

Logiciel R

Laurent Rouvière

janvier 2024

Table des matières

1	Introduction	2
1.1	Présentation du cours	2
1.2	Rstudio, quarto et packages R	10
2	Objets R	12
3	Gérer des données	19
3.1	Importer des données	19
3.2	Annexe : les fonctions read.table et read.csv	23
3.3	Base de données avancées	24
3.4	Fusion de tables	28
3.5	Manipuler les données avec Dplyr	30
3.6	Quelques fonctions utiles de tidyr	35
4	Bases de données	37
4.1	SQL : Structured Query Language	37
4.2	Peupler une base de données	39
4.3	Requêtes SQL	39
4.4	SQL et dplyr	41
4.5	JSON : JavaScript Object Notation	44
4.6	API WEB	50
5	Programmer en R	52
5.1	Structures de contrôle	52
5.2	Les fonctions map	54

6	Visualiser des données	56
6.1	Graphes conventionnels	56
6.2	Visualisation avec ggplot2	59
7	Cartes	71
7.1	ggplot2	72
7.2	Contours shapefile contours avec sf	75
7.3	Cartes interactives avec leaflet	82
7.4	Autres packages carto	88
8	Quelques outils de visualisation dynamiques	91
8.1	rAmCharts, plotly et ggiraph	92
8.2	Graphes avec visNetwork	98
8.3	Tableau de bord avec flexdashboard	101

1 Introduction

1.1 Présentation du cours

Présentation

- *Enseignant* : Laurent Rouvière, laurent.rouviere@univ-rennes2.fr
 - *Recherche* : statistique non paramétrique, apprentissage statistique.
 - *Enseignement* : statistique et probabilités (Université, école d'ingénieur, formation continue).
 - *Consulting* : énergie (ERDF), finance, marketing, sport.
- *Prérequis* : bases en programmation, probabilités et statistique.
- *Objectifs* : **comprendre et utiliser les outils R classiques** en datascience, plus particulièrement en visualisation :
 - importer et assembler des tables, manipuler des individus et des variables.
 - calculer des indicateurs statistiques
 - visualiser des données

Documents de cours

- *Slides* disponibles à l'url https://lrouviere.github.io/page_perso/cours/visualisationR.html

- *Tutoriel* : compléments de cours et exercices disponibles à l'url https://lrouviere.github.io/TUTO_VISU_R/

Ressources

- Le *net* : de nombreux tutoriels
- Livre : *R pour la statistique et la science des données*, PUR



Pourquoi R ?

- De plus en plus de *données*, dans de plus en plus de *domaines* (énergie, santé, sport, économie. . .)
- La *science des données* contient tous les outils qui permettent d'*extraire de l'information* à partir de données. Elle comprend :
 - l'importation de données
 - la manipulation
 - la visualisation
 - le choix et l'entraînement de modèles
 - la visualisation de modèles (ils sont de plus en plus complexes. . .)
 - la restitution et la visualisation des résultats (applications web)

Remarque importante

- Toutes ces notions peuvent être réalisées avec R.
- R (data scientists) et Python (informaticiens) font partie des outils les plus utilisés en sciences des données.

Quelques mots sur R

- R est un *logiciel libre et gratuit*.
- Il est distribué par le CRAN (Comprehensive R Archive Network) à l'url suivante : <https://www.r-project.org>.
- Tous les statisticiens (notamment) *peuvent contribuer* en créant des fonctions et en les distribuant à la communauté (*packages*).

Conséquence

- Le logiciel est *toujours à jour*.
- Une des principales raisons de son succès.

Exemple : Les Iris de Fisher

```
> data(iris)
> summary(iris)
  Sepal.Length   Sepal.Width   Petal.Length   Petal.Width
Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
Median :5.800   Median :3.000   Median :4.350   Median :1.300
Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500

  Species
setosa   :50
versicolor:50
virginica :50
```

Objectifs

Le problème

Expliquer **Species** par les autres variables.

- **Species** est *variable qualitative*.
- Confronté à un problème de *classification supervisée*.

Manipulation des données

```
> apply(iris[,1:4],2,mean)
Sepal.Length Sepal.Width Petal.Length Petal.Width
  5.843333    3.057333    3.758000    1.199333
> apply(iris[,1:4],2,var)
Sepal.Length Sepal.Width Petal.Length Petal.Width
  0.6856935    0.1899794    3.1162779    0.5810063
```

Remarque

Non informatif pour le problème (expliquer Species).

Manipulation avec dplyr

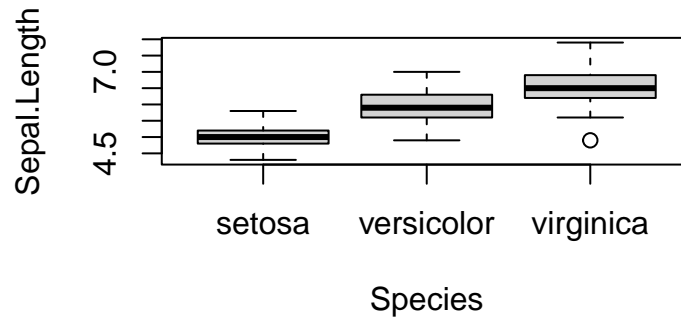
- **dplyr** est un package de **tidyverse** qui permet de faciliter la manipulation des données, notamment en terme de **syntaxe**.

```
> library(dplyr)
> iris |> group_by(Species) |> summarise_all(mean)
# A tibble: 3 x 5
  Species Sepal.Length Sepal.Width Petal.Length Petal.Width
  <fct>    <dbl>         <dbl>    <dbl>         <dbl>
1 setosa      5.01           3.43      1.46           0.246
2 versicolor  5.94           2.77      4.26           1.33
3 virginica   6.59           2.97      5.55           2.03
```

- *Plus intéressant* : nous obtenons les moyennes pour **chaque espèce**.

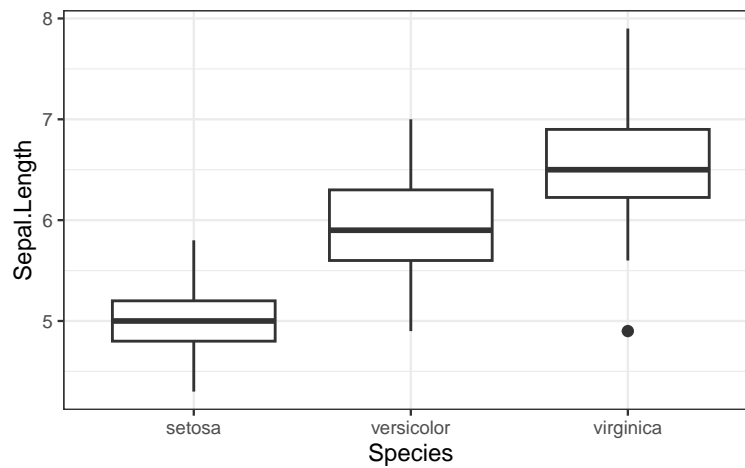
Visualisation

```
> boxplot(Sepal.Length~Species,data=iris)
```



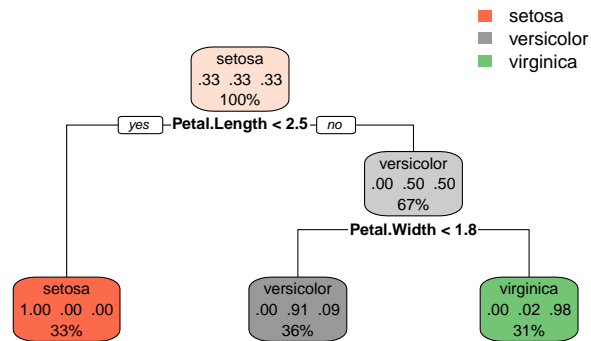
Visualisation avec ggplot2

```
> library(ggplot2)
> ggplot(iris)+aes(x=Species,y=Sepal.Length)+geom_boxplot()
```



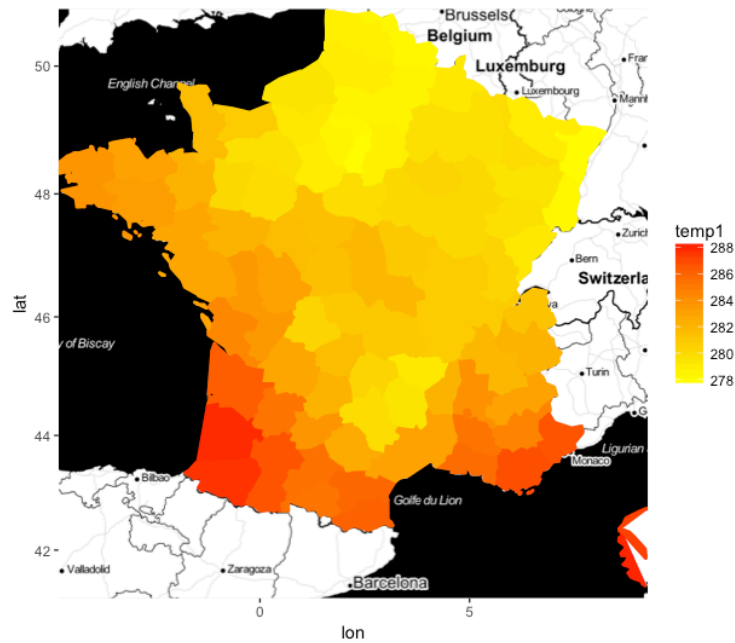
Un modèle d'arbre

```
> library(rpart)
> tree <- rpart(Species~.,data=iris)
> library(rpart.plot)
> rpart.plot(tree)
```



Carte avec ggmap

- *Objectif* : visualiser les températures en france pour une date donnée.



Chargement des données + fond de carte

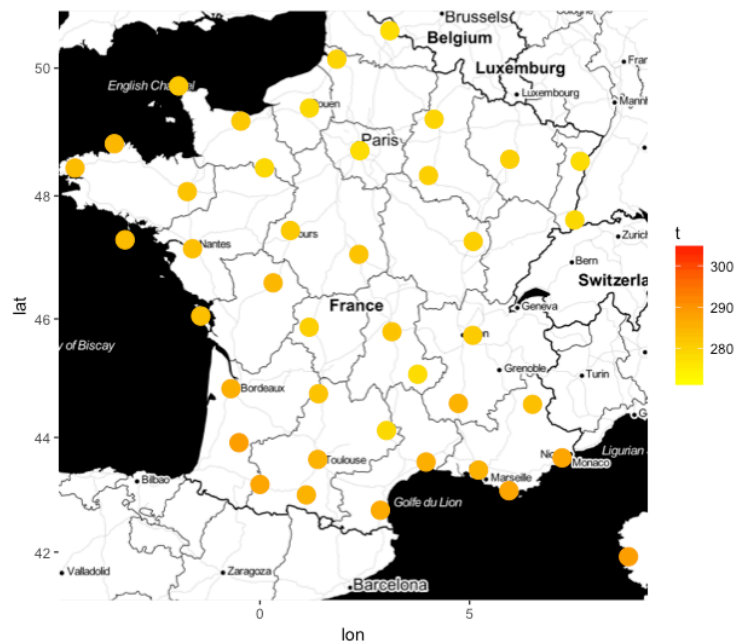
- Données téléchargées sur le site de meteofrance (températures d'à peu près 60 stations).

```

> donnees <- fread("https://donneespubliques.meteofrance.fr/
+                 donnees_libres/Txt/Synop/synop.2017082815.csv")
> station <- fread("https://donneespubliques.meteofrance.fr/
+                 donnees_libres/Txt/Synop/postesSynop.csv")
> fond <- get_map("France",maptype="toner",zoom=6)
> ggmap(fond)+geom_point(data=D,
+   aes(y=Latitude,x=Longitude,color=t),size=5)+
+   scale_color_continuous(low="yellow",high="red")

```

Une première carte



Modèle de prévision

- Algorithme de *plus proche voisins* pour estimer la température sur toutes les longitudes et latitudes du territoire.

```

> library(FNN)
> mod <- knn.reg(train=D[,.(Latitude,Longitude)],y=D[,t],
+               test=Test1[,.(Latitude,Longitude)],k=1)$pred

