test whether models `ancW2` and `ancW4` differ significantly, because they are not nested: neither is model `ancW2` a special case of `ancW4`, nor vice versa. A less formal comparison of the models can be made with the `anova` function: we see that the residual sum of squares of model `ancW4` is 76811 lower than that of model `ancW2`. So, model `ancW4` fits better.

```
anova(ancW2,ancW4)
```

```
## Analysis of Variance Table
##
## Model 1: weight ~ Diet + Time
## Model 2: weight ~ Diet:Time
##   Res.Df    RSS Df Sum of Sq F Pr(>F)
## 1    573 742336
## 2    573 665525  0     76811
```

Reproduce the F-statistic for comparing larger model 3 with smaller model 4

Test statistic F is a corrected difference in RSS:

$F = \frac{\frac{RSS_4 - RSS_3}{p_3 - p_4}}{\frac{RSS_3}{n - p_3}}$

$F \sim F_{p_4 - p_3, n - p_3}$

```
RSS4 <- deviance(ancW4)
RSS3 <- deviance(ancW3)
p4 <- ancW4$rank
p3 <- ancW3$rank
df3 <- ancW3$df.residual
(F <- ((RSS4 - RSS3)/(p3 - p4))/ (RSS3/df3))
```

```
## [1] 1.15
```

```
(p <- 1-pf(F,df1 = p3 - p4, df2 = df3))
```

```
## [1] 0.33
```

```
p<0.05
```

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

End

- Exercises ANCOVA
  - Solutions exercises ANCOVA
- Return to main page

Mixed models

The effect of chick

Until now, we ignored differences due to chicken; but these do exist and are substantial:

```
ggplot(data = ChickWeight,aes(x = Time,y = weight,colour = Chick,group = Chick)) +
       geom_line() + facet_grid(~Diet) + theme(legend.position = 'none')
```

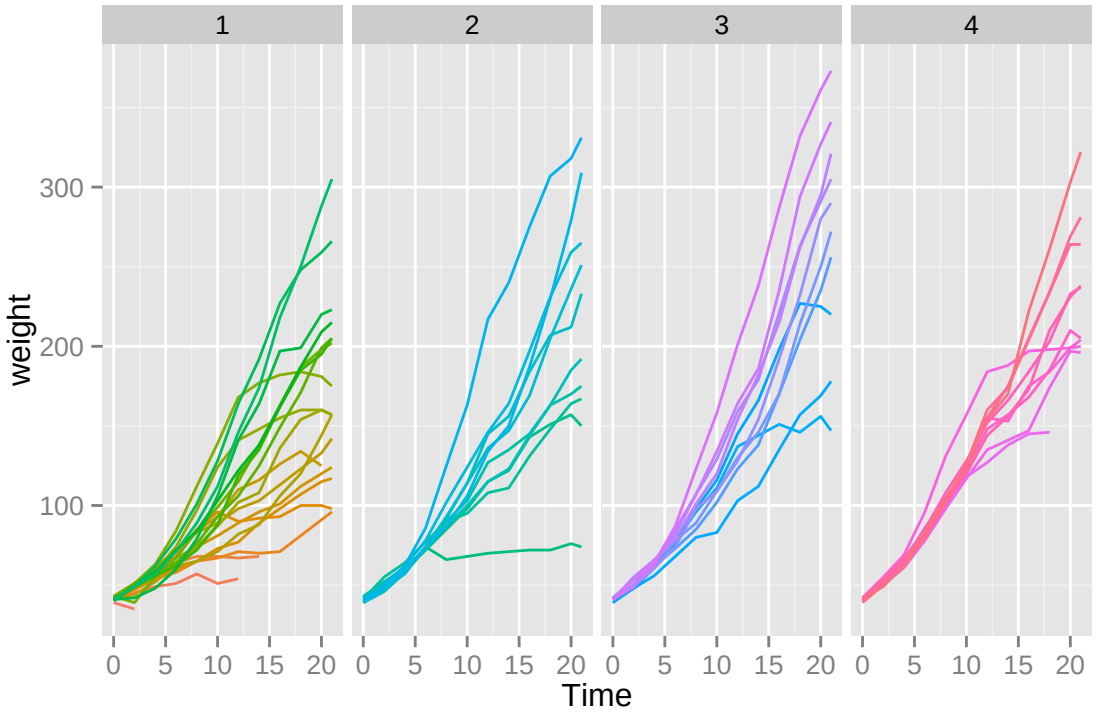


Figure 52: Plot of the `ChickWeight` data, showing the individual animals.

