Review of the course "R for Data Science" Part 01(Talk 01~04)

By Haoran Nie @ HUST Life ST

Multi-omics data analysis and visualisation, #1

Talk 01

This section has nothing to explain :)

R language basics, part 1

Talk 02

Fundamental Data Type

The most basic data types include **numbers**, **logical symbols** and **strings** and are the basic building blocks of the other data types.

Simple Data Types

This includes vectors and matrices, both of which can contain multiple values of a certain basic data type, such as a **matrix** consisting of multiple numbers, a **vector** consisting of multiple strings, and so on. However, **they can only contain a single data type.**

```
c(100, 20, 30) ## Interger vector
c("String", "Array", "It's me".) ## String vector
c(TRUE, FALSE, TRUE, T, F) ## A logic vector
```

As shown above, arrays are usually defined with the function `c()`. In addition, a `vector` containing consecutive integers can be defined using the `:` operator.

Conversion between data types

1. Automatic Conversion

   A `vector` can contain only one basic data type. Therefore, when defining arrays, if the input values are mixed, certain basic data types are automatically converted to other types to ensure consistency of the numeric types; this is called `coerce` in English, and has the meaning of forced conversion. The priority of this conversion is:

   - Logical types -> numeric types
   - Logical Type -> String
   - numeric type -> string

2. Manual switchover

   In addition to the automatic conversion, you can manually convert the types of the elements in a vector:

   - Checking the type of a variable `class()`
   - Checking of classes `is.type()`
   - Conversion of classes `as.type()`

Some special values in matrices

- `NA` (Not Available) missing values
- `NaN` (Not a Number)is meaningless
- `-Inf` Negative Infinity
- `Inf` Positive Infinity
- `NULL` Null

Some functions to determine these special values:

- `is.na()`
- `is.finite()`
- `is.infinite()`

Vectors and Arrays

Both are arrays. A `vector` is a one-dimensional array and a matrix is a two-dimensional array.

This means.

- There can be more dimensional arrays
- High-dimensional arrays, like `vector` and matrices, can contain only one basic data type.
- Higher dimensional arrays can be defined by the `array()` function.

Vector maniulation

```
dim(m);
nrow(m);
ncol(m);
range(m); ## Available when the content is numeric
summary(m); ## Can also be used in vector
```

Extra:

- Incorporation `ab = c(a, b)`
- Take part `ab[1]`
- Replacement of individual values `ab[1] = c`
- Replacing multiple values `ab[c(2, 3)] = c("Weihua", "Chen")`
- Naming elements and replace values `names(ab) = as.character(ab)`
- Reverse `rev(1:10)`

- Sort&order

```
lts = sample(LETTERS[1:20])
sort(lts)
```

- Fetch one line or multiple lines

```
# (There's already some data in workspace)

$ m
> (List the content of matrix "m")

$ m[1, ]
> (List the first row of matrix 'm')

$ m[1:2, ]
> (List the first two rows of matrix 'm')
```

  You can also let the console to fetch multiple lines as the order you give.

  `m[c("row_B", "row_A")]`

  The console will output the contents of matrix "m" in the order of "row_B" and then "row_A".

- Fetch one column or multiple columns

  As can be seen from the same principle, I only list codes here

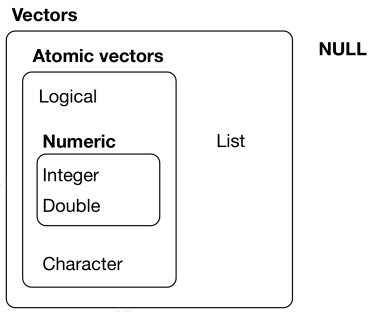
```
m[ , 1]
m[ , c(1:2)]
m[ , c("col_B", "col_A")]
```

- Fetch parts `m[1:2, 2:3]`
- Replacement

```
m[1, ] = c(10)
m[, "C"] = c(230, 140)
m[1:2,] = matrix( 1:6, nrow=2)
m[1, c("C", "B")] = matrix(110:111, nrow = 1)
```

- Transparent `t(m)`

The hierarchy of R's vector types



You can use function `typeof()` to know the type of a vector.

Here are some examples of other `is.xxx()` function:

```
is.null( NULL )
is.numeric( NA )
is.numeric( Inf );
is.list(); # This is a function which can take the place of "typeof()"
is.logical()
is.character()
is.vector();
# more ...
```

R language basics, part 2

Talk 03

`data.frame`

What is a `data.frame`?

```
library(tidyverse);
library(kableExtra)
kbl(head(mpg),
    booktabs = T)
```

Here's the result:

manufacturer	model	displ	year	cyl	trans	drv	cty	hwy	fl	class
audi	a4	1.8	1999	4	auto(l5)	f	18	29	p	compact
audi	a4	1.8	1999	4	manual(m5)	f	21	29	p	compact
audi	a4	2.0	2008	4	manual(m6)	f	20	31	p	compact
audi	a4	2.0	2008	4	auto(av)	f	21	30	p	compact
audi	a4	2.8	1999	6	auto(l5)	f	16	26	p	compact
audi	a4	2.8	1999	6	manual(m5)	f	18	26	p	compact

Usage of `head()` and `tail()`

- `head()` is a function to display the first rows of some data (vectors etc.)
- `tail()` is a function to display the last rows of some data (vectors etc.)

Components of `data.frame` and common functions

Components:

- Two-dimensional table
- consists of different columns; each column is a vector, different columns can have different data types, but a column contains only one data type (`int`, `num`, `chr` ...)
- Each column has the same length

Common functions:

```
nrow() # Show the number of rows
ncol() # Show the number of columns
dim()  # Show the dimension
```

Structure of `data.frame` & `tibble`

`str(mpg)`

This command shows the structure of the tibble `mpg`:

```
## tibble [234 x 11] (S3: tbl_df/tbl/data.frame)
##  $ manufacturer: chr [1:234] "audi" "audi" "audi" "audi" ...
##  $ model       : chr [1:234] "a4" "a4" "a4" "a4" ...
##  $ displ       : num [1:234] 1.8 1.8 2 2 2.8 2.8 3.1 1.8 1.8 2 ...
##  $ year        : int [1:234] 1999 1999 2008 2008 1999 1999 2008 1999 1999 2008 ...
##  $ cyl         : int [1:234] 4 4 4 4 6 6 6 4 4 4 ...
##  $ trans       : chr [1:234] "auto(l5)" "manual(m5)" "manual(m6)" "auto(av)" ...
##  $ drv         : chr [1:234] "f" "f" "f" "f" ...
##  $ cty         : int [1:234] 18 21 20 21 16 18 18 18 16 20 ...
##  $ hwy         : int [1:234] 29 29 31 30 26 26 27 26 25 28 ...
##  $ fl          : chr [1:234] "p" "p" "p" "p" ...
##  $ class       : chr [1:234] "compact" "compact" "compact" "compact" ...
```

Make a new `data.frame`

You can use the function `data.frame()` to make a new `data.frame`

```
data2 =
        data.frame(
          data = sample(1:100, 10),
    group = sample(LETTERS[1:3], 10, replace = TRUE)
    data2 = 0.1
  )
```

How to add row(s)/col(s) to an existing data.frame

Create the "table header" first, then populate the `data.frame`

```
df2 =
        data.frame(
    x = character(),
    y = integer(),
    z = double() ,
    stringsAsFactors = FALSE
  )

df2 =
        rbind(
    df2,
    data.frame(
      x = "a",
      y = 1L,
      z = 2.2
    )
  )

df2 =
```



```
        rbind(
    df2,
    data.frame(
      x = "b",
      y = 2,
      z = 4.4
    )
  )
```

ATTENTION

- Use `rbind()` function to add rows, use `cbind()` function to add columns.
- Define the new line using `data.frame()` function, the "header" needs to be the same as the merged table.

You can also use these functions to bind several data.frames.

`tibble`

`tibble` is kind of similar to `data.frame`.

Make new `tibble`

`tibble` related functionality is provided by the `tibble` or `tidiverse` packages.

Almost all of the functions that you'll use in this book produce tibbles, as tibbles are one of the unifying features of the tidyverse. Most other R packages use regular data frames, so you might want to coerce a data frame to a tibble. You can do that with `as_tibble()`:

```
as_tibble(iris)
#> # A tibble: 150 × 5
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#>          <dbl>       <dbl>        <dbl>       <dbl> <fct>
#> 1          5.1         3.5          1.4         0.2 setosa
#> 2          4.9         3            1.4         0.2 setosa
#> 3          4.7         3.2          1.3         0.2 setosa
#> 4          4.6         3.1          1.5         0.2 setosa
#> 5          5           3.6          1.4         0.2 setosa
#> 6          5.4         3.9          1.7         0.4 setosa
#> #  144 more rows
```

Another way to create a tibble is with `tribble()`, short for **tr**ansposed tibble. `tribble()` is customised for data entry in code: column headings are defined by formulas (i.e. they start with `~`), and entries are separated by commas. This makes it possible to lay out small amounts of data in easy to read form.

```
tribble(
  ~x, ~y, ~z,
  #--/--/----
  "a", 2, 3.6,
  "b", 1, 8.5
)
#> # A tibble: 2 × 3
#>   x         y     z
```

```
#>   <chr> <dbl> <dbl>
#> 1 a         2   3.6
#> 2 b         1   8.5
```

- add_row()
- add_column()

Manipulate the tibble

See “Manipulate the data.frame”

tibble to data.frame

- as.data.frame()
- as_tibble()

e.g.

```
library(tibble)
as.data.frame(head(as_tibble(iris)))
```

Differences between tibble and data.frame

Tibble evaluates columns sequentially

```
rm(x,y) # Delete possible x, y
tibble(x = 1:5, y = x^2); # You can do this with tibble
data.frame(x = 1:5, y = x ^ 2); # But data.frame doesn't work.
```

data.frame causes trouble when fetching subset operations

```
df1 =
        data.frame(x = 1:3, y = 3:1)
class(df1[, 1:2])

#> [1] "data.frame"

# Subset operation :takes a column and expects a data.frame ()
class(df1[, 1]) # The result is a vector ...

#> [1] "integer"

## Tibble doesn't.
df2 =
        tibble(x = 1:3, y = 3:1)
class(df2[, 1]) ## Tibble forever

#> [1] "tbl_df" "tbl" "data.frame"
```

tibble allows controlled data type conversion

There’s no proper example here.
:_(

Recycling

```
data.frame(a = 1:6, b = LETTERS[1:2]) # data.frame CAN!!!
```

OUTPUT

```
#   a b
# 1 1 A
# 2 2 B
# 3 3 A
# 4 4 B
# 5 5 A
# 6 6 B

tibble(a = 1:6, b = LETTERS[1:2]); ## But tibble CAN'T!!!
```

OUTPUT

```
# Error:
# ! Tibble columns must have compatible sizes. ## * Size 6: Existing data.
# * Size 2: Column `b`.
#   Only values of size one are recycled.
```

ATTENTION!

The recycling of tibble is limited to lengths of 1 or equal; data.frame is just divisible.

data.frame will do partial matching, while tibble will NEVER do it.

```
df = data.frame(abc = 1)
df$ab; # Unwanted result ...

df2 = tibble(abc = 1)
df2$a; # Produce a warning and return NULL
```

OUTPUT

```
# Warning: Unknown or uninitialised column: `a`.
# NULL
```

Advanced tips for using data.frame and tibble

- attach()
- detach()
- with()
- within()

Following is the introduction (Produced by ChatGPT)

These functions—attach(), detach(), with(), and within()—are incredibly useful when working with data frames or tibbles in R, aiding in smoother workflows and code readability. Here's a breakdown of their functionality:

`attach()` and `detach()`

- **Purpose**: These functions allow you to temporarily attach a data frame to the search path, making its columns directly accessible by their names.
- **Usage**:
  - `attach(df)` attaches the specified data frame `df`.
  - `detach(df)` detaches the specified data frame `df`.
- **Example**:

```
data(mtcars) # Loading a sample dataset
attach(mtcars) # Attaching mtcars

# Now, columns can be accessed directly
summary(mpg)
mean(mpg)

detach(mtcars) # Detaching mtcars
```

- **Note**: While convenient, using `attach()` can sometimes lead to confusion or unintended consequences, such as masking variables in your environment. It's often recommended to avoid using `attach()` due to potential side effects.

`with()`

- **Purpose**: `with()` allows you to execute expressions in an environment where the data frame's columns can be referenced without using `$`.
- **Usage**:
  - `with(data, expr)` evaluates `expr` in the context of the specified data frame `data`.
- **Example**:

```
data(mtcars) # Loading a sample dataset

with(mtcars, {
  mean(mpg)
  summary(cyl)
  })
```

- **Advantage**: It helps avoid repetitive use of the data frame name while working with its columns.

`within()`

- **Purpose**: Similar to `with()`, `within()` allows modification of a data frame by evaluating expressions within it.
- **Usage**:
  - `within(data, expr)` modifies `data` according to `expr` and returns the modified data frame.
- **Example**:


```
data(mtcars) # Loading a sample dataset

modified_mtcars <- within(mtcars, {
  mpg_square <- mpg^2
  hp_doubled <- hp * 2
  })
head(modified_mtcars) # Checking the modified data frame
```

- **Advantage**: `within()` is useful when you want to create or modify columns within the data frame without having to repeatedly refer to the data frame name.

Remember, while these functions can streamline your code, it's crucial to use them judiciously to avoid unexpected behavior or cluttering your global environment.