```

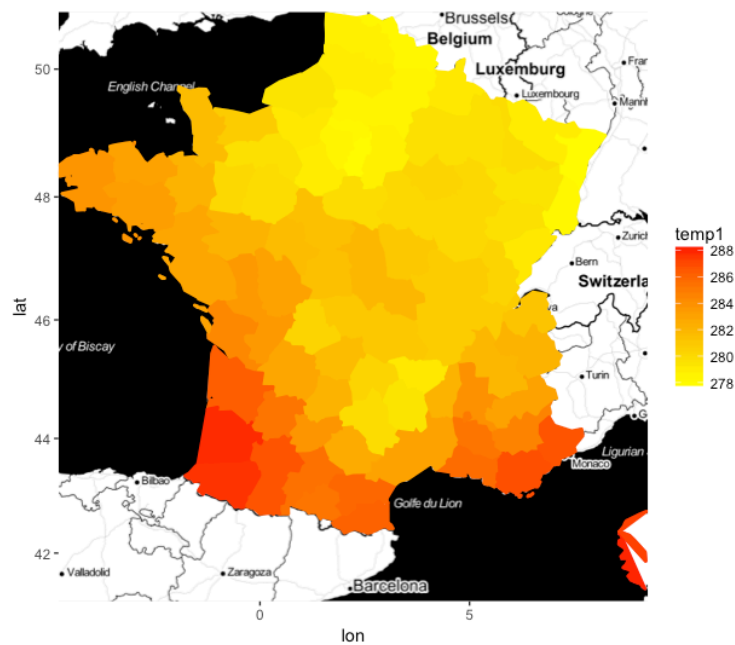
- Visualisation avec *ggmap*.


```

> library(ggmap)
> ggmap(fond)+geom_polygon(data=Test5,
+   aes(y=Latitude,x=Longitude,
+   fill=temp1,color=temp1,group=dept),size=1)+
+   scale_fill_continuous(low="yellow",high="red")+
+   scale_color_continuous(low="yellow",high="red")

```

La carte finale



Application web avec shiny

- Shiny est un package R qui permet la création de [pages web interactives](https://lrouviere.shinyapps.io/DESC_APP/).
- Exemples :
 - Graphiques descriptifs pour un jeu de données : https://lrouviere.shinyapps.io/DESC_APP/
 - Visualisation des *stations velib* à Rennes : <https://lrouviere.shinyapps.io/velib/>

Dans ce cours

- Logiciel R
 - *Environnement Rstudio* et reporting *quarto*
 - *Objets R*
 - *Manipuler* des données (dplyr)
- Visualisation
 - *Graphe statique* avec ggplot2
 - *Cartographie*
 - * *statique* avec ggmap et sf
 - * *dynamique* avec leaflet
 - *Visualisation dynamique*
 - * *graphes standards* avec rAmCharts et plotly
 - * *réseaux* avec visNetwork
 - * *tableaux de bord* avec flexdashboard
 - *Application web* avec shiny

1.2 Rstudio, quarto et packages R

Rstudio

- **RStudio** est une *interface* facilitant l'utilisation de R.
- Également *libre et gratuit* : <https://www.rstudio.com>.

L'écran est divisé en 4 parties :

- **Console** : pour entrer les commandes et visualiser les sorties.
- **Workspace and History** : visualiser l'historique des objets créés.
- **Files Plots...** : voir les répertoires et fichiers dans l'environnement de travail, les graphes de sortie, installer les packages...
- **Script** : éditeur pour entrer les commandes R et les commentaires. Penser à *régulièrement sauvegarder ce fichier* !

Quarto

Fichier Quarto

- Successeur de **Rmarkdown**
- Un fichier **Quarto (.qmd)** permet de produire un document de travail.
- Il contient le **code**, les **sorties** et des **commentaires** sur le travail réalisé.
- Il produit des **rapports de très bonne qualité** sous différents formats (documents, diaporama, etc...).
- Ce diaporama est réalisé avec *quarto*.
- *Recherche Reproductible* : en cliquant sur un bouton, on peut ré-exécuter tout le code du fichier et **exporter les résultats sous un format rapport**.
- *Documents dynamiques* : possibilité d'**exporter le rapport final dans différents formats** : html, pdf, rtf, slides, notebook...

Packages

- *Ensemble de programmes R* qui complètent et améliorent les fonctions de **R**.
- Un package est généralement dédié à des *méthodes ou domaines d'application spécifiques*.
- Plus de *19 000* packages actuellement.
- Contribue au *succès* de R (**toujours à jour**).

2 phases

- Installation: `install.packages(package.name)` (une seule fois).
- Chargement: `library(package.name)` (chaque fois).
- On peut aussi utiliser le bouton **package** dans **Rstudio**.

L'approche tidyverse

- *Meta package* dont les package partagent la même **philosophie**, **grammaire** et **structures de données** :
 - **readr** : lire les données
 - **dplyr**, **tidyr** : manipuler les données
 - **stringr** : chaînes de caractères

- `forcats` : facteurs (variables qualitatives)
- `ggplot2` : visualiser les données
- `purrr` : programmer
- ...
- *Doc* : <https://www.tidyverse.org>
- *Livre* : R for data science, Wickham et Grolemund, 2017. <https://r4ds.had.co.nz/index.html>
- La part d'utilisateurs qui utilise ces outils ne *cesse de croître*.

2 Objets R

Numérique et caractères

- Numérique (facile)

```
> x <- pi
> x
[1] 3.141593
> is.numeric(x)
[1] TRUE
> exp(x)
[1] 23.14069
```

Vecteurs

- *Création*: `c`, `seq`, `rep`

```
> x1 <- c(1,3,4)
> x2 <- 1:5
> x3 <- seq(0,10,by=2)
> x4 <- rep(x1,3)
> x5 <- rep(x1,3,each=3)
```

- *Extraction*: `[]`

```
> x3[c(1,3,4)] # pareil que x3[x1]
[1] 0 4 6
```

Caractères

- Avec des guillemets

```
> b <- "X"
> bb <- c("X", "Y", "Z")
> bb[c(1,3)]
[1] "X" "Z"
```

- Quelques fonctions utiles

```
> paste(b, 1:5, sep="")
[1] "X1" "X2" "X3" "X4" "X5"
> substr("livre", 2, 5)
[1] "ivre"
```

Caractères avec **stringr**

- Combiner plusieurs chaînes :

```
> str_c(b, 1:5, sep="-") # proche de paste
[1] "X-1" "X-2" "X-3" "X-4" "X-5"
```

- Autres fonctions :

```
> nom <- c("Marie", "Pierre", "Paul")
> str_length(nom)
[1] 5 6 4
> str_sub(nom, 1, 2) # proche de substr
[1] "Ma" "Pi" "Pa"
> str_detect(nom, "Mr")
[1] FALSE FALSE FALSE
> str_detect(nom, "[Mr]")
[1] TRUE TRUE FALSE
> str_replace(nom, "[Mr]", "?")
[1] "?arie" "Pie?re" "Paul"
```

Logique

- *Vrai ou Faux*

```
> 1<2
[1] TRUE
> 1==2
[1] FALSE
> 1!=2
[1] TRUE
```

- Souvent utile pour *sélectionner des composantes* d'un vecteur

```
> x <- 1:3
> test <- c(TRUE,FALSE,TRUE)
> x[test]
[1] 1 3
```

```
> size <- runif(5,150,190) #5 tailles générées aléatoirement entre 150 and 190
> size
[1] 178.8362 185.0309 180.4393 185.4450 168.2592
```

Problème

Sélectionner les tailles plus grandes que 174.

```
> size>174
[1] TRUE TRUE TRUE TRUE FALSE
> size[size>174]
[1] 178.8362 185.0309 180.4393 185.4450
```

Facteurs

- Pour représenter les *variables qualitatives* :

```
> x1 <- factor(c("a","b","b","a","a"))
> x1
[1] a b b a a
Levels: a b
> levels(x1)
[1] "a" "b"
```

Variable mal définie

- On suppose que les données sont *codées* : 0=homme, 1=femme

```
> X <- c(1,1,0,0,1)
> summary(X)
  Min. 1st Qu.  Median    Mean 3rd Qu.   Max.
   0.0    0.0    1.0    0.6    1.0    1.0
```

- *Problème* : **R** interprète X comme un vecteur *continu* cela peut générer des *problèmes* dans l'étude statistique.

- **Solution :**

```
> X <- as.factor(X)
> levels(X) <- c("man", "woman")
> X
[1] woman woman man   man   woman
Levels: man woman
> summary(X)
  man woman
    2     3
```

Facteurs avec **forcats**

- Exemple

```
> mois <- c("Dec", "Apr", "Jan", "Mar", "Apr")
> fct(mois)
[1] Dec Apr Jan Mar Apr
Levels: Dec Apr Jan Mar
```

les **levels** sont ordonnés par ordre d'apparition.

- Autres fonctions :

```
> fct_relevel(mois, c("Jan", "Mar", "Apr", "Dec"))
[1] Dec Apr Jan Mar Apr
Levels: Jan Mar Apr Dec
> fct_recode(mois, "decembre"="Dec", "janvier"="Jan")
[1] decembre Apr   janvier Mar   Apr
Levels: Apr decembre janvier Mar
```

- Compter :

```
> fct_count(mois)
# A tibble: 4 x 2
  f         n
  <fct> <int>
1 Apr     2
2 Dec     1
3 Jan     1
4 Mar     1
```

- Renommer

```
> sexe <- c(0,1,0,0,1)
> fct_recode(as.character(sexe), "woman"="1", "man"="0")
[1] man  woman man  man  woman
Levels: man woman
```

Matrice

- Création

```
> m <- matrix(1:4,nrow=2,byrow=TRUE)
> m
      [,1] [,2]
[1,]     1     2
[2,]     3     4
```

- Extraction

```
> m[1,2]
> m[1,] #Première ligne
> m[,2] #Seconde colonne
```

Liste

- Permet de regrouper *plusieurs objets* de *différents types* dans un même objet :

```
> mylist <- list(vector=1:5,mat=matrix(1:8,nrow=2))
> mylist
$vector
[1] 1 2 3 4 5

$mat
      [,1] [,2] [,3] [,4]
[1,]     1     3     5     7
[2,]     2     4     6     8
```

- Extraction:

```
> mylist[[1]]
> mylist$vector
> mylist[["vector"]]
```

Dataframe

- Objets pour représenter des *données* dans R.


```

> name <- c("Paul","Mary","Steven","Charlotte","Peter")
> sex <- c(0,1,0,1,0)
> size <- c(180,165,168,170,175)
> data <- data.frame(name,sex,size)
> data
  name sex size
1  Paul  0  180
2  Mary  1  165
3 Steven  0  168
4 Charlotte 1  170
5  Peter  0  175

```

```

> summary(data)
  name                sex                size
Length:5             Min.   :0.0      Min.   :165.0
Class :character      1st Qu.:0.0      1st Qu.:168.0
Mode  :character      Median :0.0      Median :170.0
                        Mean   :0.4      Mean   :171.6
                        3rd Qu.:1.0      3rd Qu.:175.0
                        Max.   :1.0      Max.   :180.0

```

Problème 1

`sex` est interprété comme une *variable continue*. C'est une *variable qualitative*.

Solution

Il faut la convertir en *facteur*.

```

> data$sex <- as.factor(data$sex)
> levels(data$sex) <- c("man","woman")
> summary(data)
  name                sex                size
Length:5             man   :3      Min.   :165.0
Class :character      woman:2      1st Qu.:168.0
Mode  :character                        Median :170.0
                        Mean   :171.6
                        3rd Qu.:175.0
                        Max.   :180.0

```

Problème 2

`name` est interprété comme une *variable*. C'est plutôt un *identifiant*.

```
> row.names(data) <- data$name
> data <- data[,-1] #suppression de la colonne name
> data
```

	sex	size
Paul	man	180
Mary	woman	165
Steven	man	168
Charlotte	woman	170
Peter	man	175

Conclusion

Il est *crucial* de toujours vérifier que les données sont *correctement interprétées* par **R** (avec `summary` ou `mode` par exemple).

Tibbles

- Un *tibble* est une version *moderne* du dataframe, qui conserve les avantages et supprime les inconvénients (selon les créateurs du tibble).
- C'est la version dataframe du *tidyverse* (nécessité de charger ce package).
- *Deux différences notables* :
 - les variables *qualitatives* sont par défaut des *caractères* (et non des facteurs) ;
 - pas de *rownames* par défaut (possibilité de les définir en convertissant le tibble en dataframe).