Including random effects in the model

Models with fixed effects and random effects other than error are called mixed models. We use the package `lme4` for fitting mixed models.

In matrix notation a mixed model is

$\mathbf{y} = \mathbf{Xb} + \mathbf{Zu} + \mathbf{e}$

with $\mathbf{u} \sim \mathbf{N}(0, \sigma_u^2)$ and $\mathbf{e} \sim \mathbf{N}(0, \sigma_e^2)$. In this model, vector $\mathbf{u}$ may be a vector of chickeffects.

Fitting the model in `lme4`

```
lme41 <- lmer(weight~Diet + Time + (1|Chick),ChickWeight)
lme41
```

```
## Linear mixed model fit by REML ['lmerMod']
## Formula: weight ~ Diet + Time + (1 | Chick)
##    Data: ChickWeight
## REML criterion at convergence: 5584
## Random effects:
##  Groups   Name        Std.Dev.
##  Chick    (Intercept) 22.9
##  Residual             28.3
## Number of obs: 578, groups:  Chick, 50
## Fixed Effects:
## (Intercept)        Diet2        Diet3
##       11.24        16.21        36.54
##       Diet4         Time
##       30.01         8.72
```

Checking differences

We can wonder if this model is an improvement over the ANCOVA model `ancW2`. We do this indirectly because the `anova` function can not handle an object of class `lmer` and one of class `lm`.

```
ancW2 <- lm(weight~Diet + Time, ChickWeight)
AIC(logLik(ancW2))
```

```
## [1] 5790
```

```
AIC(logLik(lme41))
```

```
## [1] 5598
```

We cautiously conclude (this is no a hypothesis test) that including a random intercept for `Chick` improves the fit of our model.

Including a random regression coefficient

We could also think that not only the intercept but also the slope is dependent on the chick:

```
lme42 <- lmer(weight~Diet + Time + (Time|Chick), ChickWeight)
lme42
```

```
## Linear mixed model fit by REML ['lmerMod']
## Formula: weight ~ Diet + Time + (Time | Chick)
##    Data: ChickWeight
## REML criterion at convergence: 4804
## Random effects:
```

```
##  Groups   Name        Std.Dev. Corr
##  Chick    (Intercept) 12.40
##           Time         3.76    -0.98
##  Residual             12.78
## Number of obs: 578, groups:  Chick, 50
## Fixed Effects:
## (Intercept)        Diet2        Diet3
##       26.36         2.84         2.00
##       Diet4         Time
##        9.25         8.44
```