Here's the console print output for the examples provided earlier:

```
# Using attach() and detach()
data(mtcars) # Loading a sample dataset
attach(mtcars) # Attaching mtcars

# Now, columns can be accessed directly
summary(mpg)
# Output:
#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#  10.40   15.43   19.20   20.09   22.80   33.90

mean(mpg)
# Output:
# [1] 20.09062

detach(mtcars) # Detaching mtcars

# Using with()
data(mtcars) # Loading a sample dataset

with(mtcars, {
  mean(mpg)
  # Output:
  # [1] 20.09062

  summary(cyl)
  # Output:
  #   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  #   4.00   4.00    6.00    6.188   8.00    8.00
})

# Using within()
data(mtcars) # Loading a sample dataset

modified_mtcars <- within(mtcars, {
  mpg_square <- mpg^2
  hp_doubled <- hp * 2
})
head(modified_mtcars) # Checking the modified data frame
# Output:
#                   mpg cyl disp  hp drat    wt  qsec vs am gear carb mpg_square hp_doubled
# Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4     441.00       220
# Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4     441.00       220
```

```
# Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1     519.84       186
# Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1     457.96       220
# Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2     349.69       350
# Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1     327.61       210
```

File IO

Read from files

Using functions from the `readr` package

```
# readr is part of tidyverse
library(tidyverse) # or alternatively
library(readr)
```

Some available functions:

- `read_csv()`: comma separated (CSV) files
- `read_tsv()`: tab separated files
- `read_delim()`: general delimited files
- `read_fwf()`: fixed width files
- `read_table()`: tabular files where columns are separated by white-space. read_log(): web log files

  Full documentation of the package is available through https://www.rdocumentation.org/packages/readr/versions/1.3.1.

Usage

- Read with predifined column types

```
myiris2 =
        read_csv("../data/talk03/iris.csv",
           col_types =
                   cols(
              Sepal.Length = col_double(),
              Sepal.Width = col_double(),
              Petal.Length = col_double(),
              Petal.Width = col_double(),
              Species = col_character()
            )
          )
```

- To read from other formats, you can try the following packages:

  Similar to Python

  - `haven` - SPSS, Stata, and SAS files
  - `readxl` - excel files (.xls and .xlsx) DBI - databases
  - `jsonlite` - json
  - `xml2` - XML
  - `httr` - Web APIs
  - `rvest` - HTML (Web Scraping)

Write to files

Use the following functions to write object(s) to external files:

Default parameters are listed.

More related documents can be found in https://r4ds.had.co.nz/data-import.html?q=file#writing-to-a-file.

- Comma delimited file:

```
write_csv(
  x,
  path,
  na = "NA",
  append = FALSE,
  col_names = !append
)
```

- File with arbitrary delimiter:

```
write_delim(
  x,
  path,
  delim = " ",
  na ="NA",
  append = FALSE,
  col_names = !append
)
```

- CSV for excel:

```
write_excel_csv(
  x,
  path,
  na = "NA",
  append = FALSE,
  col_names = !append
)
```

- String to file:

```
write_file(
  x,
  path,
  append = FALSE
)
```

- String vector to file, one element per line:

```
write_lines(
  x,
  path,
  na = "NA",
  append = FALSE
)
```

- Object to RDS file:

```
write_rds(
  x,
  path,
  compress =
          c(
      "none",
      "gz",
      "bz2",
      "xz"
    ),
  ...
)
```

- Tab delimited files:

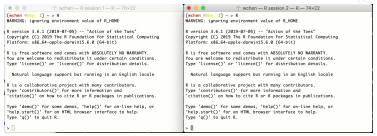
```
write_tsv(
  x,
  path,
  na = "NA",
  append = FALSE,
  col_names = !append
)
```

R language basics, part 3: factor

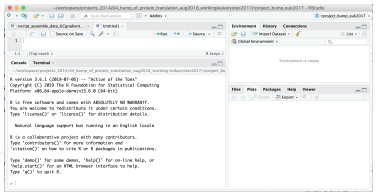
Talk 04

IO and working enviroment management

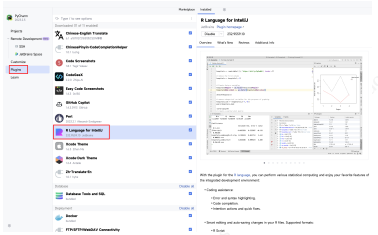
Each R session is a separate **work space** containing its own data, variables, and operation history.



Each RStudio session is automatically associated with a R session



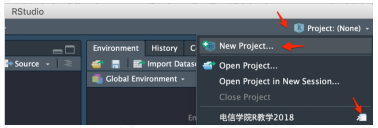
If you want to coding with R using PyCharm or other JetBrains IDE (i.e. IntelliJ, CLion, etc.), remember to install the *R Language Plug-in*.



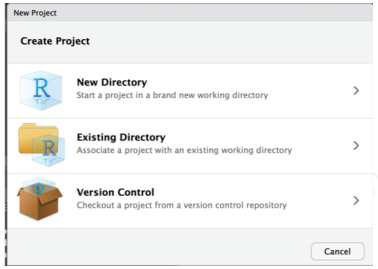
Start a new RStudio session by creating a new project

To start a new session in PyCharm, simply press the bottom corner and select a new session.

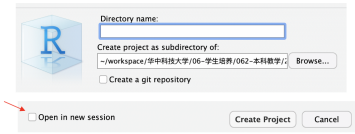
- Click the Project button in the upper right corner and select New Project in the pop-up menu ...



- Select: New directory -> New Project in the popup window



- Enter a new directory name, choose its mother directory ...



Working Space

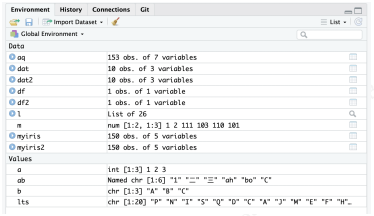
Current workspace, including all loaded data, packets and homebrew functions.

Variables can be managed with the following code:

```
ls() # Show all the variables in current workspace/session
rm(x) # Remove a variable
rm(list = ls()) # Remove ALL variables in current workspace/session
```

Variables in working space in RStudio

The "Environment" window in the upper right corner of RStudio shows all the variables of the current workspace.



Save and restore work space

```
# Save all loaded variables into an external .RData file
save.image(file = "prj_r_for_bioinformatics_aug3_2019.RData")
# Restore (load) saved work space
load(file = "prj_r_for_bioinformatics_aug3_2019.RData")
```

Notes:

- Existing variables will be kept, however, those will the same names will be replaced by loaded variables
- Please consider using `rm(list=ls())` to remove all existing variables to have a clean start
- You may need to reload all the packages

Save selected variables

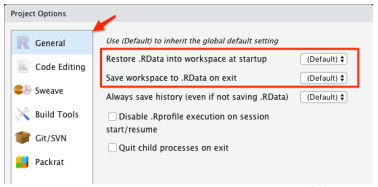
Sometimes you need to transfer processed data to a collaborator ...

```
# Save selected variables to external
save(
  city,
  country,
  file="1.RData"
)
# You can specify directory name
load("1.RData")
```

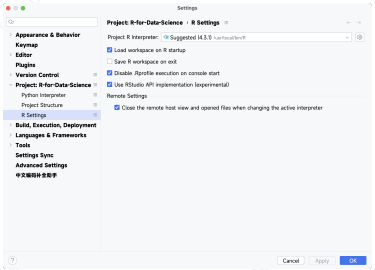

Close and (re)open a project

- To close a project

- In RStudio and similar IDEs, there are some preferences to choose



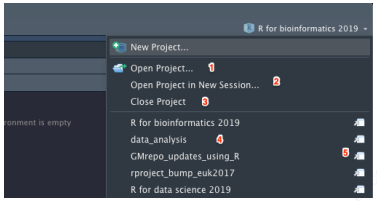
The UI in PyCharm



Notes:

- Save on exit
- Load on opening
- When the data is large, the loading time may be too long ...

Open a project



When in PyCharm, simply drag the working directory to its main window, remember to trust the project.

Factors

Factor is a data structure used for fields that takes only predefined, finite number of values (categorical data).

It will limit the selection of input data.

Play around with `levels()`

Here are instructions of modifying factor levels

> Based on the textbook

The levels are terse and inconsistent. Let's tweak them to be longer and use a parallel construction. Like most rename and recoding functions in the tidyverse, the new values go on the left and the old values go on the right:

```
load(gss_cat)

mutate(
  partyid = fct_recode(partyid,
    "Republican, strong"    = "Strong republican",
    "Republican, weak"      = "Not str republican",
    "Independent, near rep" = "Ind,near rep",
    "Independent, near dem" = "Ind,near dem",
    "Democrat, weak"        = "Not str democrat",
    "Democrat, strong"      = "Strong democrat"
  )
)

count(partyid)

#> # A tibble: 10 × 2
#>   partyid                n
#>   <fct>                <int>
#> 1 No answer              154
#> 2 Don't know               1
#> 3 Other party            393
#> 4 Republican, strong    2314
#> 5 Republican, weak      3032
#> 6 Independent, near rep 1791
#> #   4 more rows
```

Use this technique with care: if you group together categories that are truly different you will end up with misleading results.

The order of the `levels` determines the sorting order.

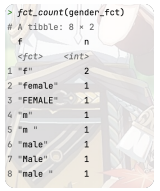
Use factor to clean data

> Usage of `fct_xxx()` functions.

Suppose I have a set of gender data that is written in a very irregular way:

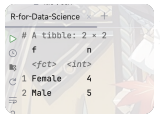

```
gender =
        c("f", "m ", "male ","male", "female", "FEMALE", "Male", "f", "m")

gender_fct =
  as.factor(gender)

fct_count(gender_fct)
```

The output looks like this:



Now I request to replace with Female, Male.

```
gender_fct =
  fct_collapse(
    gender,
    Female = c("f", "female", "FEMALE"),
    Male = c("m ", "m", "male ", "male", "Male")
  )

fct_count(gender_fct)
```



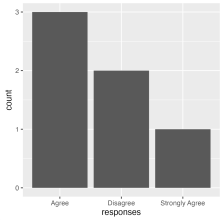
You can also use `fct_relabel()` to do the same thing

```
fct_relabel(
  gender,
  ~ ifelse(
    tolower(
      substring(., 1, 1)) == "f",
    "Female",
    "Male"
  )
)
```

Usage of factors in drawing plots

```
library(ggplot2)

responses =
```

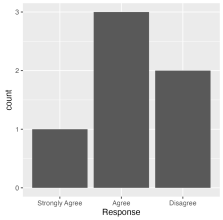
```
  factor(
    c("Agree", "Agree", "Strongly Agree", "Disagree", "Disagree", "Agree")
  )

response_barplot =
  ggplot(
    data = data.frame(responses),
    aes(x = responses)
  ) +
  geom_bar()
```



By default, `factor` is sorted alphabetically.

`ggplot2` also plots `factor` in that order, so you can adjust the `factor` to adjust the drawing order.

```
res =
  data.frame(responses)
# Sort by level of agreement from strong -> weak
res$res =
  factor(
    res$res,
    levels =
       c("Strongly Agree", "Agree", "Disagree")
  )

response_barplot2 =
  ggplot(
    data = res,
    aes(x = res)
  ) +
  geom_bar() +
  xlab("Response")
```



You can also use the parameter `ordered` to let others know that your `factor` is ordered properly.

```
responses =
  factor(
    c("Agree", "Agree", "Strongly Agree", "Disagree", "Disagree", "Agree"),
    ordered = TRUE
  )
```

```
> is.ordered(responses)
[1] TRUE
```

Using `factor` to vhange values

You can use `recode()` in `dplyr` package to change `value`

`dplyr` is a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges:

- `mutate()` adds new variables that are functions of existing variables
- `select()` picks variables based on their names.
- `filter()` picks cases based on their values.
- `summarise()` reduces multiple values down to a single summary.
- `arrange()` changes the ordering of the rows.

These all combine naturally with `group_by()` which allows you to perform any operation “by group”. You can learn more about them in `vignette("dplyr")`. As well as these single-table verbs, dplyr also provides a variety of two-table verbs, which you can learn about in `vignette("two-table")`.

> Based on the introduction on the official website of `dplyr`.