Exemple : data frame

```

> name <- c("Paul","Mary","Steven","Charlotte","Peter")
> sex <- c(0,1,0,1,0)
> size <- c(180,165,168,170,175)
> age <- c("old","young","young","old","old")
> data <- data.frame(sex,size,age)
> rownames(data) <- name
> data

```

	sex	size	age
Paul	0	180	old
Mary	1	165	young
Steven	0	168	young
Charlotte	1	170	old
Peter	0	175	old

Exemple : tibble

```

> data1 <- tibble(name,sex,size,age)
> #data2 <- column_to_rownames(data1,var="name")
> data1
# A tibble: 5 x 4
  name      sex  size age
<chr>   <dbl> <dbl> <chr>
1 Paul      0    180 old
2 Mary      1    165 young
3 Steven    0    168 young
4 Charlotte  1    170 old
5 Peter     0    175 old

```

dataframe vs tibble

Principale différence : **pas de facteur** dans les **tibbles** (par défaut), ni de **rownames**.

3 Gérer des données

3.1 Importer des données

- Les données sont généralement contenues dans des *fichiers* avec les individus en ligne et les variables en colonnes.

- Les fonctions `read.table` et `read.csv` permettent d'importer des données à partir de fichiers `.txt` et `.csv`.
- Le package `readr` du `tidyverse` propose des fonctions du même style dans l'esprit `tidy`, par exemple


```
> data <- read_table("file",...)
> data <- read_csv("file",...)
```
- ... correspondent à un ensemble d'options souvent très *importantes* car les fichiers de données contiennent *toujours des spécificités* (données manquantes, noms de variables...)
- Fichiers `.xls` : on pourra les *convertir* en `.csv` ou utiliser des packages spécifiques ou utiliser les fonctions `read_xls` ou `read_excel` du package `readxl`.

Indiquer le chemin

- Le **fichier des données** doit être placé dans le *répertoire de travail*. Sinon, il faut indiquer le *chemin* à `read.table`.
- *Exemple*: importer le fichier `data.csv` enregistré dans `~/lectureR/Part1` :
- Changement du répertoire de travail

```
> setwd("~/lectureR/Part1")
> df <- read_csv("data.csv",...)
```

- Spécification du chemin dans `read_csv`

```
> df <- read_csv("~/lecture_R/Part1/data.csv",...)
```

- Utilisation de la fonction `file.path`

```
> path <- file.path("~/lecture_R/Part1/", "data.csv")
> df <- read_csv(path,...)
```

Le package `readr`

- Il propose plusieurs fonctions à utiliser en fonction du *contexte* :
 - `read_delim` : permet de spécifier explicitement le *séparateur de champ* ;
 - `read_csv` : lorsque le séparateur est la *virgule* ;
 - `read_csv2` : lorsque le séparateur est le *point virgule* ;
 - `read_tsv` : lorsque le séparateur est la *tabulation* ;
 - `read_table`, `read_table2`... voir <https://readr.tidyverse.org>.

Quelques options importantes

Plusieurs *options importantes* sont proposées dans la fonction de `readr` :

- `col_names` : logique pour indiquer si le *nom des variables* est spécifié à la première ligne du fichier
- `na` : vecteur de caractères pour identifier les *données manquantes*
- `code_select` : spécifier les colonnes à lire
- `skip` : nombre de lignes à retirer avant de lire le fichier
- ...

Exemple

- Fichier *data_imp.txt*

```
name;size;age
John;174;32
Peter;?;28
Mary;165.5;NA
```

Caractéristiques

- 3 variables (ou plutôt 2...)
- Première ligne = nom des variables
- Données manquantes = NA, ?

Un premier essai

```
> path <- file.path("../data/", "data_imp.txt")
```

```
> tbl <- read_csv(path)
> tbl
# A tibble: 3 x 1
  `name;size;age`
  <chr>
1 John;174;32
2 Peter;?;28
3 Mary;165.5;NA
```

Problème

R lit trois lignes et *une* colonne ! On n'a *pas utilisé le bon délimiteur* !

Solution

- On choisit `read_delim` avec les bonnes options :

```
> tbl <- read_delim(path, delim=";", na=c("NA", "?"),
+                   locale = locale(decimal_mark = "."))
> tbl
# A tibble: 3 x 3
  name    size age
<chr> <dbl> <dbl>
1 John   174   32
2 Peter   NA   28
3 Mary  166.   NA
```

- On peut compléter en spécifiant les identifiants (on perd la classe `tibble` dans ce cas là) :

```
> (tbl1 <- column_to_rownames(tbl, var="name"))
      size age
John  174.0 32
Peter   NA 28
Mary  165.5 NA
```

Vérifier l'importation

- Cela peut s'effectuer avec les fonctions suivantes :

```
> summary(tbl)
      name              size              age
Length:3           Min.   :165.5   Min.   :28
Class :character    1st Qu.:167.6   1st Qu.:29
Mode  :character    Median :169.8   Median :30
                        Mean  :169.8   Mean  :30
                        3rd Qu.:171.9   3rd Qu.:31
                        Max.   :174.0   Max.   :32
                        NA's   :1       NA's   :1

> glimpse(tbl)
Rows: 3
Columns: 3
$ name <chr> "John", "Peter", "Mary"
$ size <dbl> 174.0, NA, 165.5
$ age <dbl> 32, 28, NA
```

```
> spec(tbl)
cols(
  name = col_character(),
  size = col_double(),
  age = col_double()
)
```

Remarque

Dans *Rstudio*, on peut lire des données avec `readr` en cliquant sur **Import Dataset** (pas toujours efficace pour des données complexes).

3.2 Annexe : les fonctions `read.table` et `read.csv`

Quelques options importantes

Il y a plusieurs *options importantes* dans `read.table` et `read.csv` :

- `sep` : le caractère de *séparation* (espace, virgule...)
- `dec` : le caractère pour le *séparateur décimal* (virgule, point...)
- `header` : logique pour indiquer si le *nom des variables* est spécifié à la première ligne du fichier
- `row.names` : vecteurs des *identifiants* (si besoin)
- `na.strings` : vecteur de caractères pour identifier les *données manquantes*.
- ...

Exemple

- Fichier *data_imp.txt*

```
name;size;age
John;174;32
Peter;?;28
Mary;165.5;NA
```

Caractéristiques

- 3 variables (ou plutôt 2...)
- Première ligne = nom des variables
- Données manquantes = NA, ?

Un premier essai

```
> path <- file.path("./data/", "data_imp.txt")
```

```
> df <- read.table(path)
> summary(df)
      V1
Length:4
Class  :character
Mode   :character
```

Problème

R lit quatre lignes et **une** colonne !

Solution

```
> df <- read.table(path,header=TRUE,sep=";",dec=".",
+                  na.strings = c("NA","?"),row.names = 1)
> df
      size age
John  174.0  32
Peter   NA  28
Mary  165.5  NA
> summary(df)
      size      age
Min.   :165.5  Min.   :28
1st Qu.:167.6  1st Qu.:29
Median :169.8  Median :30
Mean   :169.8  Mean   :30
3rd Qu.:171.9  3rd Qu.:31
Max.   :174.0  Max.   :32
NA's   :1      NA's   :1
```

3.3 Base de données avancées

- Les méthodes précédentes permettent de travailler avec des tables relativement *simples* :

- format tableau ;
- peu volumineuse.
- Les données étant de plus en plus *nombreuses et complexes*, il n'est *pas toujours possible* d'utiliser ces méthodes.

Exemple

- Données *trop volumineuses* impossible d'importer la base complète.
- *Autres formats* adaptés aux données complexes (JSON par exemple).

Le package DBI

- Interface de *communication* entre **R** et *différentes bases de données* de type SQL.
- Doc : <https://dbi.r-dbi.org>.
- Permet de se connecter à une base *sans la lire entièrement*.
- L'utilisateur pourra faire ses *requêtes* et importer les résultats.

Exemple

- Une base de données au format *SQLite* : *LEveloSTAR.sqlite3*.
- Connexion à la base :

```
> library(DBI)
> con <- dbConnect(RSQLite::SQLite(),
+                  dbname = "data/LEveloSTAR.sqlite3")
> dbListTables(con)
[1] "Etat"      "Topologie" "left"      "tbl_left"
```

- 4 tables
- On peut lire la table (enfin *juste en lire une partie...*) avec

```
> tbl(con, "Etat") |> select(1:5)
# Source:   SQL [?? x 5]
# Database: sqlite 3.43.2 [/Users/laurent/Google Drive/LAURENT/COURS/SNS/VISU/SLIDES/data/LEveloSTAR.sqlite]
   id nom          latitude longitude etat
   <int> <chr>      <dbl>    <dbl> <chr>
1     1 République  48.1      -1.68 En fonctionnement
2     2 Mairie     48.1      -1.68 En fonctionnement
3     3 Champ Jacquet 48.1      -1.68 En fonctionnement
4    10 Musée Beaux-Arts 48.1      -1.67 En fonctionnement
5    12 TNB        48.1      -1.67 En fonctionnement
6    14 Laënnec    48.1      -1.67 En fonctionnement
7    17 Charles de Gaulle 48.1      -1.68 En fonctionnement
8    20 Pont de Nantes 48.1      -1.68 En fonctionnement
9    22 Oberthur   48.1      -1.66 En fonctionnement
10   25 Office de Tourisme 48.1      -1.68 En fonctionnement
# i more rows
```

- Si on veut la récupérer pour faire des développements sur notre machine ou serveur, on utilise la fonction `collect`

```
> tbl(con, "Etat") |> collect() |> select(1:5) |> head()
# A tibble: 6 x 5
   id nom          latitude longitude etat
   <int> <chr>      <dbl>    <dbl> <chr>
1     1 République  48.1      -1.68 En fonctionnement
2     2 Mairie     48.1      -1.68 En fonctionnement
3     3 Champ Jacquet 48.1      -1.68 En fonctionnement
4    10 Musée Beaux-Arts 48.1      -1.67 En fonctionnement
5    12 TNB        48.1      -1.67 En fonctionnement
6    14 Laënnec    48.1      -1.67 En fonctionnement
```

- On n'oublie pas de *fermer la connexion*

```
> dbDisconnect(con)
```

API et JSON

- *JavaScript Object Notation*.
- Format proposé par de *nombreuses bases sur le web*.
- On peut fréquemment y accéder via une *interface de programmation applicative* (API).

Un exemple : le vélo star à Rennes

L'URL permettant d'obtenir les données est composée de 3 parties :

- nom de *domaine* : `https://data.rennesmetropole.fr/`
- chemin d'accès à l'*API* : `api/records/1.0/search/`
- la *requête*, elle-même composée de plusieurs parties :
 - jeu de données à utiliser : `?dataset=etat-des-stations-le-velo-star-en-temps-reel`
 - liste de facettes séparées par desesperluettes & : `&facet=nom&facet=etat&...`

Importation

```
> url <- paste0(  
+ "https://data.rennesmetropole.fr/api/records/1.0/search/",  
+ "?dataset=etat-des-stations-le-velo-star-en-temps-reel",  
+ "&q=&facet=nom",  
+ "&facet=etat",  
+ "&facet=nombreemplacementsactuels",  
+ "&facet=nombreemplacementsdisponibles",  
+ "&facet=nombrevelosdisponibles"  
+ )
```

```
> ll <- jsonlite::fromJSON(url)  
> tbl <- ll$records$fields |> as_tibble()  
> tbl |> select(3:5)  
# A tibble: 10 x 3  
  nom                               nombreemplacementsactuels idstation  
  <chr>                                <int> <chr>  
1 Sainte-Anne                        24 5505  
2 Saint-Georges Piscine              18 5509  
3 Musée Beaux-Arts                   15 5510  
4 Bonnets Rouges                    24 5514  
5 Charles de Gaulle                  24 5517  
6 Colombier                          24 5519  
7 Pont de Nantes                     20 5520  
8 Oberthur                           30 5522  
9 Auberge de Jeunesse                29 5537  
10 Croix Saint-Hélier                 20 5540
```

Autres outils importations

- `readxl` : fichier au format Excel.
- `sas7bdat` : importation depuis SAS.
- `foreign` : formats SPSS ou STATA
- `jsonlite` : format JSON
- `rvest` : webscrapping

3.4 Fusion de tables

Concaténer des données

- L'information utile pour une analyse provient (souvent) de *plusieurs tableaux de données*.
- Besoin de *correctement assembler ces tables* avant l'étude statistique.
- **Fonctions R standard** : `rbind`, `cbind`, `cbind.data.frame`, `merge`...
- **Fonctions R tidyverse** : `bind_rows`, `bind_cols`, `left_join`, `inner_join`.