Testing

Test whether model with random slopes and intercepts fits better than model with random intercepts only

```
anova(lme41, lme42)
```

```
## refitting model(s) with ML (instead of REML)

## Data: ChickWeight
## Models:
## lme41: weight ~ Diet + Time + (1 | Chick)
## lme42: weight ~ Diet + Time + (Time | Chick)
##       Df  AIC  BIC logLik deviance Chisq Chi Df
## lme41  7 5619 5650  -2803     5605
## lme42  9 4834 4873  -2408     4816   789      2
##       Pr(>Chisq)
## lme41
## lme42     <2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Return to main page

- Exercises mixed models
  - Solutions exercises mixed models
- Return to main page

Exercises R basics

Exercises : Introduction

- Before you start: You may want to type all your exercises in the R script window such that you can save your work.
  - Open a script window from the File menu
  - Type getwd() and then use `CTRL-Enter` to run the command
  - Close (and save) the script to a location that you can take home such as a usb drive
  - Reopen the script.
- Familiarize yourself with the help in R and the R website.
  - Look for help on the functions `data.frame`, `dim`, and `mean`.
  - Go to the R project website (note the manuals pages)
  - Go to the CRAN site (what is the most current version of R ?)
- Get a list of all the functions in R that contain the text "t test" or another word of your interest.
  - For information on how to do this, read the help for the function `help.search`.

Exercises : Basics-1

- Packages
  - Use the instruction in the course notes to install a new package `pedigree` or any other package that you might like (see CRAN for a list of packages)
- Vector
  - Create a `vector` aa which contains 8 numbers
  - Copy the first 3 elements from aa into another vector aaa
  - Check the length and class of the resulting object
  - Use aa to make another `vector` bb which contains squared values of aa
  - Calculate the mean of bb.

Exercises : Basics-2

- Matrix
  - Create a 10x10 `matrix` my.matrix which contains numbers 1 to 100.
  - Calculate the mean of the matrix
  - Calculate the mean of the first column of my.matrix
  - Calculate the mean of the third row of my.matrix

Return to main page

- Back to the presentation
- Solutions exercises R basics
- Return to main page

Exercises data manipulation in R

Exercises : Data Manipulation

- We will use an online dataset `http://rcursus.dairyconsult.nl/lime.prn`
- Read the dataset into an object called `lime`
- Use the function is.na() to check if the column `pH` contains any `NA` values ?
- Make a subset `lime.sub` that only contains `pH` values above 5.90.
- Compare the sizes of `lime` and `lime.sub`
- In `lime`, change the values in the column `type`:
  - `AL` should become `Agricultural`
  - `GL` should become `Granular`
- Make a subset `lime.sub2` from `lime` with only the columns `type` and `pH` and only rows where `rate` is smaller than 2. What are the number of rows and columns in `lime.sub2` ?

Exercises : Data Manipulation

- Order `lime` on the column `pH` and check the result.
- Save your the resulting object as a lime.csv document.
- Save your scripts as a .R file.
- (if time permits : calculate the mean `pH` for each level of `rate` in `lime` and in `lime.sub`).

Exercises : Basic Statistics

- create a vector `r1` with 10 random numbers from a normal distribution with mean 1 and s.d. 1
- create a vector `r0` with 10 random numbers from a normal distribution with mean 0 and s.d. 1
- Perform a t-test for difference of means of `r0` and `r1` (equal variances)
- Write down the p-value from this test
- (IF time permits : In the `lime` dataset, test wether treatment with Agricultural Lime has a different effect from treatment with Granular Lime).

Return to main page

- Back to the presentation
- Solutions exercises data manipulation
- Return to main page

Exercises programming in R

Programming 1

Objects of class `numeric`

1. Declare a vector `n1` with 20 random numbers (you could use function `sample` for this).
2. Copy elements at positions 1 to 10 to new vector `n11`. Copy elements at positions 11 to 20 to vector `n12`.
3. Combine `n11` and `n12` into `nnew`.
4. **Challenge**: copy element at the even positions from `n1` to `neven` (use function `seq` and read about the `by` argument of this function).

Objects of class `character`:

1. Declare `fName` with your first name and `lName` with your last name.
2. Combine `fName` and `lName` into vector `Name`.
3. Paste `Name` into `pName` with a space between your first name and your last name.

Matching and replacing:

1. Test `(1,2,3)` occur in vector `n1`. If not, make them occur in `n1`.
2. Obtain the indices where `(1,2,3)` occur in `n1`.
3. Replace the elements of `n1` where `(1,2,3)` occur with `NA` (`NA` is used in R for `Not Available`).

Splitting a character string:

1. Split character string `Name` back into `fName1` and `lName1`.
2. Test if `fName` and `fName1` are identical (hint: use the `identical` function).
3. Split `fName1` into single characters and count the number of characters.

End of exercises programming 1

- Return to presentation programming
- Go to solutions programming 1
- Return to main page

Exercises programming 2

Matrices

1. Create a square matrix object `mat1` of three rows. The elements of `mat1` should be `NA`.
2. Fill the diagonal elements of `mat1` with 1 and the off-diagonal elements with increasing numbers (use function `diag` to obtain and replace the diagonals, and functions `upper.tri` and `lower.tri` to obtain and replace the off-diagonals of the matrix.
3. Replace the elements of row 2 in `mat1` with 2 times their value.
4. Calculate the column and row sums of `mat1`.
5. Make `mat2` as the combination of two `mat1` by row and `mat3` as the combination by columns.

Data.frames

1. Read about the `ChickWeight` dataset (`?ChickWeight`). Summarize this `data.frame`.
2. Copy the subset of `ChickWeight` where `Time == 0` to `CW`.
3. Calculate the mean weight of the chicks at `Time == 0`.
4. Calculate the mean weight of the chicks per diet (first know how many diets there are).

5. Replace the chick ids with a unique name for each chick. If it does not work, have a look at what the `Chick` column in reality is.

Writing and reading data

1. Write the `ChickWeight` `data.frame` into a *.csv file.
2. Read the data into `ChickWeight1` and compare if the two data.frames are identical.

End of exercises programming 2

- Return to presentation programming
- Go to solutions programming 2
- Return to main page

Exercises programming 3

Iteration

1. Calculate the standard deviation of weight at each moment and for each diet in the `ChickWeight` data (use function `sd` to calcute the standard deviation).
2. Count the number of chicks at each timepoint in the `ChickWeight` data.
3. Use function `tapply` to repeat the questions in the exercise above. If you did already, repeat the previous exercises using a loop.

End of exercises programming 3

- Return to presentation programming
- Go to solutions programming 3
- Return to main page

Exercises graphics in R

Basic graphics

Plotting:

1. Load the nlme package and the make dataset `Soybean` available (code `require(nlme); data(Soybean)`). Explore this `Soybean` dataset.
2. Plot the relationship between `weight` and `Time`.
3. Explore the relationships between `weight` and `Variety`; `weight` and `year`.
4. In one plot, display the relationship between `weight` and `Time` for each `Variety`. Add a legend to this plot.
5. Save the plot as a pdf-file. Control if the plot has been saved on the approprate place.

Customizing a plot:

1. Plot `weight` against `Time` of the `Soybean` dataset.
2. Add a linear regression line for each `Variety`, use a different color for each `Variety`.
3. Add a legend.
4. Save the plot as a jpeg-file and check the file.

End exercises basic graphics

- Return to presentation graphics
- Go to solutions basic graphics
- Return to main page

Plotting with ggplot2

1. Make the dataset `Soybean` from package `nlme` available (code `require(nlme); data(Soybean)`).
2. Plot `weight` against `Time`, with a different color and character per variety.
3. Let `year` define the panels of the plot.
4. Add trendlines to the plot

End exercises ggplot2 graphics

- Return to presentation graphics
- Go to solutions ggplot2 graphics
- Return to main page

Exercises regression in R

Exercises Linear regression

1. Load the `BodyWeight` data from the `nlme` package.
2. Make a plot of weight against Time.
3. Fit a linear regression model to examine the relationship of time with Weight.
4. Extract the regression coefficients and the anova table.
5. Calculate the residual variance of the fitted model.
6. Make some diagnostic plots using standardized residuals.
7. Make a QQ-plot of the standardized residuals.
8. Use simulation to get some feeling for the size of deviations to be expected in QQ-plots, if data are normally distributed: draw repeatedly 25 observations from a standard normal distribution and make QQ-plots.

End exercises linear regression

- Return to presentation regression
- Solutions exercises regression
- Return to main page

Exercises ANOVA

1. Make a new dataset which contains the final weigth of the rats:

```
BodyWeight <- BodyWeight[with(BodyWeight,order(Rat,-Time)),]
BWf <- BodyWeight[!duplicated(BodyWeight$Rat),]
```

2. Plot the final weight for each diet.
3. Test the effect of diet on final weight.
4. Reparameterize the model and test if diet 1, 2, and/or 3 differ significantly from diet 4.

End exercises ANOVA

- Return to presentation ANOVA
- Solutions exercises anova
- Return to main page

Exercises ANCOVA

1. Make a plot of weight against time, where you can differentiate between the diets.
2. Fit an ancova model that includes a linear effect for time and an effect of diet.
3. Test if time and diet have a significant relationship with weight.
4. Optional: make a new data.frame with initial and final weight of the rats. Test if initial weight has a relationship with final weight.

End exercises ANCOVA

- Return to presentation ANCOVA
- Solutions exercises ancova
- Return to main page

Exercises Mixed models

1. Make a plot of weight against time, identify the effect of individuals rats.
2. Fit a model with rat as a random effect. Test if the improvement of fit is significant compared to a model without random rat effect.
3. Fit a model with a random slope for rats. Test the improvement of fit of this model.

End exercises Mixed models

- Return to presentation mixed models
- Solutions exercises mixed models
- Return to main page

Colophon

This document was written in Rmd format, and converted to markdown using the `knitr` package of R (see http://yihui.name/knitr/). The markdown document was converted to html slides and a pdf document with the `render` function of the `rmarkdown` package.