Here’s an example:

```
x =
  factor(
    c("alpha", "beta", "gamma", "theta", "beta", "alpha")
  )

x =
  recode(
    x,
    alpha = "a",
    beta = "b",
    gamma = "c",
    theta = "d"
  )
```

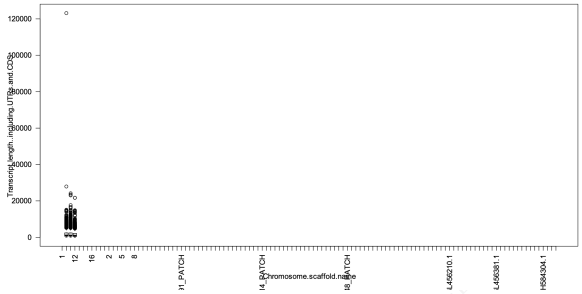
```
+     theta = "d"
+   )
> str(x)
 Factor w/ 4 levels "a","b","c","d": 1 2 3 4 2 1
>
```

Delete useless `levels`

```
mouse.genes =
  read.delim(
    file = "data/talk04/mouse_genes_biomart_sep2018.txt",
    sep = "\t",
    header = T,
    stringsAsFactors = T
  )
```

```
+   )
> str(mouse.genes)
'data.frame':   138532 obs. of  6 variables:
 $ Gene.stable.ID                      : Factor w/ 55029 levels "ENSMUSG00000000001",..: 17960 17959 17958 17957 17956 17955 17954 17953 17952 17951 ...
 $ Transcript.stable.ID                : Factor w/ 138532 levels "ENSMUST00000000001",..: 17312 17311 17310 17309 17308 17307 17306 17305 17304 17303 ...
 $ Protein.stable.ID                   : Factor w/ 65897 levels "","ENSMUSP00000000001",..: 1 1 16260 1 16259 16258 1 1 1 16257 ...
 $ Transcript.length..including.UTRs.and.CDS.: int  67 67 1144 69 519 1024 71 59 67 1378 ...
 $ Transcript.type                     : Factor w/ 48 levels "3prime_overlapping_ncRNA",..: 38 38 24 38 24 24 24 38 38 38 24 ...
 $ Chromosome.scaffold.name            : Factor w/ 117 levels "1","10","11",..: 115 115 115 115 115 115 115 115 115 115 115 ...
```

If you draw a plot without deleting the useless `levels`, you will get this result:



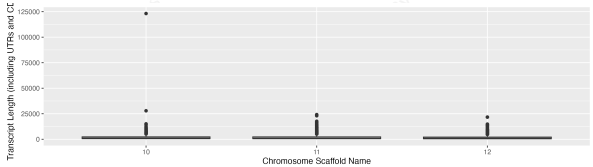
But when you delete the useless `level` using these commands:

```
mouse.chr_10_12$Chromosome.scaffold.name =
  droplevels(mouse.chr_10_12$Chromosome.scaffold.name)
```

You will see that:

```
+   droplevels( mouse.chr_10_12$Chromosome.scaffold.name )
> levels(mouse.chr_10_12$Chromosome.scaffold.name)
[1] "10" "11" "12"
```

Then, you'll get the plot like this:

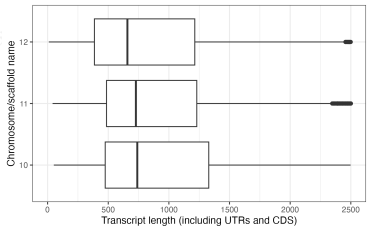


Source code:

```
mouse_gene_plot02 =
  ggplot(
    mouse.chr_10_12,
    aes(
      x = Chromosome.scaffold.name,
      y = Transcript.length..including.UTRs.and.CDS.
    )
  ) +
  geom_boxplot() +
  labs(
    x = "Chromosome Scaffold Name",
    y = "Transcript Length (including UTRs and CDS)"
  )
```

You can also use `tibble` to solve these problems:

```
mouse.tibble =
  read_delim(
    file = "data/talk04/mouse_genes_biomart_sep2018.txt",
    delim = "\t",
    quote = "",
    show_col_types = FALSE
  )

mouse.tibble.chr10_12 =
  mouse.tibble %>% filter(
    `Chromosome/scaffold name` %in% c("10", "11", "12"))


mouse_gene_plot03 =
    ggplot(
        mouse.tibble.chr10_12,
        aes(
        x = Chromosome.scaffold.name,
        y = Transcript.length..including.UTRs.and.CDS.
        )
    ) +
    geom_boxplot() +
    labs(
        x = "Chromosome",
        y = "Transcript length (bp)"
    ) +
    coord_flip() +
    ylim(0, 2500) +
    theme_bw()
```



Advance usage

- Use `reorder()` function to reorder the level.

```
x = reorder(
  `Chromosome/scaffold name`,
  `Transcript length (including UTRs and CDS)`,
  median
)
```

- Use `forcats::fct_reorder()` to reorder factors

```
x = fct_reorder(
  `Chromosome/scaffold name`,
  `Transcript length (including UTRs and CDS)`,
  median
)
```


Review of the course "R for Data Science" Part 02(Talk 05~08)

By Haoran Nie @ HUST Life ST

To reduce the size, all the codes listed will ***NOT*** include the output as picture.

R for bioinformatics, data wrangler, part 1

> Talk 05

TOC

pipe

dplyr

tidyr, part 1

Pipe in R

What is pipe in R?

- Pipe is `%>%`.
- It comes from the `magrittr` package by **Stefan Milton Bache**.
- Packages in the `tidyverse` load `%\>%` for you automatically, so you don't usually load `magrittr` explicitly.
- The essence is the passing of intermediate values.

Example

```
library(tidyverse)
library(magrittr)
a =
 subset(swiss, Fertility > 20)
cor.test(a$Fertility, a$Education)
```

The code above can be replaced by:

```
swiss %>%
  subset(., Fertility > 20) %$%
  cor.test(Education, Fertility)
```

You should remember that almost **all** the funtions support pipe.

Other kinds of pipe

- %T>%: Return left-side values
- %$>%: Attach ...
- %<>%

ATTENTION

1. The use of pipe makes the idea clearer;
2. Therefore, try to use %>% (which has a clear direction) instead of the other pipe, which has an unclear direction.

Data Wrangler - dplyr

What is dplyr?

- The next iteration of plyr,
- Focusing on only data frames (also tibble),
- Row-based manipulation,
- dplyr is faster and has a more consistent API.

dplyr provides a consistent set of verbs that help you **solve the most common data manipulation challenges**:

- select() Select columns, based on column name rules.
- filter() Filter rows by rule.
- mutate() Add new column, calculated from other columns (no change in number of rows).

- summarise() Converts multiple values to a single value (via mean, median, sd, etc.), generating new columns (total number of rows is reduced, usually in conjunction with group_by).
- arrange() Sorting the rows.

```
# Read the file
library(tidyverse)
mouse.tibble =
  read_delim(
    file = "data/mouse_genes_biomart_sep2018.txt",
    delim = "\t",
    quote = "",
    show_col_types = FALSE
  )

# View mouse.tibble content
ttype.stats =
  mouse.tibble %>%
    count(`Transcript type`) %>%
    arrange(-n)

# View mouse.tibble content, cont.
chr.stats =
  mouse.tibble %>%
    count(`Chromosome/scaffold name`) %>%
    arrange(-n)
```

R for bioinformatics, data wrangler, part 2

Talk 06

TOC

tidyr

- pivot_longer() to take the place of gather
- pivot_wider() to take the place of spread

Data Wrangler - `tidyr`

You can get `tidyr` in the package set `tidyverse`, or simply install it the first time you want to use it via `install.packages("tidyr")`.

The usage of `tidyr`

- Interconversion of wide and long data

```
# Eg 1
library(tidyverse)
grades2 =
        read_tsv(file = "data/grades2.txt")

grades3 =
        grades2 %>%
        pivot_longer(
    - name,
    names_to = "course",
    values_to = "grade"
  )

# Eg 2
grades3_wide = grades3_long %>%
  pivot_wider(
    names_from = "course",
    values_from = "grade"
  )
```

What's the difference between wide and long data?

Here are pros and cons of wide data:

- Pros:
- Natural and easy to understand;
- Cons:
- Not easy to handle;
  - More problematic when sparse.

If you meet `NA` in the 1st example, you can do like this:

```
grades3_1 =
        grades3[!is.na(grades3$grade), ]
grades3_2 =
        grades3[complete.cases(grades3), ]

# A better solution
grades3_long = grades2 %>%
  pivot_longer( - name,
                names_to = "course",
                values_to = "grade",
                values_drop_na = TRUE)

# Pay attention to the variant named "values_drop_na"
```

More functions in `tidyr`: (See @ https://r4ds.hadley.nz/data-tidy.html)

- `tidyr::separate()`

Usage:

```
separate(
  data,
  col,
  into,
  sep = "[^[:alnum:]]+",
  remove = TRUE,
  convert = FALSE,
  extra = "warn",
  fill = "warn",
  ...
)

# Default parameters are listed.
```

- `tidyr::unite()`

Usage:

```
unite(
  data,
  col,
  ...,
  sep = "_",
  remove = TRUE,
  na.rm = FALSE
)

# Default parameters are listed.
```

R for bioinformatics, Strings and regular expression

Talk 07

TOC

stringr

1. basics

- length
- uppercase, lowercase
- unite, separate
- string comparisons, sub string

1. regular expression

Note that each link listed below provides many further links to other interesting facilities and topics, you can ***ONLY*** open the links through VSCode built-in WebView. Sorry for the temporary inconvenience, but due to time constraints I can only fix this bug *in the next release*, please contact me if you need help.

Basics

Before you start...

```
library(stringr)
```


Also notice other famous packages used to manipulating string:

`stringi`(Following are based on the official R Documentation)

Description `stringi` is THE R package for fast, correct, consistent, and convenient string/text manipulation. It gives predictable results on every platform, in each locale, and under any native character encoding.

Facilities available Refer to the following:

- about_search for string searching facilities; these include pattern searching, matching, string splitting, and so on. The following independent search engines are provided:
- about_search_regex – with ICU (Java-like) regular expressions,
  - about_search_fixed – fast, locale-independent, byte-wise pattern matching,
  - about_search_coll – locale-aware pattern matching for natural language processing tasks,
  - about_search_charclass – seeking elements of particular character classes, like “all whites-paces” or “all digits”,
  - about_search_boundaries – text boundary analysis.
- `stri_datetime_format` for date/time formatting and parsing. Also refer to the links therein for other date/time/time zone- related operations.
- `stri_stats_general` and `stri_stats_latex` for gathering some fancy statistics on a character vector's contents.
- `stri_join`, `stri_dup`, `%s+%`, and `stri_flatten` for concatenation-based operations.
- `stri_sub` for extracting and replacing substrings, and `stri_reverse` for a joyful function to reverse all code points in a string.
- `stri_length` (among others) for determining the number of code points in a string. See also `stri_count_boundaries` for counting the number of Unicode characters and `stri_width` for approximating the width of a string.
- `stri_trim` (among others) for trimming characters from the beginning or/and end of a string, see also about_search_charclass, and `stri_pad` for padding strings so that they are of the same width. Additionally, `stri_wrap` wraps text into lines.
- `stri_trans_tolower` (among others) for case mapping, i.e., conversion to lower, UPPER, or Title Case, `stri_trans_nfc` (among others) for Unicode normalization, `stri_trans_char` for translating individual code points, and `stri_trans_general` for other universal text transforms, including transliteration.

- `stri_cmp`, `%s<%`, `stri_order`, `stri_sort`, `stri_rank`, `stri_unique`, and `stri_duplicated` for collation-based, locale-aware operations, see also about_locale.
- `stri_split_lines` (among others) to split a string into text lines.
- `stri_escape_unicode` (among others) for escaping some code points.
- `stri_rand_strings`, `stri_rand_shuffle`, and `stri_rand_lipsum` for generating (pseudo)random strings.
- `stri_read_raw`, `stri_read_lines`, and `stri_write_lines` for reading and writing text files.

Usage of `writeLines()` (from official R Documentation)

Description Write text lines to a connection.

Usage

```
writeLines(text, con = stdout(), sep = "\n", useBytes = FALSE)
```

Arguments

`text`	A character vector
`con`	A connection object or a character string.
`sep`	character string. A string to be written to the connection after each line of text.
`useBytes`	logical. See 'Details'.

Details If the `con` is a character string, the function calls `file` to obtain a file connection which is opened for the duration of the function call. (tilde expansion of the file path is done by `file`.)

If the connection is open it is written from its current position. If it is not open, it is opened for the duration of the call in `"wt"` mode and then closed again.

Normally `writeLines` is used with a text-mode connection, and the default separator is converted to the normal separator for that platform (LF on Unix/Linux, CRLF on Windows). For more control, open a binary connection and specify the precise value you want written to the file in `sep`. For even more control, use `writeChar` on a binary connection.

`useBytes` is for expert use. Normally (when false) character strings with marked encodings are converted to the current encoding before being passed to the connection (which might do further re-encoding). `useBytes = TRUE` suppresses the re-encoding of marked strings so they are passed byte-by-byte to the connection: this can be useful when strings have already been re-encoded by e.g. `iconv`. (It is invoked automatically for strings with marked encoding `"bytes"`.)

Difference between double quote("") and single quote('')

In R and its string manipulation package `stringr`, there is no difference between strings defined with double quotes (`"`) and single quotes (`'`). Both are used to define strings and you can use either depending on your preference or the situation.

For instance, if your string contains a single quote, you might find it easier to enclose the string in double quotes, and vice versa. Here's an example:

```r
# Using double quotes when the string contains a single quote
string1 = "It's a beautiful day"

# Using single quotes when the string contains a double quote
string2 = 'He said, "Hello, world!"'
```

In both cases, R will interpret the contents between the quotes as a string.

Some of the functions in the `stringr` package are similar in function to those that come with the system.

Here are examples comparing some functions from the `stringr` package with their counterparts from base R:

1. **`str_length()` vs. `nchar()`:**

```r
library(stringr)

# Using str_length from stringr
string = c("apple", NA, "banana", "")
str_length(string)
# Output: 5   NA   6   0

# Using nchar from base R
nchar(string)
# Output: 5  NA   6   0
```

Both `str_length()` and `nchar()` count the number of characters in each string element. However, `str_length()` handles missing values (`NA`) more consistently by returning `NA`, whereas `nchar()` might treat `NA` differently in certain cases.

1. **`str_sub()` vs. `substr()`:**

```r
# Using str_sub from stringr
string = c("hello", "world", "example")
```

```r
str_sub(string, start = 2, end = 4)
# Output: "ell" "orl" "xam"

# Using substr from base R
substr(string, start = 2, stop = 4)
# Output: "ell" "orl" "xam"
```

Both `str_sub()` and `substr()` extract substrings based on specified start and end positions. However, `str_sub()` allows negative indices to count from the end of the string, and it handles missing values more consistently.