Un exemple avec 2 tables

```
> df1
# A tibble: 4 x 2
  name  nation
<chr> <chr>
1 Peter  USA
2 Mary   GB
3 John   Aus
4 Linda  USA
> df2
# A tibble: 3 x 2
  name  age
<chr> <dbl>
1 John   35
2 Mary   41
3 Fred   28
```

Objectif

Un **tableau** de données avec **3 colonnes** : name, nation et age.

bind_rows

```
> bind_rows(df1,df2)
# A tibble: 7 x 3
  name nation age
<chr> <chr> <dbl>
1 Peter USA    NA
2 Mary GB      NA
3 John Aus     NA
4 Linda USA    NA
5 John <NA>     35
6 Mary <NA>     41
7 Fred <NA>     28
```

Mauvais choix ici (2 lignes pour certains individus).

full_join

```
> full_join(df1,df2)
# A tibble: 5 x 3
  name nation age
<chr> <chr> <dbl>
1 Peter USA    NA
2 Mary GB      41
3 John Aus     35
4 Linda USA    NA
5 Fred <NA>     28
```

tous les individus sont conservés (NA sont ajoutés pour les quantités non mesurées.)

left_join

```
> left_join(df1,df2)
# A tibble: 4 x 3
  name nation age
<chr> <chr> <dbl>
1 Peter USA    NA
```

2	Mary	GB	41
3	John	Aus	35
4	Linda	USA	NA

seuls les individus du *premier tableau (gauche)* sont conservés.

inner_join

```
> inner_join(df1,df2)
# A tibble: 2 x 3
  name nation age
<chr> <chr> <dbl>
1 Mary   GB    41
2 John   Aus    35
```

on garde les individus pour lesquels **nation** et **age** sont mesurés.

Conclusion

- Plusieurs possibilités pour assembler des données.
- Important de faire le bon choix en fonction du contexte.

3.5 Manipuler les données avec Dplyr

- **dplyr** est un package du **tidyverse** efficace pour *transformer et résumer* des tableaux de données.
- Il propose une **syntaxe claire** (basée sur une *grammaire*) permettant de manipuler les données.
- Par exemple, pour calculer le moyenne de **Sepal.Length** de l'espèce **setosa**, on utilise généralement

```
> mean(iris[iris$Species=="setosa",]$Sepal.Length)
[1] 5.006
```

- La même chose en **dplyr** s'obtient avec

```
> iris |> filter(Species=="setosa") |>
+ summarise(Moy_SL=mean(Sepal.Length))
Moy_SL
1 5.006
```

Grammaire dplyr

dplyr propose une *grammaire* dont les principaux *verbes* sont :

- `select()` : sélectionner des colonnes (variables)
- `filter()` : filtrer des lignes (individus)
- `arrange()` : ordonner des lignes
- `mutate()` : créer des nouvelles colonnes (nouvelles variables)
- `summarise()` : calculer des résumés numériques (ou résumés statistiques)
- `group_by()` : effectuer des opérations pour des groupes d'individus

Penser à consulter la *cheat sheet*.

Select

But

Sélectionner des *variables*.

```
> df <- select(iris,Sepal.Length,Petal.Length)
> head(df)
  Sepal.Length Petal.Length
1          5.1           1.4
2          4.9           1.4
3          4.7           1.3
4          4.6           1.5
5          5.0           1.4
6          5.4           1.7
```

Filter

But

Filtrer des *individus*.

```
> df <- filter(iris,Species=="versicolor")
> head(df)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          7.0         3.2         4.7         1.4 versicolor
2          6.4         3.2         4.5         1.5 versicolor
3          6.9         3.1         4.9         1.5 versicolor
4          5.5         2.3         4.0         1.3 versicolor
5          6.5         2.8         4.6         1.5 versicolor
6          5.7         2.8         4.5         1.3 versicolor
```

Arrange

But

Ordonner des **individus** en fonction d'une variable.

```
> df <- arrange(iris,Sepal.Length)
> head(df)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          4.3         3.0         1.1         0.1 setosa
2          4.4         2.9         1.4         0.2 setosa
3          4.4         3.0         1.3         0.2 setosa
4          4.4         3.2         1.3         0.2 setosa
5          4.5         2.3         1.3         0.3 setosa
6          4.6         3.1         1.5         0.2 setosa
```

Mutate

But

Définir des **nouvelles variables** dans le jeu de données.

```
> df <- mutate(iris,diff_petal=Petal.Length-Petal.Width)
> head(select(df,Petal.Length,Petal.Width,diff_petal))
  Petal.Length Petal.Width diff_petal
1          1.4         0.2         1.2
2          1.4         0.2         1.2
3          1.3         0.2         1.1
4          1.5         0.2         1.3
5          1.4         0.2         1.2
6          1.7         0.4         1.3
```


Summarise

But

Calculer des **résumés statistiques**.

```
> summarise(iris,mean=mean(Petal.Length),var=var(Petal.Length))
  mean      var
1 3.758 3.116278
```

Summarise_all et summarise_at

On peut également calculer des résumés pour des groupes de variables :

- `summarize_all` : **toutes les variables** du tibble

```
> iris1 <- select(iris,-Species)
> summarise_all(iris1,mean)
  Sepal.Length Sepal.Width Petal.Length Petal.Width
1      5.843333      3.057333      3.758      1.199333
```

- `summarize_at` : **choisir les variables** du tibble

```
> summarise_at(iris,1:3,mean)
  Sepal.Length Sepal.Width Petal.Length
1      5.843333      3.057333      3.758
```

group_by

But

Faire des opérations pour des **groupes de données**.

```
> summarise(group_by(iris,Species),mean(Petal.Length))
# A tibble: 3 x 2
  Species      `mean(Petal.Length)`
  <fct>          <dbl>
1 setosa          1.46
2 versicolor      4.26
3 virginica        5.55
```

L'opérateur pipe |>

- L'opérateur de *chaînage* ou *pipe* |> permet d'*enchaîner les commandes* pour une syntaxe plus claire.

- Par exemple,

```
> mean(iris[iris$Species=="setosa", "Sepal.Length"])  
[1] 5.006
```

ou (un peu plus lisible)

```
> df1 <- iris[iris$Species=="setosa",]  
> df2 <- df1$Sepal.Length  
> mean(df2)  
[1] 5.006
```

- ou (encore un peu plus lisible avec **dplyr**)

```
> df1 <- filter(iris, Species=="setosa")  
> df2 <- select(df1, Sepal.Length)  
> summarize(df2, mean(Sepal.Length))  
  mean(Sepal.Length)  
1           5.006
```

Pas satisfaisant

Création de deux objets **dataframe** (inutiles) pour un calcul "simple".

- Avec le *pipe*, on *décompose* et *enchaîne* les opérations:

1. Les données

```
> iris
```

2. On filtre les individus **setosa**

```
> iris |> filter(Species=="setosa")
```

3. On garde la variable d'intérêt

```
> iris |> filter(Species=="setosa") |> select(Sepal.Length)
```

4. On calcule la moyenne

```
> iris |> filter(Species=="setosa") |>
+   select(Sepal.Length) |> summarize_all(mean)
  Sepal.Length
1          5.006
```

Plus généralement

- L'opérateur pipe `|>` applique l'*objet de droite* en considérant que le premier argument est l'*objet de gauche* (non symétrique).

```
> X <- as.numeric(c(1:10,"NA"))
> mean(X,na.rm = TRUE)
[1] 5.5
```

ou, de façon équivalente,

```
> X |> mean(na.rm=TRUE)
[1] 5.5
```

3.6 Quelques fonctions utiles de tidyr

Le package tidyr

- Il propose un ensemble de fonctions qui aident à obtenir des données (*tibble*) propres.
- Souvent utile avec *dplyr* pour manipuler les données et *ggplot* pour les visualiser.

Reformater les données

- Certaines analyses statistiques nécessitent un *format particulier* pour les données.
- Un exemple jouet

```
> df <- iris |> group_by(Species) |> summarize_all(mean)
> head(df)
# A tibble: 3 x 5
  Species    Sepal.Length Sepal.Width Petal.Length Petal.Width
<fct>      <dbl>         <dbl>         <dbl>         <dbl>
1 setosa      5.01          3.43          1.46          0.246
2 versicolor  5.94          2.77          4.26          1.33
3 virginica   6.59          2.97          5.55          2.03
```

pivot_longer

- **Assembler** des colonnes en lignes avec `pivot_longer` (anciennement `gather`) :

```
> df1 <- df |> pivot_longer(-Species,names_to="variable",
+                           values_to="valeur")
> head(df1)
# A tibble: 6 x 3
  Species    variable    valeur
<fct>      <chr>         <dbl>
1 setosa   Sepal.Length  5.01
2 setosa   Sepal.Width    3.43
3 setosa   Petal.Length    1.46
4 setosa   Petal.Width     0.246
5 versicolor Sepal.Length  5.94
6 versicolor Sepal.Width    2.77
```

Remarque

Même information avec un **format long**.

pivot_wider

- Décomposer une ligne en plusieurs colonnes avec `pivot_wider` (anciennement `spread`).

```
> df1 |> pivot_wider(names_from=variable,values_from=valeur)
# A tibble: 3 x 5
  Species    Sepal.Length Sepal.Width Petal.Length Petal.Width
<fct>      <dbl>         <dbl>         <dbl>         <dbl>
1 setosa      5.01          3.43          1.46          0.246
2 versicolor  5.94          2.77          4.26          1.33
3 virginica   6.59          2.97          5.55          2.03
```

Separer une colonne en plusieurs

- Fonctions `separate_wider_delim`, `separate_wider_position` et `separate_wider_regex`.
Par exemple

```
> (df <- tibble(date=as.Date(c("01/03/2015", "05/18/2017",  
+ "09/14/2018"), "%m/%d/%Y"), temp=c(18, 21, 15)))  
# A tibble: 3 x 2  
  date      temp  
  <date>    <dbl>  
1 2015-01-03    18  
2 2017-05-18    21  
3 2018-09-14    15  
  
> (df1 <- df |> separate_wider_delim(date, delim="-",  
+                                   names=c("year", "month", "day")))  
# A tibble: 3 x 4  
  year month day    temp  
  <chr> <chr> <chr>  <dbl>  
1 2015   01   03      18  
2 2017   05   18      21  
3 2018   09   14      15
```

Assembler des colonnes

- `unite` permet de faire l'opération inverse :

```
> df1 |> unite(date, year, month, day, sep="/") |>  
+ mutate(date1=lubridate::as_date(date))  
# A tibble: 3 x 3  
  date      temp date1  
  <chr>    <dbl> <date>  
1 2015/01/03    18 2015-01-03  
2 2017/05/18    21 2017-05-18  
3 2018/09/14    15 2018-09-14
```

4 Bases de données

4.1 SQL : Structured Query Language

Le package DBI

- SQL* est un *langage* commun permettant de *commander de nombreuses bases de données*.

- Utilisé par les bases de données les plus populaires comme

Base	Package
MySQL	RMySQL
MariaDB	RMariaDB
Postgres	RPostgres
SQLite	RSQLite

- La package **DBI** (DataBase Interface) offre une *interface de communication* entre **R** et différentes bases de données de type SQL à l'aide de pilotes dédiés : <https://dbi.r-dbi.org>

Bases de données relationnelles

Les bases de données de type SQL utilisent le *paradigme individus/variables* :

- les *bases* contiennent des *tables* (équivalentes aux tibbles) ;
- les *tables* contiennent des *colonnes* (ou champs) qui regroupent des informations de même type ;
- les enregistrements ou entrées d'une tables correspondent aux lignes de cette table.

Les tables sont reliées entre elles grâce à des *identifiants* (clés primaires/clés étrangères).

La fonction **dbConnect**

```
> con <- dbConnect(
+   RPostgres::Postgres(),
+   dbname = "DATABASE_NAME",
+   host = "HOST",
+   port = 5432,
+   user = "USERNAME",
+   password = "PASSWORD")
```

Remarques

- La fonction **RPostgres::Postgres()** qui fournit un pilote (ou *driver*) pour la base de données voulue.
- On peut remplacer cette fonction par **RMariaDB::MariaDB()** pour se connecter à une base de données MariaDB.