1. **`str_replace()` vs. `sub()` or `gsub()`:**

```r
# Using str_replace from stringr
string = c("apple pie", "banana bread", "cherry cake")
str_replace(string, pattern = "a", replacement = "X")
# Output: "Xpple pie" "bXnXnX breXd" "chXrry cXke"

# Using sub from base R
sub(pattern = "a", replacement = "X", x = string)
# Output: "Xpple pie" "bXnana bread" "cherry cake"
```

Both `str_replace()` and `sub()` are used for replacing parts of a string. However, `str_replace()` has a more intuitive interface and handles missing values more gracefully compared to `sub()`.

These examples demonstrate how `stringr` functions can be more consistent and user-friendly in handling various string operations compared to their base R counterparts.

Some of the functions in the `stringi` package are similar in function to those that come with the system.

Here are some functions in the `stringi` package that share similar functionalities with base R's string functions, along with examples showcasing their differences:

1. **`stri_length()` vs. `nchar()`**:
   - `stri_length()` in `stringi` calculates the number of code points in a string, accounting for Unicode characters.
   - `nchar()` in base R counts the number of characters in a string, but it might not handle Unicode characters as accurately as `stri_length()`.

```r
library(stringi)
```



```r
# Using stri_length from stringi
string = "café"
stri_length(string)
# Output: 4

# Using nchar from base R
nchar(string)
# Output: 4
```

In this example, both `stri_length()` and `nchar()` return the same count for ASCII characters. However, when dealing with Unicode characters, `stri_length()` can accurately count them as individual code points, whereas `nchar()` might not handle them correctly.

2. **`stri_split_*()` vs. `strsplit()`**:
   - `stri_split_*()` functions in `stringi` split a string based on various criteria like fixed patterns, regular expressions, or character classes.
   - `strsplit()` in base R performs a similar operation but might differ in handling certain edge cases and Unicode characters.

```r
# Using stri_split_* from stringi
string = "apple, orange, café"
stri_split_fixed(string, pattern = ", ")
# Output: list("apple", "orange", "café")

# Using strsplit from base R
strsplit(string, split = ", ")
# Output: list("apple", "orange", "caf", "é")
```

Here, `stri_split_fixed()` correctly splits the string, including the accented character "é," while `strsplit()` treats the accented "é" as two separate characters due to how it handles Unicode.

3. **`stri_detect()` vs. `grepl()`**:
   - `stri_detect()` in `stringi` checks if a pattern exists in a string and returns a logical value.
   - `grepl()` in base R performs a similar task but might differ in its handling of Unicode characters and certain pattern matching options.

```r
# Using stri_detect from stringi
string = "This is a café"
stri_detect(string, regex = "café")
# Output: TRUE
```

```r
# Using grepl from base R
grepl("café", string)
# Output: FALSE
```

In this example, `stri_detect()` correctly detects the presence of the word "café," while `grepl()` returns a different result due to potential differences in Unicode handling or pattern matching options.

The examples highlight how `stringi` functions like `stri_length()`, `stri_split_*()`, and `stri_detect()` differ from their base R counterparts (`nchar()`, `strsplit()`, and `grepl()`) by providing more accurate handling of Unicode characters and often more versatile string manipulation options.

(In the slide) Difference between `toupper()`, `tolower()` and `stri_reverse()`

The functions `toupper()` and `tolower()` in base R and `stri_reverse()` in the `stringi` package perform similar tasks, but there are some differences in their functionality and usage:

1. **`toupper()` and `tolower()` in Base R:**
   - `toupper()` converts characters in a string to uppercase.
   - `tolower()` converts characters in a string to lowercase.

```r
# Using toupper and tolower from base R
string = "Hello World!"

toupper(string)
# Output: "HELLO WORLD!"

tolower(string)
# Output: "hello world!"
```

These functions are straightforward and work well for ASCII characters, converting them to uppercase or lowercase, respectively. However, they might not handle Unicode characters or locale-specific transformations.

2. **`stri_reverse()` in `stringi`:**
   - `stri_reverse()` reverses the order of characters in a string, including handling multibyte characters and Unicode sequences.

```r
library(stringi)

# Using stri_reverse from stringi
string = "café"

stri_reverse(string)
# Output: "éfac"
```

`stri_reverse()` reverses the characters in the string accurately, even when dealing with Unicode characters or multibyte sequences. It ensures correct reversal of characters irrespective of their encoding.

The key distinction lies in the handling of character cases and character sequence reversal. While `toupper()` and `tolower()` focus on case transformations for ASCII characters, `stri_reverse()` in `stringi` concentrates on accurately reversing character sequences, making it more suitable for handling multibyte characters and Unicode strings.

Tricks

- `stringi` The functions in the package all start with `stri_`.
- `strinr` starts with `str_`.

Regex - Regular Expression

1. Character classes:What characters are (not) matched?

```r
# Example 01
"abc_123_??$$^" %>% str_extract("\\s+") # Does this string include spaces?
"abc_123_??$$^" %>% str_extract("\\d+") # Numbers?
"abc_123_??$$^" %>% str_extract("\\w+") # [A-z0-9_]
```

`str_extract` : Take out the first match.

1. Matching position

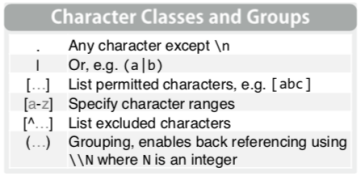
```r
# Example 02
# STRING ending in 'wei'
c("chen wei hua", "chen wei", "chen") %>% str_subset("wei$")

# CHARACTER ending in 'wei'
c("chen wei hua", "chen wei", "chen") %>% str_subset("wei\\b")
```

1. Number of matches

```r
# Example 03
"1234abc" %>% str_extract("\\d+")
```

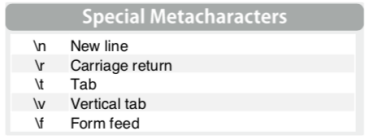
```
"1234abc" %>% str_extract("\\d{3}")
"1234abc" %>% str_extract("\\d{5,6}")
"1234abc" %>% str_extract("\\d{2,6}")
```

1. Classes and groups



Character Classes and Groups	
.	Any character except \n
|	Or, e.g. (a|b)
[…]	List permitted characters, e.g. [abc]
[a-z]	Specify character ranges
[^…]	List excluded characters
(…)	Grouping, enables back referencing using \\N where N is an integer

2. Special characters



Special Metacharacters	
\n	New line
\r	Carriage return
\t	Tab
\v	Vertical tab
\f	Form feed

R for bioinformatics, data iteration & parallel computing

Talk 08

TOC

- for loop
- `apply` functions
- The essence of `dplyr` is traversal.
- `map` functions in `purrr` package
- Iteration and Parallel Computing

Iteration Basics

`for loop` , getting data ready

Lookat this example:

```
df =
        tibble(
    a = rnorm(100),
    b = rnorm(100),
    c = rnorm(100),
    d = rnorm(100)
  )

# Calculate row means
res1 =
        vector("double", nrow(df))
for(row_idx in 1:nrow(df)){
  res1[row_idx] =
          mean( as.numeric(df[row_idx, ]))
}

res2 = c()
for(row_idx in 1:nrow(df)){
  res2[length(res2) + 1] =
          mean(as.numeric(df[row_idx, ]))
}

# Similar to Python

# Calculate column means
res2 =
        vector("double", ncol(df))
for(col_idx in 1:ncol(df)){
  res2[col_idx] =
          mean(df[[col_idx]])
}
```

You can replace it with `for loop`:

```
rowMeans(df)
colMeans(df)
```

Here are some other functions:

```
rowSums(df)
colSums(df)
```

apply functions

You can use `apply` with customizable function.

```
df %>% apply(
  .,
  2,
  function(x) {
    return(
      c(
        n = length(x),
        mean = mean(x),
        median = median(x)
      )
    )
  }
)
```

Something about `tapply()`:

The `tapply()` function in R is used to apply a function over subsets of a vector, splitting it by a factor or list of factors. It stands for "table apply" and is particularly useful for summarizing data by groups or categories.

Here's a breakdown of its usage:

```
tapply(X, INDEX, FUN)
```

- `X`: The vector (or array) on which you want to apply the function.
- `INDEX`: A factor or list of factors that define the groups. These factors determine how the vector `X` is split.
- `FUN`: The function to be applied to each subset of `X`.

For example:

```
# Creating a sample vector and a corresponding factor
values = c(10, 20, 30, 40, 50)
```



```
categories = factor(c("A", "B", "A", "B", "A"))

# Applying the mean function over subsets defined by the categories
tapply(values, categories, mean)
```

In this example, `tapply()` calculates the mean of the `values` vector for each category defined by the `categories` factor. It splits the `values` vector into subsets based on the categories and applies the `mean()` function to each subset, returning the mean values for categories "A" and "B".

The result would be something like:

```
A    B
30   30
```

This indicates that the mean value for category "A" is 30, and the mean value for category "B" is also 30 in this case.

`lapply()`, `sapply()`, and similar functions work on elements of lists or vectors. In contrast, `tapply()` focuses on splitting a vector by factors and applying a function to these subsets, providing aggregated results for each subset determined by the factors.

Diffrences between `apply` in base R and the package `dplyr`:

1. **`apply` functions in base R:**
   - The `apply` family of functions (`apply()`, `lapply()`, `sapply()`, `vapply()`, etc.) in base R are used for applying a function over margins of arrays or data structures like matrices, arrays, and lists.
   - `apply()` is used primarily for applying functions to the rows or columns of matrices or arrays, while `lapply()` and `sapply()` are more focused on lists.
   - These functions are useful for repetitive operations across rows or columns without explicitly using loops.

   Example:

   ```
   # Creating a matrix
   mat = matrix(1:12, nrow = 3, ncol = 4)

   # Applying sum function to rows (1) or columns (2) of the matrix
   apply(mat, 1, sum)  # Sums of each row
   apply(mat, 2, sum)  # Sums of each column
   ```

2. **`dplyr` package:**

- `dplyr` is a powerful package in R for data manipulation and transformation. It provides a set of functions (`filter()`, `mutate()`, `select()`, `group_by()`, `summarize()`, etc.) that enable easy and intuitive data manipulation.
- It's designed to work well with data frames and offers a more streamlined and readable syntax for performing common data manipulation tasks.

Example:

```
library(dplyr)

# Creating a sample data frame
df = data.frame(
  Name = c("Alice", "Bob", "Charlie"),
  Age = c(25, 30, 28),
  Salary = c(40000, 50000, 45000)
)

# Filtering and selecting specific rows and columns
filtered_df = df %>%
  filter(Age > 25) %>%
  select(Name, Salary)

filtered_df
```

This `dplyr` example filters rows where `Age` is greater than 25 and selects only the `Name` and `Salary` columns. The `%>%` operator (pipe) chains together multiple operations, making the code more readable and concise.

In summary, the `apply` family in base R is ideal for applying functions to matrices, arrays, or lists across rows or columns, while `dplyr` focuses on intuitive data manipulation operations for data frames, providing a cleaner syntax and ease of use for common data transformation tasks.

More on iteration: purrr package

About purrr (from official website https://purrr.tidyverse.org)

`purrr` enhances R's functional programming (FP) toolkit by providing a complete and consistent set of tools for working with functions and vectors. If you've never heard of FP before, the best place to start is the family of `map()`functions which allow you to replace many for loops with code that is both more succinct and easier to read. The best place to learn about the `map()` functions is the iteration chapter in R for data science.

Usage

The following example uses `purrr`to solve a fairly realistic problem: split a data frame into pieces, fit a model to each piece, compute the summary, then extract the R^2.

```
library(purrr)

mtcars |>
  split(mtcars$cyl) |>  # from base R
  map(\(df) lm(mpg ~ wt, data = df)) |>
  map(summary) %>%
  map_dbl("r.squared")

#>         4         6         8
#> 0.5086326 0.4645102 0.4229655
```

This example illustrates some of the advantages of `purrr` functions over the equivalents in base R:

- The first argument is always the data, so purrr works naturally with the pipe.
- All `purrr` functions are type-stable. They always return the advertised output type (`map()` returns lists; `map_dbl()` returns double vectors), or they throw an error.
- All `map()` functions accept functions (named, anonymous, and lambda), character vector (used to extract components by name), or numeric vectors (used to extract by position).

Detailed Usage

`purrr` is a powerful package in R that focuses on enhancing and simplifying the process of working with functions and vectors. Developed as part of the tidyverse ecosystem, `purrr` provides a consistent and coherent set of tools for functional programming, iteration, and working with lists and vectors.

Here are some key aspects and functionalities of `purrr`:

1. **Functional Programming:**
   - `purrr` promotes functional programming paradigms in R, enabling users to work with functions as first-class objects.
   - It provides functions like `map()`, `map2()`, `pmap()`, `walk()`, and more, which allow applying functions over elements of lists or vectors.
2. **Consistency Across Data Structures:**
   - `purrr` functions exhibit consistent behavior across various data structures, such as lists, vectors, and data frames.

- These functions can work seamlessly with different data structures, making code more readable and maintainable.