SQLite

- Base de données qui n'est pas basée sur le principe *client/serveur*.
- Permet de travailler sur des bases de *données stockées dans des fichiers*, voire directement en *mémoire vive*. C'est donc très simple à mettre en uvre.
- **Exemple** : ouverture d'une connexion vers une base de données SQLite contenue en mémoire vive

```
> library(DBI)
> con <- dbConnect(RSQLite::SQLite(), dbname = ":memory:")
```

- À ce stade aucune table :

```
> dbListTables(con)
character(0)
```

4.2 Peupler une base de données

```
> set.seed(1234)
> tbl1 <- tibble(ID=sample(LETTERS),age=sample(1:100,26))
> tbl2 <- tibble(ID=sample(LETTERS),taille=sample(160:180,26,replace = TRUE))
>
> df <- data.frame(
+   x = runif(25),
+   label = sample(c("A", "B"), size = 25, replace = TRUE)
+ )
> dbWriteTable(con, name = "table1",value = tbl1)
> dbWriteTable(con, name = "table2",value = tbl2)
> dbListTables(con)
[1] "table1" "table2"
```

création de *deux tables* dans la *base*.

4.3 Requêtes SQL

SQL en bref !

```
> SELECT j FROM df WHERE i GROUP BY by
```

Remarque

Ce pseudo code rappelle les pseudo-codes suivants :

- `data-table` :

```
> dt[i, j, by]
```

- `dplyr` :

```
> df |> group_by(by) |> filter(i) |> select(j)
```

En effet SQL et la proximité entre Bdd et data-frame a inspiré les créateurs des deux packages.

Exemple 1

- `dbSendQuery` : soumettre la requête

```
> req1 <- dbSendQuery(con,"SELECT * FROM table1 WHERE age>85")
> req1
<SQLiteResult>
SQL SELECT * FROM table1 WHERE age>85
ROWS Fetched: 0 [incomplete]
Changed: 0
```

- `dbFetch` : collecter (ramener vers **R**) les données

```
> res1 <- dbFetch(req1)
> class(res1)
[1] "data.frame"
> res1
  ID age
1  F  87
2  M  96
```

- La requête est maintenant *complète* :

```
> req1
<SQLiteResult>
SQL SELECT * FROM table1 WHERE age>85
ROWS Fetched: 2 [complete]
Changed: 0
```


- `dbGetQuery` : soumettre et exécuter directement la requête

```
> dbGetQuery(con, "SELECT * FROM table1 WHERE age>85")
  ID age
1  F  87
2  M  96
```

Exemple 2 : jointure

```
> res <- dbGetQuery(con, "
+           SELECT *
+           FROM table1
+           INNER JOIN table2 ON table1.id = table2.id
+           ORDER BY ID
+           ")
> head(res)
  ID age ID taille
1  A  26 A   180
2  B  32 B   165
3  C  80 C   167
4  D  72 D   165
5  E  47 E   180
6  F  87 F   161
```

Quelques fonctions supplémentaires

- `dbExistsTable(con,name)` vérifier si la table `name` existe pour la connexion `con`.
- `dbRemoveTable(con,name,...)` effacer la table `name` de la connexion `con`.
- `dbGetRowsAffected(req,...)` nombre de lignes affectés (extraction, effacement, modification) par la requête `req`.
- `dbGetRowCount(req,...)` nombre de lignes collectées lors de la requête `req`.

4.4 SQL et dplyr

- `dplyr` permet de travailler sur des objets *plus variés* que les data-frames ou tibbles.
- *Compatible* avec les *bases SQL*.
- Nécessite l'installation du package `dbplyr`.
- Documentation : <https://dbplyr.tidyverse.org/index.html>.

- Création d'un objet avec lequel **dplyr** va travailler avec la fonction **con**:

```
> dbListTables(con)
[1] "table1" "table2"
> T1 <- tbl(con, "table1")
> T2 <- tbl(con, "table2")
> class(T1)
[1] "tbl_SQLiteConnection" "tbl_dbi"          "tbl_sql"
[4] "tbl_lazy"            "tbl"
```

- On peut ensuite travailler (ou requêter) sur les tables avec les commandes **dplyr** standards:

```
> req <- T1 |> filter(age<=40)
```

Vrai tibble ou table distante ?

```
> req
# Source:   SQL [9 x 2]
# Database: sqlite 3.43.2 [:memory:]
  ID      age
<chr> <int>
1 P         5
2 L        40
3 X         3
4 B        32
5 J         2
6 T         6
7 Y         8
8 A        26
9 R        17
```

Attention

- L'objet, de type **tbl_lazy**, ressemble à une table mais son nombre de lignes est **inconnu** la requête est juste **préparée**.
- L'envoi vers la base de données est **retardé le plus possible** afin de minimiser le nombre d'accès à la base.
- La récupération des données sous **R** se fait avec la fonction **collect**.

```
> dim(req)
[1] NA 2
> T11 <- req |> collect();class(T11)
[1] "tbl_df"      "tbl"          "data.frame"
> dim(T11)
[1] 9 2
```

- Visualisation de la requête SQL :

```
> show_query(req)
<SQL>
SELECT `table1`.*
FROM `table1`
WHERE (`age` <= 40.0)
```

Autre exemple : jointure

- La requête :

```
> req2 <- inner_join(T1,T2,by=join_by("ID"))
> dim(req2)
[1] NA 3
> show_query(req2)
<SQL>
SELECT `table1`.*, `taille`
FROM `table1`
INNER JOIN `table2`
  ON (`table1`.`ID` = `table2`.`ID`)
```

- Rapatriement des données sous **R**

```
> T12 <- req2 |> collect()
> dim(T12)
[1] 26 3
```

```
> req3 <- T1 |> filter(age==min(age))
> req4 <- T1 |> mutate(paste(ID,age))
> req5 <- T1 |> summarize(min(age))
```

```
> dbDisconnect(con)
```

4.5 JSON : JavaScript Object Notation

Des données variées

Les données ne sont *pas* toujours stockées dans *des formats tabulaires*. Il existe bien d'autres façons de les conserver et d'y accéder :

- Bases de données relationnelles/SQL ;
- Fichiers structurés : XML, YAML ou JSON ;
- Bases de données NoSQL ;
- WEB

Le format JSON

- Alternative aux bases de données relationnelles.
- *Idée* : encoder les informations dans des *fichiers textes structurés*.

Très utilisé

- *MongoDB* : base de données NoSQL.
- *API WEB* : interface de programmation applicative.
- *Open Data* : format de référence dans les bases de données publiques de l'État et des administrations, voir <https://www.data.gouv.fr/fr/>.

Fichier JSON

Objectifs

Lisible par des humains et des machines.

Syntaxe

Simple avec un petit nombre de types de données :

- Deux types **structurés** ;
- Plusieurs types **simple**.

Types structurés - les objets JSON

- Proches des *dictionnaires* **Python** ou des *listes* de **R**.
- Syntaxe :

```
{
  "clé1": valeur1,
  "clé2": valeur2,
  ...
}
```

- Principe **clé/valeur** avec des `"` autour des clés.

Types structurés - les tableaux

- Proches des *listes* **Python** ou des *vecteurs* de **R**.

```
[
  valeur1,
  valeur2,
  ...
]
```

- Les listes permettent simplement de structurer des données de **façon ordonnée**.

Types simples (1)

- *Chaîne de caractères* : elle est entre `"..."` :

```
{
  "prénom": "Jean-Sébastien",
  "nom": "Bach"
}
```

- *Nombre* : on l'écrit directement :

```
{
  "prénom": "Jean-Sébastien",
  "nom": "Bach",
  "nombre_enfants": 20
}
```

Types simples (2)

- `true` ou `false` : il faut noter (surtout quand on utilise **R**) que les valeurs booléennes sont écrites en minuscules :

```
{
  "prénom": "Jean-Sébastien",
  "nom": "Bach",
  "compositeur": true
}
```

- `null` : si l'on ne souhaite pas donner de valeur à une clé.

```
{
  "prénom": "Jean-Sébastien",
  "particule": null,
  "nom": "Bach"
}
```

Imbrication

- Pour les types structurés, les différentes valeurs peuvent être de n'importe quel type, y compris un autre type structuré.
- Pour vérifier la validité d'un fichier JSON, on peut visiter [ce site](#).

```
{
  "prénom": "Jean-Sébastien",
  "nom": "Bach",
  "épouses": [
    {
      "prénom": ["Maria", "Barbara"],
      "nom": "Bach"
    },
    {
      "prénom": ["Anna", "Magdalena"],
      "nom": "Wilcke"
    }
  ]
}
```

Le package jsonlite

Principales fonctions :

- `fromJSON` : import de fichier JSON ;
- `toJSON` : export au format JSON ;

- `stream_in` : import pour du JSON “en lignes” ;
- `stream_out` : export au format JSON “en lignes”.

Import et jeu de données jouet

```
> library(jsonlite)
> df <- tibble(x = c(0, pi), y = cos(x))
> toJSON(df);class(df)
[{"x":0,"y":1},{x":3.1416,"y":-1}]
[1] "tbl_df"      "tbl"        "data.frame"
```

Remarques

- Le résultat est une chaîne de caractère !
- Il y a un tableau JSON ;
- Paradigme individus/variables respectés.
- Perte de précision. . .

JSON et data-frame (2)

```
> df |> toJSON() |> fromJSON()
      x y
1 0.0000 1
2 3.1416 -1
```

La fonction `toJSON` prend une chaîne de caractère représentant un fichier JSON bien formaté en l’importe *si possible* sous la forme d’un data-frame

Remarques

- Même structure ;
- Confirmation de la perte de précision.

JSON et data-frame (3)

```
> fromJSON(' [{"x":1},{ "y":2}] ')
  x y
1 1 NA
2 NA 2
```

Remarques

- La situation est plus délicate ;
- On utilise deux **quotes** différentes...

JSON et data-frame (4)

```
> df1 <- fromJSON(' [{"x":[1, 2, 3]},{ "x":4}] ')
> class(df1)
[1] "data.frame"
> glimpse(df1)
Rows: 2
Columns: 1
$ x <list> <1, 2, 3>, 4
```

Un data-frame... inhabituel.

```
> df1
      x
1 1, 2, 3
2      4
> df1$x
[[1]]
[1] 1 2 3

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

JSON et data-frame (5)

- Par défaut, la fonction **fromJSON** tente de **simplifier** les vecteurs.
- On peut forcer le comportement inverse :


```

> fromJSON('{"x":[1, 2, 3]},{"x":4}', simplifyVector = FALSE)
[[1]]
[[1]]$x
[[1]]$x[[1]]
[1] 1

[[1]]$x[[2]]
[1] 2

[[1]]$x[[3]]
[1] 3

[[2]]
[[2]]$x
[1] 4

```

JSON et data-frame (6)

Lequel choisir ?

```

> fromJSON('{"x":{"xa":1, "xb":2}},{"x":3} ')
      x
1 1, 2
2   3

```

```

> fromJSON('{"x":{"xa":1, "xb":2}},{"x":3}', simplifyDataFrame = FALSE)
[[1]]
[[1]]$x
[[1]]$x$xa
[1] 1

[[1]]$x$xb
[1] 2

[[2]]
[[2]]$x
[1] 3

```

4.6 API WEB

Interface de programmation applicative

- De nombreux sites proposent d'accéder à leurs données via des *requêtes http*.
- Elles renvoient des *réponses* aux formats XML, HTML... et JSON

L'exemple du vélo star

- Site <https://data.rennesmetropole.fr/>
- Table sur l'état des stations velib.

La requête

Elle prend une forme spécifique :

- le nom de domaine : `https://data.rennesmetropole.fr/` ;
- le chemin d'accès à l'API : `api/explore/v2.1/catalog/` ;
- l'identifiant du jeu de données : `datasets/etat-des-stations-le-velo-star-en-temps-reel/` ;
- on peut ajouter des verbes comme dans *dplyr*, par exemple `select=id%2C%20nom&limit=20`

Comment utiliser ça dans **R** : *jsonlite* !

En avant !

```
> url <- str_c(  
+   "https://data.rennesmetropole.fr/",  
+   "api/explore/v2.1/catalog/",  
+   "datasets/etat-des-stations-le-velo-star-en-temps-reel/",  
+   "records"  
+ )  
> ll <- fromJSON(url)
```

Finalisation

```

> df <- ll$results
> dim(df)
[1] 10 8
> df |> select(nom,coordonnees,nombrevelosdisponibles)
      nom coordonnees.lon coordonnees.lat nombrevelosdisponibles
1 Sainte-Anne      -1.680461      48.11421             16
2 Saint-Georges Piscine -1.674417      48.11239             12
3 Musée Beaux-Arts     -1.674080      48.10960             13
4 Bonnets Rouges      -1.665814      48.10685             16
5 Charles de Gaulle    -1.677119      48.10511             12
6 Colombier           -1.680594      48.10610             13
7 Pont de Nantes       -1.684015      48.10202              7
8 Oberthur            -1.661853      48.11355             11
9 Auberge de Jeunesse  -1.681876      48.12088             13
10 Croix Saint-Hélief -1.662090      48.10305              6

```

Le tour est joué !

Le format ndjson

- Pour *Newline Delimited JSON* !
- Très souvent les API, ou les requêtes MongoDB, *renvoient des fichiers JSON non valides*. Ces fichiers ont en fait la structure typique suivante :

```

{"x":1,"y":2}
{"x":3,"y":4}
...

```

- **Chaque ligne** est un fichier JSON valide qu'on peut lire à l'aide de la fonction `stream_in`.

stream_in

```

> url <- "http://jeroen.github.io/data/diamonds.json"
> diamonds <- jsonlite::stream_in(url(url))

```

```

> diamonds |> select(1:5) |> head()
  carat      cut color clarity depth
1  0.23    Ideal     E    SI2   61.5
2  0.21  Premium     E    SI1   59.8
3  0.23     Good     E    VS1   56.9
4  0.29  Premium     I    VS2   62.4
5  0.31     Good     J    SI2   63.3
6  0.24 Very Good     J   VVS2   62.8

```

5 Programmer en R

5.1 Structures de contrôle

Boucles for

- *Syntaxe :*

```
> for (i in vecteur){  
+   expr1  
+   expr2  
+   ...  
+ }
```

- *Exemple :*

```
> for (i in 1:3){print(i)}  
[1] 1  
[1] 2  
[1] 3  
> for (i in c("lundi","mardi","mercredi")){print(i)}  
[1] "lundi"  
[1] "mardi"  
[1] "mercredi"
```

Condition while

- *Syntaxe :*

```
> while (condition) {expression}
```

- *Exemple :*

```
> i <- 1  
> while (i<=3) {  
+   print(i)  
+   i <- i+1  
+ }  
[1] 1  
[1] 2  
[1] 3
```

Condition if else

- *Syntaxe :*

```

> if (condition){
+   expr1
+   ...
+ } else {
+   expr2
+   ...
+ }

```

- *Exemple :*

```

> a <- -2
> if (a>0){
+   a <- a+1
+ } else {
+   a <- a-1
+ }
> print(a)
[1] -3

```

switch

- *Syntaxe :*

```

> switch(expression,
+   "cond1" = action1,
+   "cond2" = action2,
+   ...)

```

- *Exemple :*

```

> X <- matrix(0,nrow = 5,ncol = 5)
> switch(class(X)[1],
+   "matrix"=print("X est une matrice"),
+   "data.frame"=print("X est un data.frame"),
+   "numeric"=print("X est de classe numérique"))
[1] "X est une matrice"

```

Écrire une fonction

- *Syntaxe :*

```

> mafonct <- function(param1,param2,...){
+   expr1
+   expr2
+   return(...)
+ }

```

- *Exemple :*

```
> factorielle <- function(n){
+   return(prod(1:n))
+ }
> factorielle(5)
[1] 120
```

Améliorer sa fonction

- Ajout de `stop` et `warning`

```
> factorielle <- function(n){
+   if (n<=0) stop("l'entier doit être strictement positif")
+   if (ceiling(n)!=n) warning(paste("arrondi de",n,"en",ceiling(n)))
+   return(prod(1:ceiling(n)))
+ }
```

- *Test :*

```
> factorielle(-2)
Error in factorielle(-2): l'entier doit être strictement positif

> factorielle(5.8)
Warning in factorielle(5.8): arrondi de 5.8 en 6
[1] 720
> factorielle(5)
[1] 120
```

5.2 Les fonctions map

- Fonctions du package `purrr` du `tidyverse` qui permettent d'appliquer des *fonctions* à des *listes*, et donc notamment à des colonnes de *tibble*.
- Version améliorée des fonction `apply`.
- Exemple

```
> set.seed(1234)
> tbl <- tibble(age=runif(5,20,50),taille=runif(5,150,180))
```

Fonctions `map`

- Appliquer *une fonction à des colonnes* :

```
> tbl |> map(mean)
$age
[1] 36.97743

$taille
[1] 162.3762
```

- Renvoyer un *vecteur* :

```
> tbl |> map_dbl(mean)
      age      taille
1 36.97743 162.37615
```

Autres fonctions

`map_int`, `map_chr`, `map_lgl`, `map_dfc`...

Fonctions `map2`

- Pour appliquer des fonctions à des *paires* d'éléments de *listes* :

```
> tbl2 <- tibble(age=runif(5,20,50),taille=runif(5,150,180))
> map2_dfc(tbl,tbl2,function(d1,d2) mean(rbind(d1,d2)))
# A tibble: 1 x 2
   age taille
<dbl> <dbl>
1  36.7  162.
```

Autres fonctions

`map2_int`, `map2_chr`, `map2_lgl`, `map2_dfc`...

Les fonctions anonymes

- Permettent de faciliter la syntaxe. Peut se faire avec une *formule* :

```
> map2_dfc(tbl,tbl2,-mean(rbind(.x,.y)))
# A tibble: 1 x 2
   age taille
<dbl> <dbl>
1  36.7  162.
```

- Ou de la façon suivante pour mieux **expliquer les arguments** :

```
> map2_dfc(tbl, tbl2, \(a, b) mean(rbind(a, b)))
# A tibble: 1 x 2
  age taille
<dbl> <dbl>
1  36.7  162.
```

Pipes et fonctions anonymes

- Les pipes `|>` (de la distribution de **R**) et `%>%` de **dplyr** sont quasi similaires
- On note une *différence* lorsqu'on les utilise avec des **fonctions anonymes** :

```
> tbl %>% .[1,1]
# A tibble: 1 x 1
  age
<dbl>
1  23.4
```

```
> tbl |> .[1,1]
Error: function '[' not supported in RHS call of a pipe (<text>:1:8)
```

- On peut corriger en *spécifiant les paramètres* de la **fonction anonyme** :

```
> tbl |> (\(x) x[1,1])()
# A tibble: 1 x 1
  age
<dbl>
1  23.4
```

6 Visualiser des données

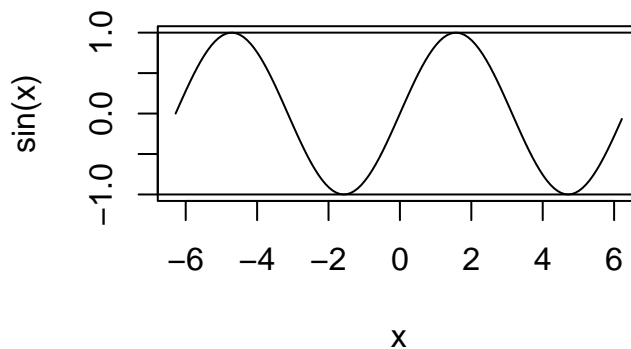
6.1 Graphes conventionnels

- *Visualisation* : cruciale à **toutes les étapes** d'une étude statistique.
- **R** Permet de créer un **très grand nombre** de type de graphes.
- On propose une (courte) présentation des *graphes classiques*,
- suivie par les graphes **ggplot2**.

La fonction `plot`

- Fonction *générique* pour représenter (presque) *tous les types de données*.
- Pour un *nuage de points*, il suffit de renseigner un *vecteur* pour l'axe des **x**, et un autre vecteur pour celui des **y**.

```
> x <- seq(-2*pi, 2*pi, by=0.1)
> plot(x, sin(x), type="l", xlab="x", ylab="sin(x)")
> abline(h=c(-1,1))
```



Graphes classiques pour visualiser des variables

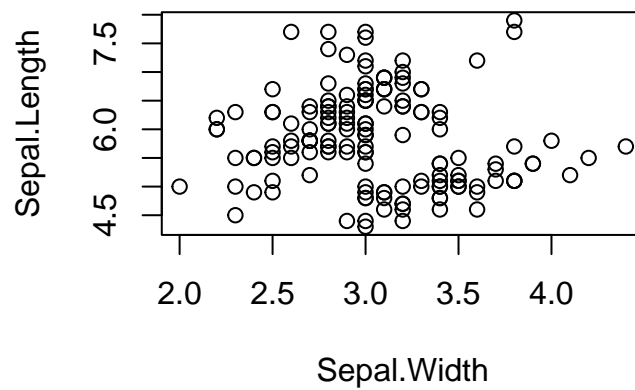
- *Histogramme* pour une variable *continue*, *diagramme en barre* pour une variable *qualitative*.
- *Nuage de points* pour 2 variables continues.
- *Boxplot* pour une distribution continue.

Constat (positif)

Il existe une *fonction R* pour toutes les représentations.

Nuage de points sur un jeu de données

```
> plot(Sepal.Length~Sepal.Width, data=iris)
```



```
> #pareil que
> plot(iris$Sepal.Width,iris$Sepal.Length)
```

Histogramme (variable continue)

```
> hist(iris$Sepal.Length,col="red")
```

Histogram of iris\$Sepal.Length

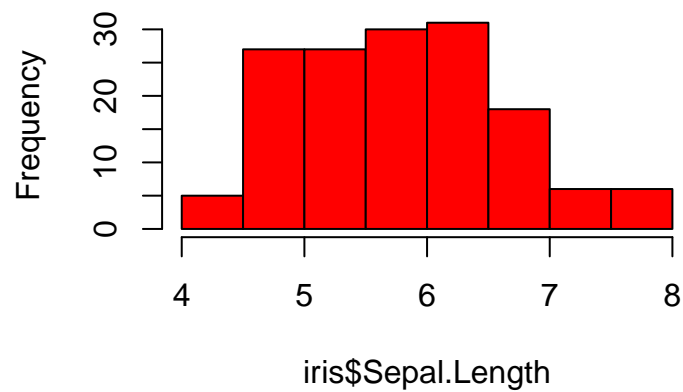
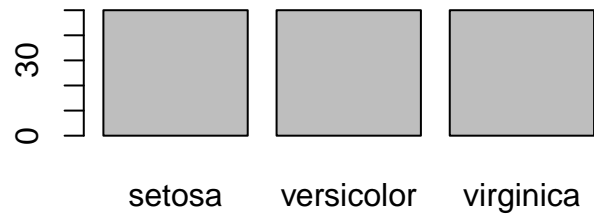


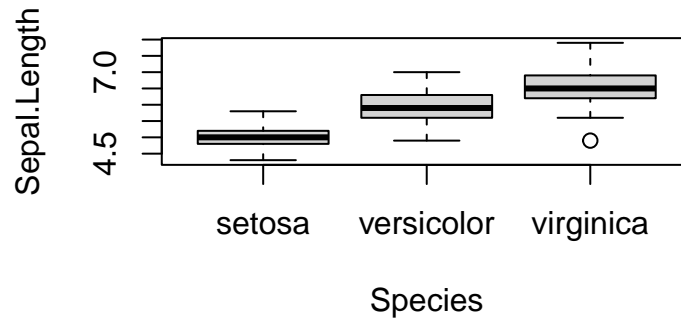
Diagramme en barres (variable qualitative)

```
> barplot(table(iris$Species))
```



Boxplot (distribution)