3. **Iteration and Mapping:**
   - `map()` is a key function in `purrr` that iterates over elements of a list or vector, applying a function to each element and returning the results.
   - `map2()` is similar to `map()` but allows iterating over two vectors simultaneously.
   - `pmap()` extends this functionality to iterate over multiple vectors or lists simultaneously.
4. **Simplified and Cleaner Syntax:**
   - `purrr` functions often provide a more consistent and cleaner syntax compared to base R functions for similar operations.
   - The use of the pipe `%>%` from the tidyverse allows chaining `purrr` functions together, resulting in more readable code.
5. **Working with Lists and Data Frames:**
   - `purrr` provides functions to efficiently manipulate and iterate over elements in lists and data frames.
   - Functions like `map()` and `map_dbl()` can be used to apply functions to each column of a data frame and collect results in an output structure.

Example of `map()` in `purrr`:

```r
library(purrr)

# Applying sqrt function to each element of a list
numbers = list(a = 1:5, b = 6:10)
result = map(numbers, sqrt)

result
# Output: List of 2
# $ a: num [1:5] 1 1.41 1.73 2 2.24
# $ b: num [1:5] 2.45 2.65 2.83 3 3.16
```

This code applies the `sqrt()` function to each element of the list `numbers`, returning a list with the square roots of each element.

`purrr` simplifies and enhances functional programming in R, offering a consistent and expressive way to work with functions, lists, vectors, and data frames, making data manipulation and iteration more straightforward and concise.

Examples

Here are some specific functionalities and examples of `purrr`:

1. **Mapping Functions:**

   `map()` applies a function to each element of a list or vector, returning a list.

   ```r
   library(purrr)

   # Squaring each element in a vector using map()
   numbers = 1:5
   squared = map(numbers, ~ .x^2)

   squared
   # Output: List of 5
   # $ : int 1
   # $ : int 4
   # $ : int 9
   # $ : int 16
   # $ : int 25
   ```

2. **Mapping Functions over Multiple Inputs:**

   `map2()` allows applying a function that takes two inputs to corresponding elements of two vectors.

   ```r
   # Multiplying elements of two vectors element-wise
   vector1 = 1:5
   vector2 = 6:10
   product = map2(vector1, vector2, ~ .x * .y)

   product
   # Output: List of 5
   # $ : int 6
   # $ : int 14
   # $ : int 24
   # $ : int 36
   # $ : int 50
   ```

3. **Working with Data Frames:**

   `map_df()` and similar functions allow applying a function to each column of a data frame and combining results into a data frame.

   ```r
   # Creating a data frame
   df = data.frame(A = 1:5, B = 6:10)
   ```

```r
# Doubling each column in the data frame
doubled = map_df(df, ~ .x * 2)

doubled
# Output: A tibble: 5 × 2
#       A     B
#   <dbl> <dbl>
# 1     2    12
# 2     4    14
# 3     6    16
# 4     8    18
# 5    10    20
```

4. **Iteration and Applying Functions:**

   `walk()` applies a function to each element without returning a result, useful for side effects or performing operations without output.

```r
# Printing each element of a list using walk()
fruits = list("apple", "banana", "orange")
walk(fruits, print)
# Output:
# [1] "apple"
# [1] "banana"
# [1] "orange"
```

`purrr` simplifies functional programming by providing intuitive functions (`map()`, `map2()`, `walk()`, etc.) that allow iteration over elements, applying functions, and collecting results, making code more concise and readable in scenarios involving lists, vectors, and data frames.

(in the slide) Function `reduce()` and `accumulate()`

Both `reduce()` and `accumulate()` are powerful functions from the `purrr` package that facilitate iterative calculations over a sequence, accumulating or reducing values based on a specified function.

Here's an explanation of each:

1. **`reduce()` Function:**

   - `reduce()` is used to successively apply a function to pairs of elements in a sequence, reducing it to a single value.
   - The function provided to `reduce()` should take two arguments and return a single value.

   - It starts by applying the function to the first two elements, then uses the result along with the next element, and so on, until the sequence is exhausted.

   Example of `reduce()`:

```r
library(purrr)

# Summing all elements in a vector using reduce()
numbers = 1:5
total_sum = reduce(numbers, `+`)

total_sum
# Output: 15 (1 + 2 + 3 + 4 + 5 = 15)
```

   Here, `reduce()` adds all the elements in the `numbers` vector by applying the addition function (`+`) iteratively.

2. **`accumulate()` Function:**

   - `accumulate()` is similar to `reduce()` but returns a sequence of intermediate results instead of a single value.
   - It applies a function cumulatively to the sequence and returns a vector of values, representing the intermediate results at each step.

   Example of `accumulate()`:

```r
# Calculating cumulative product of elements in a vector using accumulate()
factors = c(2, 3, 4, 5)
cumulative_product = accumulate(factors, `*`)

cumulative_product
# Output: 2 6 24 120 (2, 2*3, 2*3*4, 2*3*4*5)
```

   Here, `accumulate()` applies the multiplication function (`*`) to each element in `factors`, returning a vector of cumulative products at each step.

`reduce()` aggregates a sequence into a single value based on a function, while `accumulate()` returns a sequence of intermediate results. Both functions are helpful for iterative calculations and provide different ways to process sequences of values in R.

Parallel Computing

Related Packages

- `parallel` package: displays the number of CPU cores to assign all or part to a task.

- `foreach` package: provides `%do%` and `%dopar%` operators to submit tasks for sequential or parallel computation
- `` ` iterators `` package: splits `data.frame, tibble, matrix` into rows/columns for submitting parallel tasks.

Step-by-step Guidance

1. **Prepare Data:**

   Assume you have a large data frame named `my_data` that you want to process in parallel.

2. **Setup Parallel Processing:**

   Load necessary packages and initialize parallel processing capabilities.

```
library(parallel)
library(doParallel)
library(foreach)
library(iterators)

# Set the number of cores/processors to be used
num_cores = detectCores()

# Initialize parallel backend
cl = makeCluster(num_cores)
registerDoParallel(cl)
```

3. **Split Data Frame into Chunks:**

   Use the `iter()` function from the `iterators` package to create an iterator for chunks of your data frame.

```
# Define chunk size
chunk_size = nrow(my_data) / num_cores

# Create an iterator for the chunks
my_iterator = iter(my_data, by = "row", chunksize = chunk_size)
```

4. **Perform Parallel Computation:**

   Use `foreach()` from the `foreach` package along with `%dopar%` to apply a function to each chunk in parallel.

```
# Define a function to process each chunk
process_chunk = function(chunk) {
  # Your processing logic for each chunk goes here
  # For example: summary(chunk)
```



```
  # Replace summary() with your specific data processing task
}

# Apply the function to each chunk in parallel
results = foreach(chunk = my_iterator, .combine = rbind) %dopar% {
  process_chunk(chunk)
}
```

5. **Combine Results:**

   Collect and combine the results obtained from parallel processing.

```
# Combine or process the results obtained from parallel computation
final_result = do.call(rbind, results)

# Close the parallel cluster
stopCluster(cl)
```

Replace the `process_chunk()` function with your specific data processing task. This approach parallelizes the processing of chunks of the data frame across multiple cores, allowing for faster computations, especially with large datasets.

Note:

- When the task is completed, the allocated CPU core is reclaimed.
- Ensure that your specific data processing task is compatible with parallelization and that the benefits of parallel computing outweigh the overhead of parallelization.
- Also, consider potential dependencies or shared resources among iterations when parallelizing computations.

(in the slide) Function `foreach()`

Simple usage

Description `%do%` and `%dopar%` are binary operators that operate on a `foreach` object and an `R` expression. The expression, `ex`, is evaluated multiple times in an environment that is created by the `foreach` object, and that environment is modified for each evaluation as specified by the `foreach` object. `%do%` evaluates the expression sequentially, while `%dopar%` evaluates it in parallel. The results of evaluating `ex` are returned as a list by default, but this can be modified by means of the `.combine` argument.

Usage

```
foreach(
  ...,
  .combine,
  .init,
  .final = NULL,
  .inorder = TRUE,
  .multicombine = FALSE,
  .maxcombine = if (.multicombine) 100 else 2,
  .errorhandling = c("stop", "remove", "pass"),
  .packages = NULL,
  .export = NULL,
  .noexport = NULL,
  .verbose = FALSE
)
e1 %:% e2
when(cond)
obj %do% ex
obj %dopar% ex
times(n)
```

Nested `foreach`

Expanded knowledge, not featured on slide, for understanding only.

Nested `foreach` loops in R allow for the iteration over multiple levels of nested structures or combinations of iterators. This approach is particularly useful when dealing with hierarchical data or when you need to perform computations on multiple levels of nested objects simultaneously.

Here's an overview of nested `foreach` loops:

1. **Nested Iteration:**

   `foreach` supports nesting, allowing you to iterate over multiple levels of nested structures, such as lists within lists or matrices within lists.

   ```
   library(foreach)

   # Example: Nested foreach loop iterating over a list of lists
   outer_list = list(list(a = 1, b = 2), list(c = 3, d = 4))

   foreach(inner_list = outer_list) %:% {
     foreach(element = inner_list) %do% {
       # Process each element within the nested structure
       print(element)
     }
   }
   ```

   This code iterates over each element of `outer_list`, which contains inner lists. Within each inner list, it iterates over the elements.

2. **Combining Iterators:**

   You can combine different iterators using `%:%` to create nested iterations.

   ```
   # Example: Nested foreach loop with combined iterators
   values = 1:3
   letters = letters[1:4]

   foreach(i = values) %:% foreach(letter = letters) %do% {
     # Perform operations using both iterators
     print(paste("Value:", i, "| Letter:", letter))
   }
   ```

   This code creates nested iterations, iterating over `values` and `letters` simultaneously.

3. **Applying Nested Functions:**

   Nested `foreach` loops are valuable when applying functions or performing operations that require iterating over multiple levels of nested data structures or combinations.

   ```
   # Example: Applying a function with nested foreach loops
   matrix_list = list(matrix(1:4, nrow = 2), matrix(5:8, nrow = 2))

   foreach(mat = matrix_list) %:% foreach(element = as.vector(mat)) %do% {
     # Perform computations on each element of each matrix
     print(element * 2)
   }
   ```

   Here, it iterates over a list of matrices and then iterates over each element within the matrices to perform computations.

Nested `foreach` loops in R allow for flexible and efficient iterations over hierarchical or nested structures, enabling complex computations, data manipulations, or simulations involving multiple levels of nested objects or iterators.

Review of the course “R for Data Science” Part 03(Talk 09~12)

By Haoran Nie @ HUST Life ST

R for bioinformatics, data visualisation

Talk 09

TOC

Basic plot functions using R

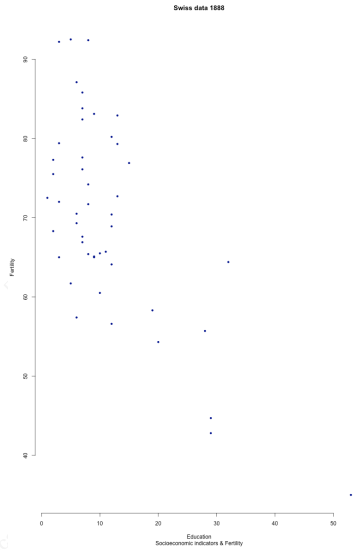
Dot plot

An example:

```
with(
  swiss,
  plot(
    Education,
    Fertility,
    type = "p",
    main = "Swiss data 1888",
    sub = "Socioeconomic indicators & Fertility",
    xlab = "Education",
    ylab = "Fertility",
    col = "darkblue",
    xlim = range( Education ),
```



```
    ylim = range( Fertility ),
    pch = 20,
    frame.plot = F
  )
)
```



Function usage:

```
## Default S3 method:
plot(
  x,
  y = NULL,
  type = "p",
  xlim = NULL, ylim = NULL,
  log = "",
  main = NULL, sub = NULL,
  xlab = NULL, ylab = NULL,
  ann = par("ann"), axes = TRUE, frame.plot = axes,
  panel.first = NULL, panel.last = NULL, asp = NA,
  xgap.axis = NA, ygap.axis = NA,
  ...
```

)
 # Default Parameters are listed.

Arguments

PARAMETERS	DETAILS
`x, y`	the `x` and `y` arguments provide the x and y coordinates for the plot. Any reasonable way of defining the coordinates is acceptable. See the function `xy.coords` for details. If supplied separately, they must be of the same length.
`type`	1-character string giving the type of plot desired. The following values are possible, for details, see `plot`: `"p"` for points, `"l"` for lines, `"b"` for both points and lines, `"c"` for empty points joined by lines, `"o"` for overplotted points and lines, `"s"` and `"S"` for stair steps and `"h"` for histogram-like vertical lines. Finally, `"n"` does not produce any points or lines.
`xlim`	the x limits (x1, x2) of the plot. Note that `x1 > x2` is allowed and leads to a 'reversed axis'.The default value, `NULL`, indicates that the range of the finite values to be plotted should be used.
`ylim`	the y limits of the plot.
`log`	a character string which contains `"x"` if the x axis is to be logarithmic, `"y"` if the y axis is to be logarithmic and `"xy"` or `"yx"` if both axes are to be logarithmic.
`main`	a main title for the plot, see also `title`.
`sub`	a subtitle for the plot.
`xlab`	a label for the x axis, defaults to a description of `x`.
`ylab`	a label for the y axis, defaults to a description of `y`.
`ann`	a logical value indicating whether the default annotation (title and x and y axis labels) should appear on the plot.
`axes`	a logical value indicating whether both axes should be drawn on the plot. Use graphical parameter `"xaxt"` or `"yaxt"` to suppress just one of the axes.
`frame.plot`	a logical indicating whether a box should be drawn around the plot.
`panel.first`	an 'expression' to be evaluated after the plot axes are set up but before any plotting takes place. This can be useful for drawing background grids or scatterplot smooths. Note that this works by lazy evaluation: passing this argument from other `plot` methods may well not work since it may be evaluated too early.
`panel.last`	an expression to be evaluated after plotting has taken place but before the axes, title and box are added. See the comments about `panel.first`.
`asp`	the `y/x` aspect ratio, see `plot.window`.
`xgap.axis, ygap.axis`	the `x/y` axis gap factors, passed as `gap.axis` to the two `axis()` calls (when `axes` is true, as per default).