```
> boxplot(Sepal.Length~Species,data=iris)
```



6.2 Visualisation avec ggplot2

- `ggplot2` permet de faire des graphes **R** en s'appuyant sur une **grammaire des graphiques** (équivalent de `dplyr` pour manipuler les données).
- Les graphes produits sont *de très bonnes qualités* (pas toujours le cas avec les graphes conventionnels).
- La **grammaire** `ggplot2` permet d'obtenir des **graphes "complexes"** avec une **syntaxe claire et lisible**.

Remarque

Aujourd'hui la plupart des **graphes statiques** faits dans les tutoriels, livres, applications... sont faits avec `ggplot2`.

Assembler des couches

Pour un tableau de données fixé, un graphe est défini comme une succession de **couches**. Il faut toujours spécifier :

- les *données*
- les *variables* à représenter
- le *type de représentation* (nuage de points, boxplot...).

Les graphes ggplot sont construits à partir de ces couches. On indique

- les données avec `ggplot`
- les variables avec `aes` (aesthetics)
- le type de représentation avec `geom_`

La grammaire

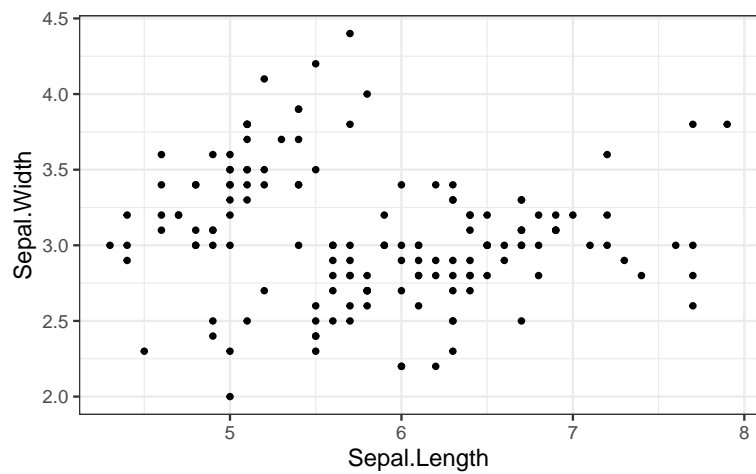
Les principaux *verbes* sont

- **Data** (`ggplot`) : les *données*, un `dataframe` ou un `tibble`.
- **Aesthetics** (`aes`) : façon dont les *variables* doivent être représentées.
- **Geometrics** (`geom_...`) : *type* de représentation.
- **Statistics** (`stat_...`) : spécifier les *transformations* des données.
- **Scales** (`scale_...`) : modifier certains *paramètres du graphe* (changer de couleurs, de taille...).

Tous ces éléments sont **séparés par un +**.

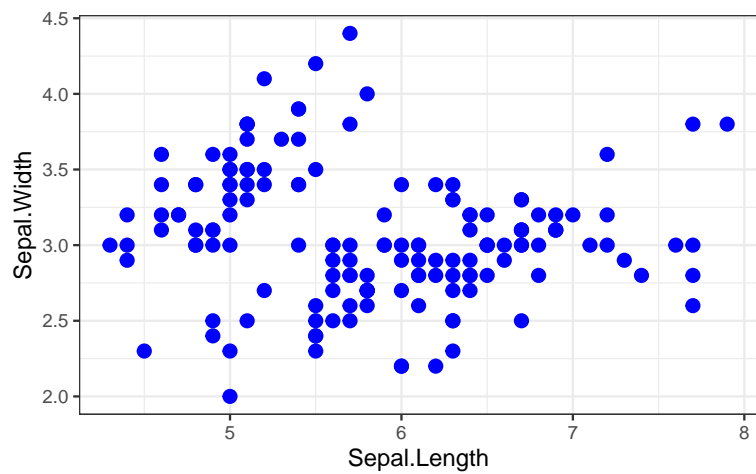
Un premier exemple

```
> ggplot(iris)+aes(x=Sepal.Length,y=Sepal.Width)+geom_point()
```



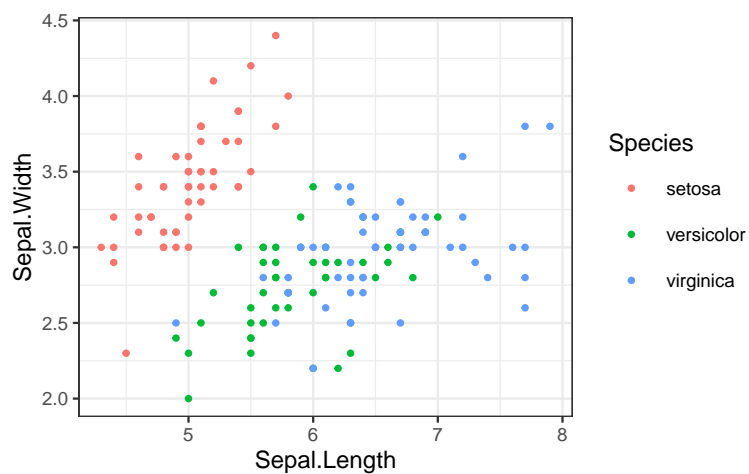
Couleur et taille

```
> ggplot(iris)+aes(x=Sepal.Length,y=Sepal.Width)+
+   geom_point(color="blue",size=2)
```



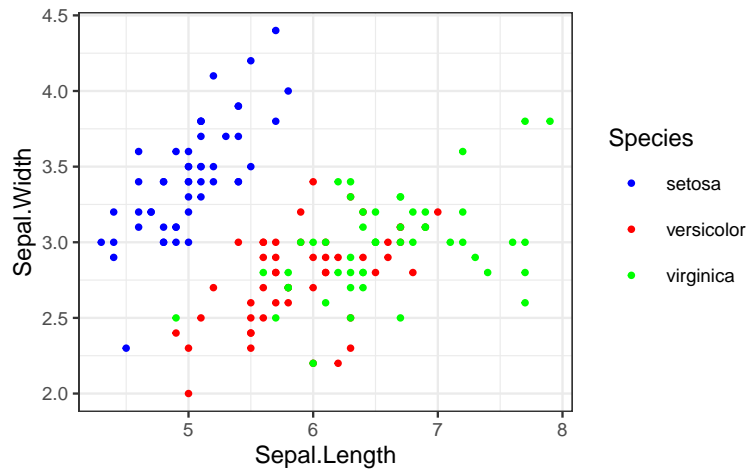
Couleur avec une variable qualitative

```
> ggplot(iris)+aes(x=Sepal.Length,y=Sepal.Width,
+   color=Species)+geom_point()
```



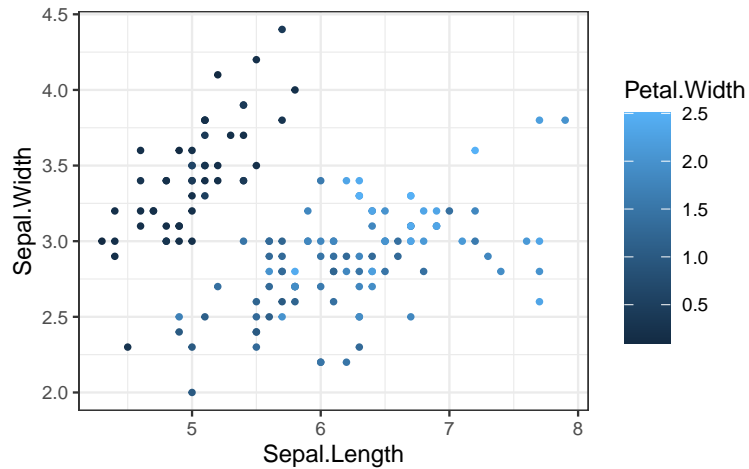
Changer la couleur

```
> ggplot(iris)+aes(x=Sepal.Length,y=Sepal.Width,
+                 color=Species)+geom_point()+
+   scale_color_manual(values=c("setosa"="blue","virginica"="green",
+                               "versicolor"="red"))
```



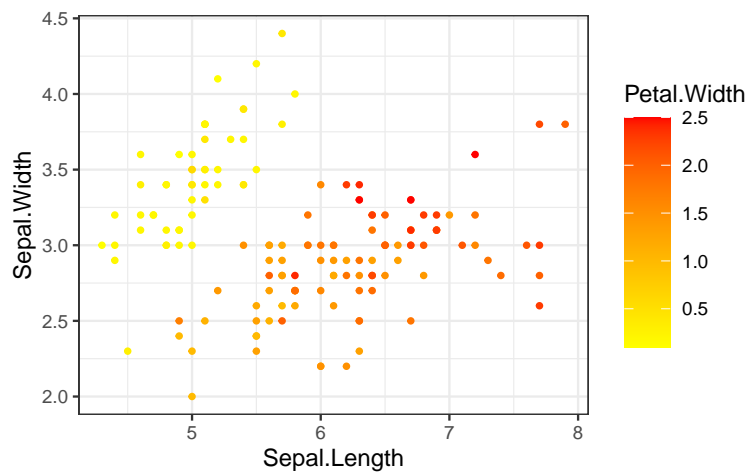
Couleur avec une variable continue

```
> ggplot(iris)+aes(x=Sepal.Length,y=Sepal.Width,
+                 color=Petal.Width)+geom_point()
```



Changer la couleur

```
> ggplot(iris)+aes(x=Sepal.Length,y=Sepal.Width,
+                 color=Petal.Width)+geom_point()+
+                 scale_color_continuous(low="yellow",high="red")
```



Histogramme

```
> ggplot(iris)+aes(x=Sepal.Length)+geom_histogram(fill="red")
```

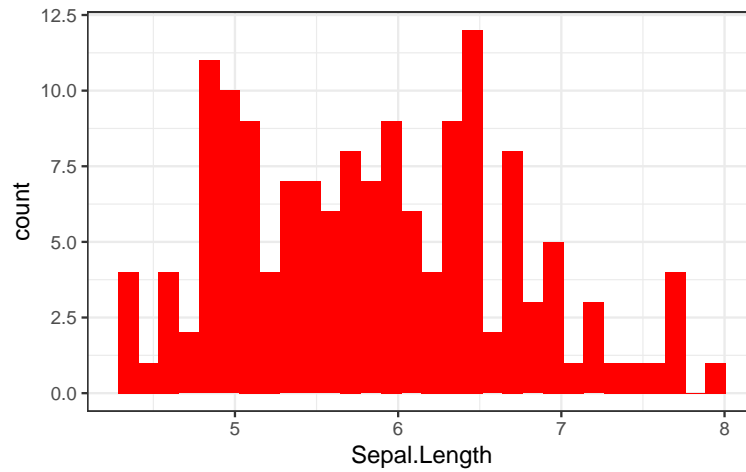
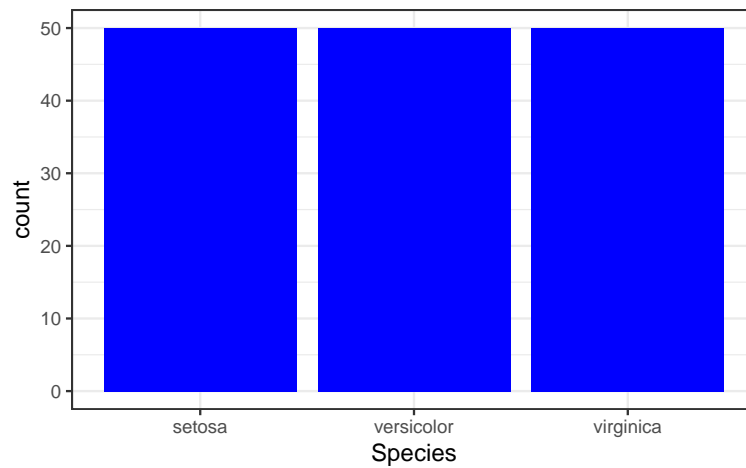


Diagramme en barres

```
> ggplot(iris)+aes(x=Species)+geom_bar(fill="blue")
```



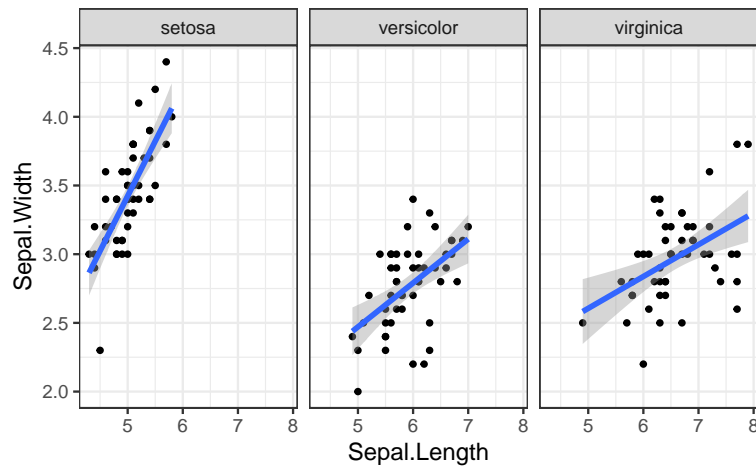
Exemples de geom

Geom	Description	Aesthetics
geom_point()	nuage de points	x, y, shape, fill
geom_line()	Ligne (ordonnée selon x)	x, y, linetype
geom_abline()	Ligne	slope, intercept
geom_path()	Ligne (ordonnée par l'index)	x, y, linetype
geom_text()	Texte	x, y, label, hjust, vjust
geom_rect()	Rectangle	xmin, xmax, ymin, ymax, fill, linetype
geom_polygon()	Polygone	x, y, fill, linetype
geom_segment()	Segment	x, y, xend, yend, fill, linetype

Geom	Description	Aesthetics
geom_bar()	Diagramme en barres	x, fill, linetype, weight
geom_histogram()	Histogramme	x, fill, linetype, weight
geom_boxplot()	Boxplot	x, fill, weight
geom_density()	Densité	x, y, fill, linetype
geom_contour()	Lignes de contour	x, y, fill, linetype
geom_smooth()	Lisseur (linéaire ou non linéaire)	x, y, fill, linetype
Tous		color, size, group

Facetting (très pertinent)