You can also use `ggplot` to draw the plot above:

```
ggplot(
  swiss,
  aes(x = Education, y = Fertility)
) +
geom_point() +
scale_x_log10() +
scale_y_log10() +
xlab("Education") +
ylab("Fertility") +
ggtitle "Swiss data 1888")
```


High-level and low-level

- **high level**: plotting functions create a new plot on the graphics device
- **low level**: plotting functions add more information to an existing plot

Low level plots

- `points` : dot plot
- `lines` : line plot
- `abline` : straightness
- `polygon` : polygonal
- `legend` : legend (of a map, etc)
- `title` : caption
- `axis` : axis

High level plots

- `plot` : Generalized Graphing Functions
- `pairs`
- `coplot`
- `qqnorm`
- `hist`
- `dotchart`
- `image`
- `contour`

Notice

You can force a high level function to be converted to low level with the `add = TRUE` parameter (if available).

Graphics-related parameters (system functions)

You can use `par()` to view all the parameters current grafical device use.

Remember to backup your default parameters before modyfy them.

ggplot2

You should know that

1. `xy -axes` will automatically adjust based on the data you give;
2. `ggplot2` plotting results can be saved in variables and more layers can be added;
3. Layers using their own data need to be specified with `data =`, while global data is not. You can just specify it via `ggplot(data = data.frame(...))`.

Some basic parameters of ggplot2

1. `geom` - Layer

   `geom_<name_of_the_layer>`

- `geom_point` , ` `geom_line` : A point and line plot used to reveal the relationship between two sets of data;
- `geom_smooth` : Often used in conjunction with `geom_point` to reveal the trend of data.
- `geom_bar` : bar charts
- `geom_boxplot` : A box plot, used to compare N sets of data, revealing differences.
- `geom_path` : similar to `geom_line` but can also draw other complex graphs
- `geom_histogram`, `geom_density` : Distribution of data, can also be used to compare multiple groups.

  ……

1. `scale` - Display Control

   `scale_<property_to _control>_<ways_to_control>`

- `scale_color_...`
  - `..._gradient()`: use gradient colors for different numbers of variables and their colors.
    - `..._gradient2()`
    - `..._gradientn()`
  - `..._brewer`: use default color palettes.
- `scale_fill_...`
  - `colour` defines the colour with which a geom is **outlined** (the shape's "stroke")

  - `fill`defines the colour with which a geom is **filled**
  - Points generally only have a `colour` and **no fill**
  - However, point shapes **21**–**25** that include **both** a colour and a fill.
- `scale_shape_...`
- `scale_size_...`

Palettes and corresponding functions in other packages

Included in ggplot2 `scale_color_hue`, `scale_color_manual`, `scale_color_grey`, `scale_colour_viridis_d`, `scale_color_brewer` ...

From the RColorBrewer package `scale_color_brewer(palette = "<palette name>")`

From the viridis package `scale_color_viridis(discrete=TRUE, option="<palette name>")`

Other packages ...

- `paletteer` package: `scale_color_paletteer_xx` functions
- `ggsci` package
- `ggsci`: You can use it in your SCI article.
  - contents

    `scale_color_<journal>` 和 `scale_fill_<journal>` functions and color palettes

- supported journals

  * NPG `scale_color_npg()`, `scale_fill_npg()`

  * AAAS, NEJM, Lancet, JAMA ...

1. `size`
2. `shape`

Question:

Parameters like `size`, `colour` etc. can be inside or outside `aes()`, what is the difference?

Answer:

In the internal case, the **size** is determined by the value of the specified column, or the **number** of colors and shapes is determined by the number of factors, and in the external case, the **specified value** prevails.

Coordinate System

1. Linear coordinate system

- `coord_cartesian()`,

  You can use parameters like `xlim()` & `ylim()` to **zoom in and out locally**.

- `coord_flip()`,

  This parameter exchanges the `x` and `y` axes.

- `coord_fixed()`

  Draw plots in specific **aspect ratio**.

1. Nonlinear coordinate system

- `coord_trans()`

  This parameter exchanges the `x` and `y` axes, **however**, it will plot the original figure, too.

- `coord_polar()`

  The default parameter is `coord_polar(x)`.

- `coord_map()`

  World Map

`faceting`

panel, strip, axis, tick, tick label, axis label…

- `facet_wrap()`

  Use to specify the number of rows, columns and orientation.

```
facet_wrap(
  facets,
  nrow = NULL,
  ncol = NULL,
  scales = "fixed",
  shrink = TRUE,
  labeller = "label_value",
  as.table = TRUE,
  switch = NULL,
  drop = TRUE,
  dir = "h",
  strip.position = "top"
)

# Default parameters are listed.
```


Different layouts

Nothing to explain, for it's too intricate.

Formulas

> Similar to LaTeX

You can just type the formula as exactly what you type in LaTeX, and using `annotiate()` function to add it in your plot.

Remember to attach the library `latex2exp`

```
fig1 =
  fig1 +
  annotate(
    "text",
    x = 25,
    y = 15,
    label = paste0("y = ", eq, "x + (", intercept, ")\n","Shaded areas are confidence interv
    family = "Aptos Serif"
  )

... +
labs(
    x = TeX("Position of $P_2/\\phi$"),
    y = TeX("Light Intensity/$10^{-7}$A"),
    title = TeX("Intensity of Light with Different $\\phi$")
  )
```

Something about `hjust` and `vjust`

```
td = expand.grid(
    hjust=c(0, 0.5, 1),
    vjust=c(0, 0.5, 1),
    angle=c(0, 45, 90),
    text="text"
)

ggplot(td, aes(x=hjust, y=vjust)) +
    geom_point() +
    geom_text(aes(label=text, angle=angle, hjust=hjust, vjust=vjust)) +
    facet_grid(~angle) +
```

```
scale_x_continuous(breaks=c(0, 0.5, 1), expand=c(0, 0.2)) +
scale_y_continuous(breaks=c(0, 0.5, 1), expand=c(0, 0.2))
```

You should always remember, the first step you want to draw a plot is to calculate the data.

R for bioinformatics, data summarisation and statistics

Talk 10

TOC

- Data summarisation functions (vector data)
  - median, mean, sd, quantile, summary
- Graphical data summarisation (two-D data/ tibble/ table)
  - dot plot
  - smooth
  - linear regression
  - correlation & variance explained
  - groupping & bar/ box/ plots
- statistics
  - parametric tests
    - t-test
    - one way ANNOVA
    - two way ANNOVA
    - linear regression
    - model / prediction / coefficients
  - non-parametric comparison

Vector Summarization

Describe Normal Distribution

You can use `mean` and `sd` to describe normal distributions.

- It's symmetrical.
- Mean and median are the same.
- Most common values are near the mean; less common values are farther from it.
- Standard deviation marks the distance from the mean to the inflection point.

Functions to generate random normal distrubions

```
qnorm()
pnorm()
dnorm()
```

Other regular distributions

1. Uniform Distributions

   ```
   dunif(x, min = 0, max = 1, log = FALSE)
   punif(q, min = 0, max = 1, lower.tail = TRUE, log.p = FALSE)
   qunif(p, min = 0, max = 1, lower.tail = TRUE, log.p = FALSE)
   runif(n, min = 0, max = 1)
   ```

2. Non-parametric Distributions

   Here's an example:

   ```
   bi =
           c(7, 3, 2, 1, 7,
       3, 4, 5, 7, 6,
       2, 2, 1, 3, 7,
       2, 6, 8, 2, 7,
       2, 2, 1, 3, 5,
       8, 2, 6, 7, 8,
       6, 2, 8, 7, 9,
       2, 7, 5, 1, 8,
       8, 2, 3, 7, 3
     )
   ggplot(
     data.frame(dat = bi),
     aes(dat)) +
   geom_density()
   ```

Quantitative descriptive data

- **mean**: aka average, is the sum of all of the numbers in the data set divided by the size of the data set;
- **median**: The median is the value that is in the middle when the numbers in a data set are sorted in increasing order;
- **sd**: standard deviation;
- **var**: measures how far a set of numbers are spread out;
- **range**: range of values.

Quantitative descriptive function

Besides the function with the same name of the data above, `quantile` and `summary` are two quantitative descriptive functions.

This chapter contains lots of functions and their usage, to know more, you can see them in the official R documentation, I only explane the things above here.

Statistics

Parametric tests

`t-test` Detect whether the distribution is consistent with expectations; e.g., whether the number of steps per day for boys is significantly different from 10,000.

The test assesses whether the means of two samples are significantly different from each other, assuming that the samples are normally distributed and have approximately equal variances.

There are different types of t-tests in R, depending on the nature of the comparison:

1. **One-Sample t-test:**

   Used to determine if the mean of a single sample differs significantly from a known or hypothesized population mean.

   Example:

   ```
   RCopy code
   # One-sample t-test example
   sample_data = c(17, 21, 19, 23, 20, 18, 22)
   t.test(sample_data, mu = 20)
   ```

   This code performs a one-sample t-test on `sample_data` to test if its mean differs significantly from 20.

2. **Independent Samples t-test (or Two-Sample t-test):**

   Compares the means of two independent groups to determine if they are significantly different from each other.

   Example:

   ```
   RCopy code
   # Independent samples t-test example
   group1 = c(23, 25, 28, 22, 20)
   group2 = c(18, 21, 24, 19, 17)
   t.test(group1, group2)
   ```

   This code performs an independent samples t-test on `group1` and `group2` to test if their means are significantly different.

3. **Paired t-test:**

   Compares the means of two related groups (e.g., before and after measurements) to determine if they are significantly different.

   Example:

   ```
   RCopy code
   # Paired t-test example
   before = c(32, 28, 30, 29, 31)
   after = c(30, 25, 28, 27, 29)
   t.test(before, after, paired = TRUE)
   ```

   This code performs a paired t-test on `before` and `after` to test if there's a significant difference.

The `t.test()` function in R is used to conduct these t-tests. It returns a test statistic (t-value), degrees of freedom, p-value, and confidence interval, providing insights into whether the difference observed is statistically significant.

It's important to ensure that the assumptions of normality and equal variances are met for reliable results when performing t-tests in R. If the assumptions are violated, alternative tests or data transformations might be more appropriate.

One-way ANNOVA In R, the one-way analysis of variance (ANOVA) is used to test for significant differences between the means of three or more independent (unrelated) groups. It assesses whether the means of these groups are significantly different from each other.

The one-way ANOVA assumes that the data meet certain assumptions, including:

- Normality: Each group should follow a normal distribution.
- Homogeneity of variances: The variances within each group should be approximately equal.

- Independence: The observations within each group should be independent of each other.

Here's an example of performing a one-way ANOVA in R:

```r
# Example of one-way ANOVA
group1 = c(15, 20, 25, 30, 35)
group2 = c(10, 18, 25, 32, 40)
group3 = c(12, 22, 28, 32, 38)

# Combining data into a data frame
my_data = data.frame(
  Values = c(group1, group2, group3),
  Group = factor(rep(1:3, each = 5))  # Creating a factor for groups
)

# Performing one-way ANOVA
result_anova = aov(Values ~ Group, data = my_data)
summary(result_anova)
```

Explanation of the code:

1. The data for three groups (`group1`, `group2`, `group3`) are created.
2. The data are combined into a data frame (`my_data`) where the `Values` column contains the measurements and the `Group` column represents the group labels as a factor.
3. The `aov()` function is used to perform the one-way ANOVA, specifying the formula `Values ~ Group`, indicating that `Values` is the dependent variable and `Group` is the independent variable.
4. `summary(result_anova)` provides the ANOVA table with the F-statistic, degrees of freedom, p-value, and other relevant statistics.

The output from `summary(result_anova)` will include information such as the F-statistic, degrees of freedom, p-value, and within-group variability, allowing you to determine if there are significant differences between the means of the groups.

If the p-value is less than a chosen significance level (commonly 0.05), it suggests that there are significant differences among the means of the groups. Additionally, post-hoc tests like Tukey's HSD test or pairwise t-tests can be performed to identify which specific groups differ significantly from each other after obtaining a significant result in the ANOVA.

Two-way ANNOVA A two-way analysis of variance (ANOVA) in R is used to examine the interaction effects between two categorical independent variables (factors) on a continuous dependent variable.

Here's an example:

```r
# Example of two-way ANOVA
# Assume we have a dataset with 'Treatment', 'Gender', and 'Response' variables

# Creating sample data
set.seed(123)
Treatment = rep(c("A", "B", "C"), each = 20)
Gender = rep(c("Male", "Female"), times = 30)
Response = rnorm(60, mean = c(50, 60, 70), sd = 10)

# Combining data into a data frame
my_data = data.frame(Treatment, Gender, Response)

# Performing two-way ANOVA
result_anova = aov(Response ~ Treatment + Gender + Treatment:Gender, data = my_data)
summary(result_anova)
```

Explanation of the code:

1. Sample data is created with three variables: `Treatment`, `Gender`, and `Response`.
2. The data are combined into a data frame (`my_data`), where `Treatment` and `Gender` are categorical factors, and `Response` is the continuous dependent variable.
3. The `aov()` function performs the two-way ANOVA. The formula `Response ~ Treatment + Gender + Treatment:Gender` specifies the main effects of `Treatment` and `Gender`, as well as their interaction effect.
4. `summary(result_anova)` provides the ANOVA table with F-statistics, degrees of freedom, p-values, and other statistics for each factor and their interaction.

The output from `summary(result_anova)` will include information about the main effects of `Treatment` and `Gender`, as well as the interaction effect between them. It allows you to determine if there are significant effects of each factor independently and whether their interaction significantly influences the `Response` variable.

The interpretation of a two-way ANOVA involves analyzing the p-values associated with each factor and their interaction. Significant p-values indicate that the corresponding factor or interaction has a significant effect on the dependent variable. Additionally, post-hoc tests or further analyses can be conducted to explore specific comparisons between groups or factors after obtaining significant results in the ANOVA.

Linear Regression Linear regression is a statistical method used to model the relationship between a dependent variable and one or more independent variables by fitting a linear equation to observed data. In R, linear regression can be performed using the `lm()` function, which stands for "linear model."

Here's an example:

```r
# Example of simple linear regression
# Suppose we have a dataset with 'x' as the independent variable and 'y' as the dependent va

# Creating sample data
set.seed(123)
x = 1:50
y = 2 * x + rnorm(50, mean = 0, sd = 5)  # Generating 'y' as a linear function of 'x' with s

# Creating a data frame
my_data = data.frame(x, y)

# Performing linear regression
model = lm(y ~ x, data = my_data)
summary(model)
```

Explanation of the code:

1. Sample data is generated with an independent variable `x` and a dependent variable `y`. In this example, `y` is generated as a linear function of `x` with some added noise using `rnorm()` to simulate real-world variability.
2. The data are combined into a data frame `my_data`.
3. The `lm()` function fits a linear regression model where `y` is the dependent variable and `x` is the independent variable (`y ~ x`). The argument `data = my_data` specifies the data frame containing the variables.
4. `summary(model)` provides a summary of the linear regression model, including coefficients, standard errors, t-values, p-values, R-squared, and other statistics.

Interpreting the output from `summary(model)`:

- The coefficients section shows the estimated coefficients for the intercept and the slope of the regression line (`Intercept` and `x`).
- The p-values associated with the coefficients indicate the significance of each variable in predicting the dependent variable. Lower p-values suggest stronger evidence against the null hypothesis of no effect.

- The R-squared value represents the proportion of variance in the dependent variable explained by the independent variable(s). Higher R-squared values indicate a better fit of the model to the data.

Linear regression in R can also be extended to multiple linear regression by including multiple independent variables in the model (`y ~ x1 + x2 + ...`). Additionally, diagnostic plots and further analyses can be performed to assess model assumptions and goodness of fit.

Model / Prediction / Coefficients Certainly! In linear regression, the model equation is expressed as:

$$[y = \beta_0 + \beta_1 \cdot x_1 + \beta_2 \cdot x_2 + ... + \beta_n \cdot x_n + \epsilon]$$

Where:

- (y) is the dependent variable.
- ($x_1, x_2, \ldots, x_n$) are the independent variables.
- (β_0) is the intercept (constant term).
- ($\beta_1, \beta_2, \ldots, \beta_n$) are the coefficients (slope parameters) that represent the change in (y) associated with a one-unit change in the corresponding (x) variable, assuming all other variables remain constant.
- (ϵ) represents the error term.

In R, after fitting a linear regression model using `lm()`:

```r
# Assuming 'model' is the linear regression model obtained previously
summary(model)
```

The output from `summary(model)` provides information including:

- **Coefficients:** This section displays the estimated coefficients (`Estimate`) for each independent variable, including the intercept. These coefficients represent the estimated change in the dependent variable for a one-unit change in the corresponding independent variable, holding other variables constant.
- **Residuals:** The residuals represent the differences between the observed and predicted values of the dependent variable. These are used to assess the goodness of fit of the model.
- **R-squared:** Indicates the proportion of variance in the dependent variable explained by the independent variables. Higher values indicate a better fit of the model to the data.

After obtaining the coefficients from the model, predictions can be made using new or existing data:

```r
# Assuming 'new_data' contains the new data for prediction
predicted_values = predict(model, newdata = new_data)
```

Replace `new_data` with the data for which you want to make predictions. The `predict()` function uses the coefficients from the model to generate predicted values of the dependent variable based on the independent variables in the new data.

The coefficients obtained from the linear regression model (`model$coefficients`) represent the slopes of the regression line and the intercept, which are crucial for predicting new values and understanding the relationships between variables in the model.

Non-parametric Comparison

Non-parametric methods are statistical techniques used when the assumptions of normality, homogeneity of variances, or linearity required by parametric methods are violated or not met by the data. These methods do not rely on specific population distribution assumptions and are useful for analyzing data that might not follow a normal distribution.

Here are some commonly used non-parametric methods for comparison in R:

1. **Mann-Whitney U Test (Wilcoxon Rank-Sum Test):**

   Used to compare two independent groups when the assumptions of t-tests are not met.

   Example:

   ```r
   # Assuming 'group1' and 'group2' are vectors of numeric data
   wilcox.test(group1, group2)
   ```

2. **Kruskal-Wallis Test:**

   An extension of the Mann-Whitney U test for comparing more than two independent groups.

   Example:

   ```r
   # Assuming 'group1', 'group2', 'group3' are vectors of numeric data
   kruskal.test(list(group1, group2, group3))
   ```

3. **Wilcoxon Signed-Rank Test:**

   Used for comparing paired samples or related samples.

   Example:

   ```r
   # Assuming 'before' and 'after' are vectors of paired numeric data
   wilcox.test(before, after, paired = TRUE)
   ```

4. **Mood's Median Test:**

   Tests the equality of medians in two or more independent groups.

   Example:


```r
# Assuming 'group1', 'group2', 'group3' are vectors of numeric data
median_test = mood.test(group1, group2, group3)
median_test
```

5. **Friedman Test:**

   An extension of the Wilcoxon Signed-Rank test for comparing more than two paired or related groups.

   Example:

   ```r
   # Assuming 'group1', 'group2', 'group3' are matrices or data frames of paired numeric da
   friedman.test(group1, group2, group3)
   ```

These non-parametric tests provide alternatives to traditional parametric tests and are robust against violations of certain assumptions. They are particularly useful when dealing with ordinal or skewed data or when the sample size is small, as they rely on fewer distributional assumptions than parametric tests.

Linear and nonlinear regression

Talk 11

This topic is particularly complex, so it is voluminous and obscure.

TOC

Linear Regression

What is linear regression?

Linear regression is a method of statistical analysis that utilizes regression analysis in mathematical statistics to determine interdependent quantitative relationships between two or more variables.

- `Y` can be explained by a variable `X`: One-way Linear Regression
- `Y` can be explained by multiple variables such as `X`, `Z`: Multiple Linear Regression

Linear regression in R is typically performed using the `lm()` function, which stands for "linear model." Here's how to fit a linear regression model in R and some useful functions related to linear regression:

Fitting a Linear Regression Model:

```
# Example of fitting a linear regression model
# Assuming 'my_data' is a data frame with 'x' as the independent variable and 'y' as the dep
model = lm(y ~ x, data = my_data)
summary(model)  # Display summary statistics of the model
```

- **`lm()` Function:** Fits a linear regression model. The formula `y ~ x` specifies that `y` is the dependent variable and `x` is the independent variable. `data = my_data` indicates the data frame containing the variables.
- **`summary()` Function:** Displays a summary of the linear regression model, including coefficients, standard errors, t-values, p-values, R-squared, and other statistics.

Other Useful Functions for Linear Regression Analysis:

1. **`coefficients()` Function:**

   Retrieves the coefficients of the linear regression model.

   Example:

   ```
   coef = coefficients(model)
   coef
   ```

2. **`predict()` Function:**

   Generates predictions using the fitted model.

   Example:

   ```
   new_data = data.frame(x = c(10, 20, 30))  # New data for prediction
   predicted_values = predict(model, newdata = new_data)
   predicted_values
   ```

3. **`residuals()` Function:**

   Retrieves the residuals (differences between observed and predicted values).

   Example:


   ```
   residuals = residuals(model)
   residuals
   ```

4. **`fitted()` Function:**

   Retrieves the fitted (predicted) values.

   Example:

   ```
   fitted_values = fitted(model)
   fitted_values
   ```

5. **`vcov()` Function:**

   Computes the variance-covariance matrix of the coefficients.

   Example:

   ```
   var_cov_matrix = vcov(model)
   var_cov_matrix
   ```

6. **`anova()` Function:**

   Performs analysis of variance (ANOVA) for the fitted model.

   Example:

   ```
   anova_table = anova(model)
   anova_table
   ```

7. **`summary()` Function:**

   Provides an overview of the fitted model, including parameter estimates, standard errors, convergence information, and goodness-of-fit statistics.

   Example:

   ```
   summary(model)
   ```

8. **`coef()` Function:**

   Extracts the estimated coefficients from the fitted model.

   Example:

   ```
   coef(model)
   ```

9. **`confint()` Function (for prediction intervals):**

   Computes prediction intervals for the predicted values from the fitted model.

   Example:

   ```
   prediction_intervals = predict(model, interval = "prediction")
   prediction_intervals
   ```

10. **`deviance()` Function:**

    Calculates the deviance of the fitted model, which is a measure of lack of fit.

    Example:

    ```
    deviance(model)
    ```

11. **`update()` Function:**

    Allows for updating or refitting a model with different settings or data.

    Example:

    ```
    updated_model = update(model, start = list(a = 2, b = 2, c = 2))
    summary(updated_model)
    ```

12. **`AIC()` and `BIC()` Functions:**

    Compute Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC) to assess model quality and compare models.

    Example:

    ```
    AIC(model)
    BIC(model)
    ```

13. **`plot()` Function (for diagnostic plots):**

    Generates diagnostic plots to assess the adequacy of the model (e.g., residuals vs. fitted values, Q-Q plots).

    Example:

    ```
    plot(model)
    ```

These functions help in obtaining and analyzing various aspects of the linear regression model, such as coefficients, predictions, residuals, variance-covariance matrix, and ANOVA tables, aiding in model interpretation, examining the fitted model's statistics, coefficients, goodness-of-fit measures, prediction intervals, model comparisons, and diagnostics, allowing for a comprehensive analysis of nonlinear regression models in R.

Nolinear Regression

Nonlinear regression is used when the relationship between variables cannot be adequately described by a linear model. In R, fitting nonlinear models involves estimating parameters to describe the nonlinear relationship between variables. Here's an example using the `nls()` function, along with some relevant functions for nonlinear regression analysis:

Fitting a Nonlinear Regression Model:

```
# Example of fitting a nonlinear regression model (assuming a quadratic function)
# Assuming 'my_data' is a data frame with 'x' as the independent variable and 'y' as the dep

# Fitting a quadratic model: y = a * x^2 + b * x + c
model = nls(y ~ a * I(x^2) + b * x + c, data = my_data, start = list(a = 1, b = 1, c = 1))
summary(model)  # Display summary statistics of the model
```

- **`nls()` Function:** Fits a nonlinear regression model. The formula `y ~ a * I(x^2) + b * x + c` specifies a quadratic function. `data = my_data` indicates the data frame containing the variables, and `start = list(a = 1, b = 1, c = 1)` provides initial parameter values.
- **`summary()` Function:** Displays a summary of the nonlinear regression model, including parameter estimates, standard errors, t-values, convergence information, and other statistics.

Other Useful Functions for Nonlinear Regression Analysis:

1. **`predict()` Function:**

    Generates predictions using the fitted nonlinear model.

    Example:

    ```
    new_data = data.frame(x = c(10, 20, 30))  # New data for prediction
    predicted_values = predict(model, newdata = new_data)
    predicted_values
    ```

2. **`residuals()` Function:**

    Retrieves the residuals (differences between observed and predicted values).

    Example:

    ```
    residuals = residuals(model)
    residuals
    ```

3. **`confint()` Function:**

    Computes confidence intervals for model parameters.

    Example:

    ```
    conf_intervals = confint(model)
    conf_intervals
    ```

4. **`nls.control()` Function:**

   Provides control parameters for the `nls()` function, allowing adjustments to the nonlinear fitting process.

   Example:

```
control_params = nls.control(maxiter = 100, tol = 1e-6)
model = nls(y ~ a * I(x^2) + b * x + c, data = my_data, start = list(a = 1, b = 1, c = 1
```

5. **`anova()` Function:**

   Performs analysis of variance (ANOVA) for the fitted nonlinear model.

   Example:

```
anova_table = anova(model)
anova_table
```

6. **`nlsLM()` Function (from the `minpack.lm` package):**

   - An alternative to `nls()` that provides enhanced convergence properties and extended functionality for nonlinear least squares.

   Example:

```
library(minpack.lm)
model_nlsLM = nlsLM(y ~ a * I(x^2) + b * x + c, data = my_data, start = list(a = 1, b =
summary(model_nlsLM)
```

7. **`augment()` Function (from the `broom` package):**

   - Creates a tidy data frame with additional columns like fitted/predicted values, residuals, and other model information.

   Example:

```
library(broom)
augmented_model = augment(model)
head(augmented_model)
```

8. **`glance()` Function (from the `broom` package):**

   - Extracts model-level statistics, providing a summary of the model in a tidy format.

   Example:

```
glance_summary = glance(model)
glance_summary
```

9. **`tidy()` Function (from the `broom` package):**

   - Extracts the model coefficients and related statistics into a tidy data frame.

   Example:

```
tidy_summary = tidy(model)
tidy_summary
```

10. **`nlstools::nlstools()` Function:**

    - Offers diagnostic tools and visualizations for nonlinear regression models, aiding in model evaluation.

    Example:

```
library(nlstools)
model_tools = nlstools(model)
plot(model_tools)
```

11. **`nls2()` Function:**

    - Provides an extended version of `nls()` with multiple start values to improve convergence.

    Example:

```
model_nls2 = nls2(y ~ a * I(x^2) + b * x + c, data = my_data, start = list(a = 1, b = 1,
summary(model_nls2)
```

These functions are commonly used for nonlinear regression analysis in R, helping in prediction, residual analysis, confidence interval computation, and controlling the fitting process.

Modeling and Prediction

When it comes to modeling and prediction using regression analysis, especially nonlinear regression, understanding the model, making predictions, and assessing the model's performance are crucial. Here's a step-by-step guide on how to approach modeling and prediction:

Modeling and Prediction Steps:

1. **Data Preparation:**

Prepare your dataset with variables for the dependent (response) and independent (predictor) variables.

1. **Fitting the Nonlinear Model:**

Fit the nonlinear regression model using `nls()` or similar functions, specifying the appropriate formula and initial parameter values.

Example:

```
model =
nls(y ~ a * I(x^2) + b * x + c,
    data = my_data,
    start = list(a = 1, b = 1, c = 1))
```

1. **Model Summary and Assessment:**

Use `summary()` and other functions (`glance()`, `tidy()`, etc.) to obtain an overview of the model, including coefficients, goodness-of-fit measures, and diagnostics.

Example:

```
summary(model)
```

1. **Prediction with the Fitted Model:**

Generate predictions using the fitted model for new or existing data.

Example:

```
new_data = data.frame(x = c(10, 20, 30))  # New data for prediction
predicted_values = predict(model, newdata = new_data)
predicted_values
```

1. **Model Evaluation:**

Evaluate the model's performance using various metrics (e.g., residuals, R-squared, RMSE) and diagnostic plots (e.g., residuals vs. fitted values, Q-Q plots) to assess how well the model fits the data.

1. **Adjustment and Refinement:**

Depending on the evaluation results, consider refining the model by adjusting parameters, exploring different models, or including/excluding variables to improve performance.

1. **Prediction Intervals and Uncertainty:**

Compute prediction intervals using `predict()` with `interval = "prediction"` to quantify the uncertainty around predictions.

1. **Model Comparison (if applicable):**

Compare multiple models using metrics like AIC, BIC, or likelihood ratio tests to select the most appropriate model.

By following these steps, you'll be able to build, evaluate, and utilize nonlinear regression models for prediction in R effectively. Remember, interpreting the model's results, assessing its assumptions, and validating predictions are crucial aspects of regression analysis.

K-fold & X times cross-validation

Both K-fold cross-validation and X times cross-validation are techniques used to assess the performance and generalization capability of machine learning models, including regression models, by partitioning the data into subsets for training and validation.

K-fold Cross-Validation:

K-fold cross-validation involves splitting the dataset into K equally sized folds. The model is trained K times, each time using K-1 folds for training and the remaining fold for validation. This process ensures that each data point is used for both training and validation.

Steps for K-fold Cross-Validation:

1. **Partition Data:** Split the dataset into K equally sized folds.
2. **Model Training:** Train the model K times, each time using K-1 folds as training data.
3. **Validation:** Validate the model's performance on the remaining fold (not used in training) and calculate evaluation metrics.
4. **Average Metrics:** Average the evaluation metrics across the K iterations to obtain an overall assessment of the model's performance.

X times Cross-Validation:

X times cross-validation, also known as repeated K-fold cross-validation, is similar to K-fold cross-validation, but the process is repeated X times. It repeatedly creates random partitions of the data into K folds, trains the model, and evaluates its performance. This method provides more robust estimates of model performance by averaging results over multiple runs.

Steps for X times Cross-Validation:

1. **Partition Data Repeatedly:** Randomly split the dataset into K folds, X times.
2. **Model Training:** Train the model for each iteration, using K-1 folds for training.

3. **Validation:** Validate the model's performance on the remaining fold and calculate evaluation metrics.

4. **Average Metrics:** Average the evaluation metrics across the X iterations to obtain more stable and reliable estimates of model performance.

Implementation in R:

In R, you can perform K-fold and X times cross-validation using functions from various packages like `caret`, `rsample`, or `crossval`. For example, using `caret`:

K-fold Cross-Validation in R with `caret`:

```
library(caret)
# Define a train control using k-fold cross-validation
train_control =
        trainControl(method = "cv", number = K)
# Specify K for the number of folds

# Train the model using k-fold cross-validation
model =
        train(
    formula,
    data = my_data,
    method = "lm",
    trControl = train_control
  )
```

X times Cross-Validation in R with `caret`:

```
library(caret)
# Define a train control using repeated k-fold cross-validation
train_control =
trainControl(
  method = "repeatedcv",
  number = K, repeats = X)
# Specify K for folds and X for repeats

# Train the model using repeated k-fold cross-validation
model =
        train(
    formula,
```



```
    data = my_data,
    method = "lm",
    trControl = train_control
  )
```

Replace `formula` and `my_data` with the appropriate regression formula and your dataset. Adjust the parameters `K` and `X` according to your preferences for the number of folds and repetitions.

These cross-validation techniques aid in estimating the model's performance and help in assessing how well the model generalizes to unseen data, thus providing insights into its robustness and reliability. Adjusting these techniques can improve the evaluation of regression models in terms of accuracy and stability.

External Validation

External validation refers to the evaluation of a predictive model's performance using an independent dataset that was not used in the model development process. It serves as an essential step to assess how well a model generalizes to new, unseen data from a different source or time period, verifying its reliability and robustness in real-world applications.

Steps for External Validation:

1. **Obtain an Independent Dataset:** Acquire a separate dataset that is distinct from the one used for model training and validation. This dataset should ideally represent the same problem or domain but come from a different source, time frame, or population.

2. **Preprocess Data:** Preprocess the independent dataset similarly to the training dataset (e.g., handling missing values, encoding categorical variables, scaling features) to ensure compatibility.

3. **Apply Trained Model:** Use the model trained on the original dataset to make predictions on the independent dataset.

4. **Evaluate Model Performance:** Assess the model's performance on the independent dataset using relevant evaluation metrics (e.g., accuracy, RMSE, precision, recall) and compare these metrics to the performance achieved on the training/validation dataset.

5. **Analyze Results:** Analyze the performance metrics obtained from the independent dataset to determine if the model maintains its predictive capability and generalizability. A well-performing model on the independent dataset suggests good generalization and reliability.

Importance of External Validation:

- **Generalization Assessment:** Validates whether the model's performance seen in training/validation data extends to new, unseen data.

- **Bias and Overfitting Detection:** Identifies if the model is overfitted to the training data or exhibits biases that could limit its real-world applicability.
- **Real-world Applicability:** Confirms the model's utility in practical scenarios and different environments.
- **Trustworthiness and Reliability:** Provides stakeholders with confidence in the model's predictions and results.

Implementation in R:

In R, the process involves loading the trained model and applying it to the independent dataset for prediction. Use appropriate evaluation functions (`predict()`, evaluation metrics) to assess the model's performance on the external dataset.

Example in R:

```r
# Load the trained model (replace 'model' with your trained model)
load("trained_model.RData")

# Load and preprocess the independent dataset
# (replace 'independent_data.csv' with your dataset)
independent_data = read.csv("independent_data.csv")
# Perform similar preprocessing steps as used for the training data

# Apply the trained model to the independent dataset for prediction
predicted_values = predict(model, newdata = independent_data)

# Evaluate model performance on the independent dataset
# Use appropriate evaluation metrics
# (e.g., RMSE, accuracy)
# and compare with training/validation results
```

Replace the file paths, data loading, and evaluation steps with your specific dataset and evaluation procedures. Ensure the compatibility of the independent dataset with the preprocessing steps applied to the original dataset for accurate evaluation.

Machine learning basics

Talk 12

This topic is particularly complex, so it is voluminous and obscure.

TOC

Machine Learning Algorithms Generalization

Machine learning can be categorized as follows:

1. Regression Algorithms
2. Example-based algorithms
3. Decision Tree Learning
4. Bayesian methods
5. Kernel-based algorithms
6. Clustering algorithms
7. Dimensionality reduction algorithms
8. Association rule learning
9. Integration algorithms
10. Artificial Neural Networks

For more information, you can read related articles.

Random Forest in Machine Learning using R

Certainly! Random Forest is a versatile and powerful machine learning algorithm used for both classification and regression tasks. It operates by building multiple decision trees during training and then merging their predictions to improve accuracy and robustness. Here's a guide on implementing Random Forest in R:

Steps to Implement Random Forest in R:

1. Load Required Library: Start by loading the necessary library (`randomForest`) if not already installed.

```r
install.packages("randomForest")  # Install package if not installed
library(randomForest)
```

2. Prepare Data: Load your dataset into R and perform necessary preprocessing steps such as handling missing values, encoding categorical variables, and splitting the data into training and testing sets.

```
# Example: Assuming 'my_data' is your dataset
# Split data into features (X) and target variable (Y)
X = my_data[, -target_column_index]  # Features
Y = my_data[, target_column_index]   # Target variable
```

3. Train the Random Forest Model: Use the `randomForest()` function to train the model by specifying the formula and training data.

```
# Example: Training a Random Forest model for regression
model = randomForest(Y ~ ., data = my_data)
# For classification: model = randomForest(factor(Y) ~ ., data = my_data)
```

4. Model Tuning (Optional): Adjust hyperparameters such as the number of trees (`ntree`), maximum depth (`max_depth`), and others to optimize model performance.

```
# Example: Setting number of trees and maximum depth
model = randomForest(Y ~ ., data = my_data, ntree = 100, max_depth = 10)
```

5. Make Predictions: Use the trained model to make predictions on new or test data.

```
# Example: Making predictions on test data
predicted_values = predict(model, newdata = test_data)
```

6. Model Evaluation: Evaluate the model's performance using appropriate metrics (e.g., RMSE for regression, accuracy, confusion matrix for classification) on test/validation data.

```
# Example: Evaluate model performance (for regression)
error = sqrt(mean((predicted_values - true_values)^2))  # Calculate RMSE
```

Example - Random Forest for Regression:

Here's an example using a built-in dataset (`mtcars`) in R for regression:

```
library(randomForest)

# Load dataset
data(mtcars)
```



```
# Split data into features and target variable
X = mtcars[, -1]  # Exclude the first column (target variable)
Y = mtcars[, 1]   # First column (target variable)

# Train the Random Forest model
model = randomForest(Y ~ ., data = mtcars)

# Make predictions on the same dataset (for demonstration)
predicted_values = predict(model, newdata = mtcars)

# Evaluate model performance (RMSE)
error = sqrt(mean((predicted_values - mtcars$mpg)^2))
```

Replace `my_data`, `test_data`, and the respective column names with your specific dataset and target variable. Adjust hyperparameters based on your problem and dataset characteristics for better model performance. Additionally, use appropriate evaluation metrics for assessing model accuracy and performance.

Feature Selection

Feature selection is a crucial step in machine learning aimed at choosing the most relevant and informative features to improve model performance, reduce computational complexity, and mitigate overfitting. In R, various methods can help identify important features.

Feature Selection Techniques in R:

1. Correlation Analysis: Calculate correlation coefficients between features and the target variable or among features themselves to identify highly correlated features. Remove redundant or highly correlated features.

```
# Example: Using cor() for correlation analysis
correlation_matrix = cor(my_data)
```

2. Filter Methods: Use statistical measures (e.g., chi-squared test, mutual information) to rank and select features based on their individual relevance to the target variable.

```
# Example: Using chi-squared test for feature selection
library(caret)
feature_selection = nearZeroVar(my_data)
```

3. Wrapper Methods: Evaluate subsets of features by training models iteratively and selecting the subset that results in the best model performance.

```
# Example: Using recursive feature elimination (RFE)
library(caret)
control = rfeControl(functions = rfFuncs, method = "cv", number = 10)
feature_selection = rfe(X, Y, sizes = c(1:10), rfeControl = control)
```

4. Embedded Methods: Feature selection during model training where algorithms inherently perform feature selection (e.g., LASSO, Random Forest, Gradient Boosting).

```
# Example: Using Random Forest for feature selection
library(randomForest)
model = randomForest(Y ~ ., data = my_data)
importance = importance(model)
```

5. Dimensionality Reduction Techniques: Techniques like Principal Component Analysis (PCA) or t-Distributed Stochastic Neighbor Embedding (t-SNE) reduce feature space by creating new variables that capture most of the original information.

```
# Example: PCA for feature reduction
pca_result = prcomp(my_data, scale. = TRUE)
```

Implementation Considerations:

- **Domain Knowledge:** Utilize domain expertise to guide feature selection, understanding the relevance of features in the context of the problem.
- **Cross-Validation:** Perform feature selection within cross-validation loops to prevent overfitting and ensure generalizability.
- **Evaluate Model Performance:** Assess the impact of feature selection on model performance using appropriate evaluation metrics.

Choose feature selection methods based on the dataset's characteristics, problem complexity, and computational resources. A combination of multiple techniques often yields the best results, as different methods capture different aspects of feature importance. Experiment and iterate to find the most suitable features for your machine learning models.

This is the end of the article. Take a rest~