```
> ggplot(iris)+aes(x=Sepal.Length,y=Sepal.Width)+geom_point()+  
+ geom_smooth(method="lm")+facet_wrap(~Species)
```



Combiner **ggplot2** et **dplyr/tidyr**

- Souvent important de *construire un bon jeu de données* pour obtenir *un bon graphe*.
- Par exemple

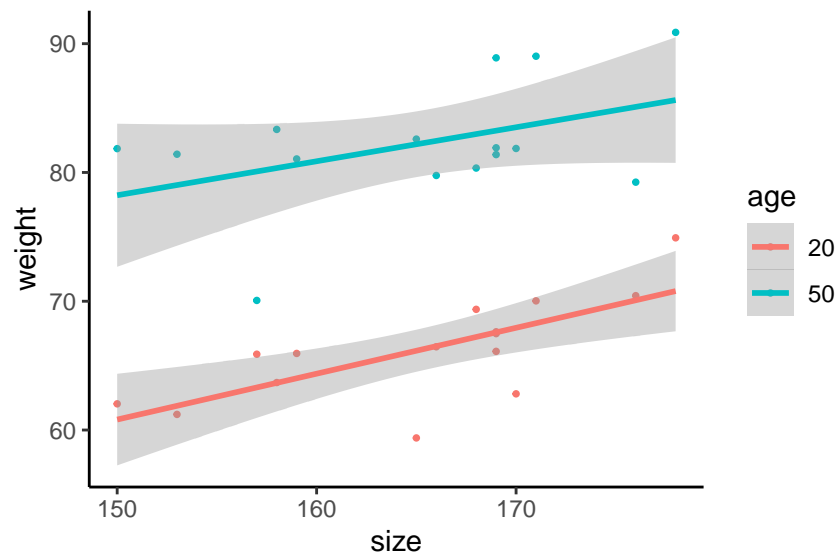
```
> head(df)
# A tibble: 6 x 3
  size weight.20 weight.50
<dbl> <dbl> <dbl>
1 153 61.2 81.4
2 169 67.5 81.4
3 168 69.4 80.3
4 169 66.1 81.9
5 176 70.4 79.2
6 169 67.6 88.9
```

Objectif

Etape **tidyr**

- Assembler les colonnes **weight.M** et **weight.W** en une colonne **weight** :

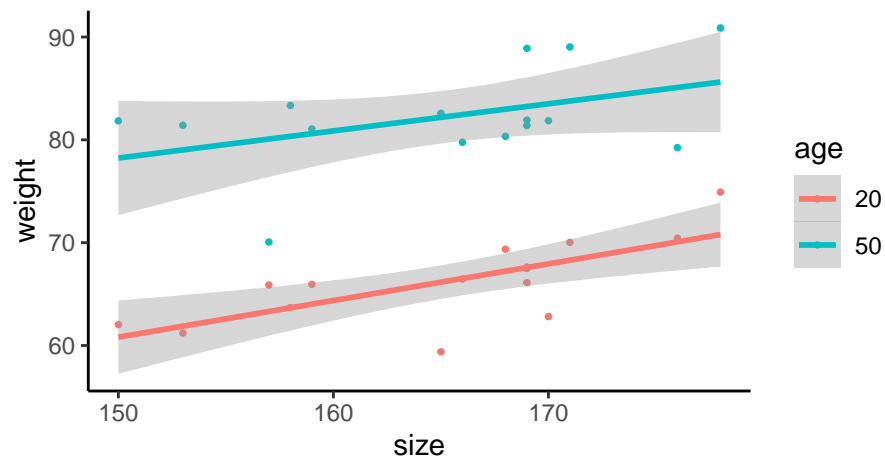
```
> df1 <- df |> pivot_longer(~size, names_to="age", values_to="weight")
> df1 |> head()
# A tibble: 6 x 3
  size age      weight
<dbl> <chr> <dbl>
1 153 weight.20 61.2
2 153 weight.50 81.4
```



```
3  169 weight.20  67.5
4  169 weight.50  81.4
5  168 weight.20  69.4
6  168 weight.50  80.3
> df1 <- df1 |> mutate(age=recode(age,
+   "weight.20"="20", "weight.50"="50"))
```

Etape ggplot2

```
> ggplot(df1)+aes(x=size,y=weight,color=age)+
+   geom_point()+geom_smooth(method="lm")+theme_classic()
```



Statistics

- Certains graphes nécessitent de calculer des *indicateurs* à partir des données.
- **Exemple de l'histogramme** : compter le nombre d'observations (ou la densité) dans chaque classe.

Conséquence

geom_histogram fait appel à la fonction stat_bin pour calculer ces indicateurs.

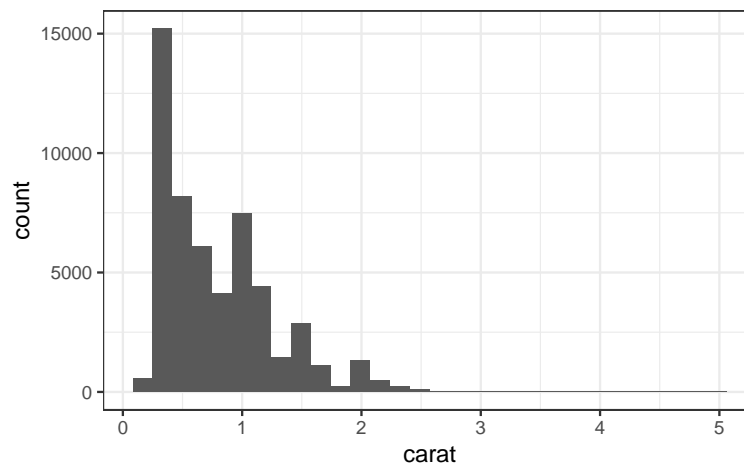
```
> geom_histogram(...,stat = "bin",...)
```

```
help(stat_bin)
Computed variables
count
number of points in bin

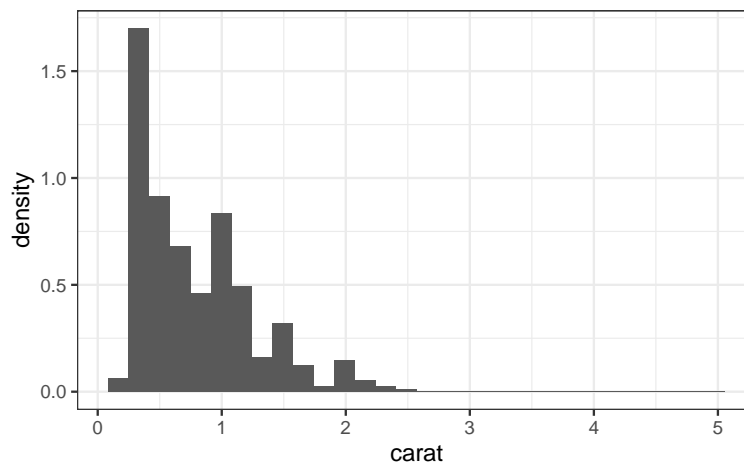
density
density of points in bin, scaled to integrate to 1
...
```

Visualiser une autre statistique

```
> ggplot(diamonds)+aes(x=carat)+
+   geom_histogram()
```



```
> ggplot(diamonds)+
+   aes(x=carat,y=after_stat(density))+
+   geom_histogram()
```

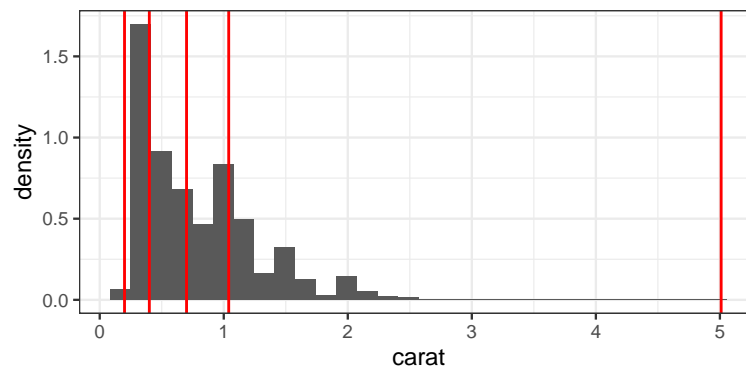


```
> #ou
> #ggplot(diamonds)+aes(x=carat,
> #                       y=..density..)+
> #   geom_histogram()
```

stat_summary

- D'une façon générale, `stat_summary` permet de calculer *n'importe quel indicateur* nécessaire au graphe.

```
> ggplot(diamonds)+aes(x=carat)+
+   geom_histogram(aes(y=after_stat(density)))+
+   stat_summary(aes(y=0,xintercept=after_stat(x)),
+     fun="quantile",geom="vline",
+     orientation = "y",color="red")
```

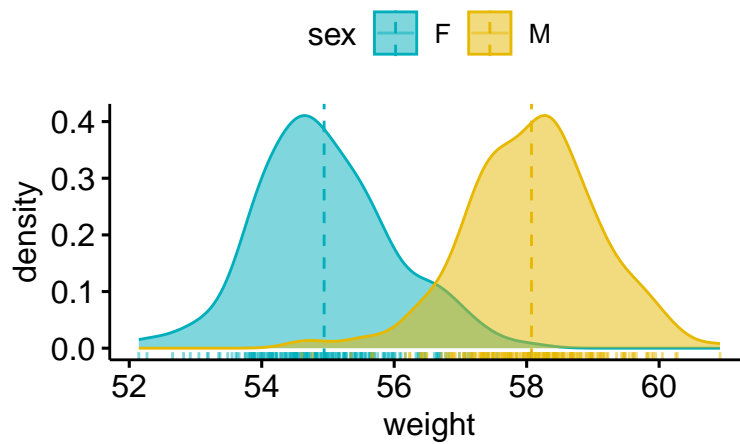


Compléments **ggpubr**

- Permet de faire des graphes ggplot relativement simples avec une *syntaxe simplifiée* (notamment sans l'utilisation de `aes`).
- Voir <https://rpkgs.datanovia.com/ggpubr/>

```
> head(wdata, 4)
  sex  weight
1  F 53.79293
2  F 55.27743
3  F 56.08444
4  F 52.65430
```

```
> library(ggpubr)
> ggdensity(wdata, x = "weight",
+   add = "mean", rug = TRUE,
+   color = "sex", fill = "sex",
+   palette = c("#00AFBB", "#E7B800"))
```



Compléments : quelques démos

```
> demo(image)
> example(contour)
> demo(persp)
> library("lattice");demo(lattice)
> example(wireframe)
> library("rgl");demo(rgl)
> example(persp3d)
> demo(plotmath);demo(Hershey)
```

7 Cartes

Introduction

- De nombreuses applications nécessitent des *cartes* pour *visualiser* des *données* ou les résultats d'un *modèle*.
- De *nombreux packages R* : *ggplot2*, *RgoogleMaps*, *maps*...
- Dans cette partie : *ggplot2*, *sf* (cartes *statiques*) et *leaflet* (cartes *dynamiques*).

7.1 ggplot2

Syntaxe

- `ggplot2` permet de récupérer des fonds de carte avec `map_data`

```
> fond <- map_data(...)
```

- La syntaxe reste similaire par la suite :

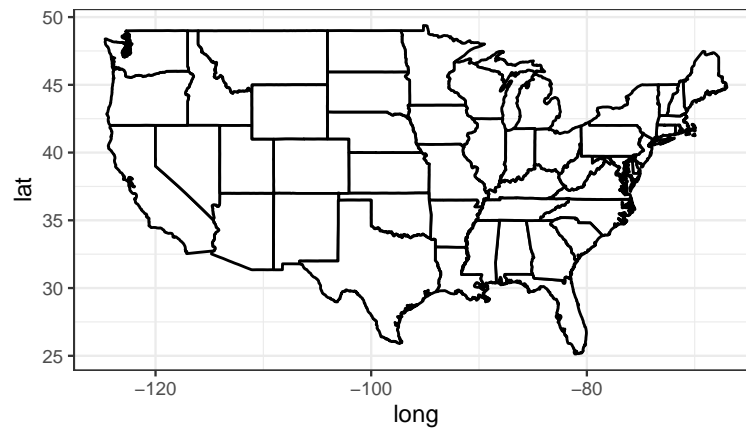
```
> ggplot(fond)+aes(...)
```

- On pourra consulter <https://ggplot2-book.org/maps#sec-polygonmaps>

Fonds de carte `map_data`

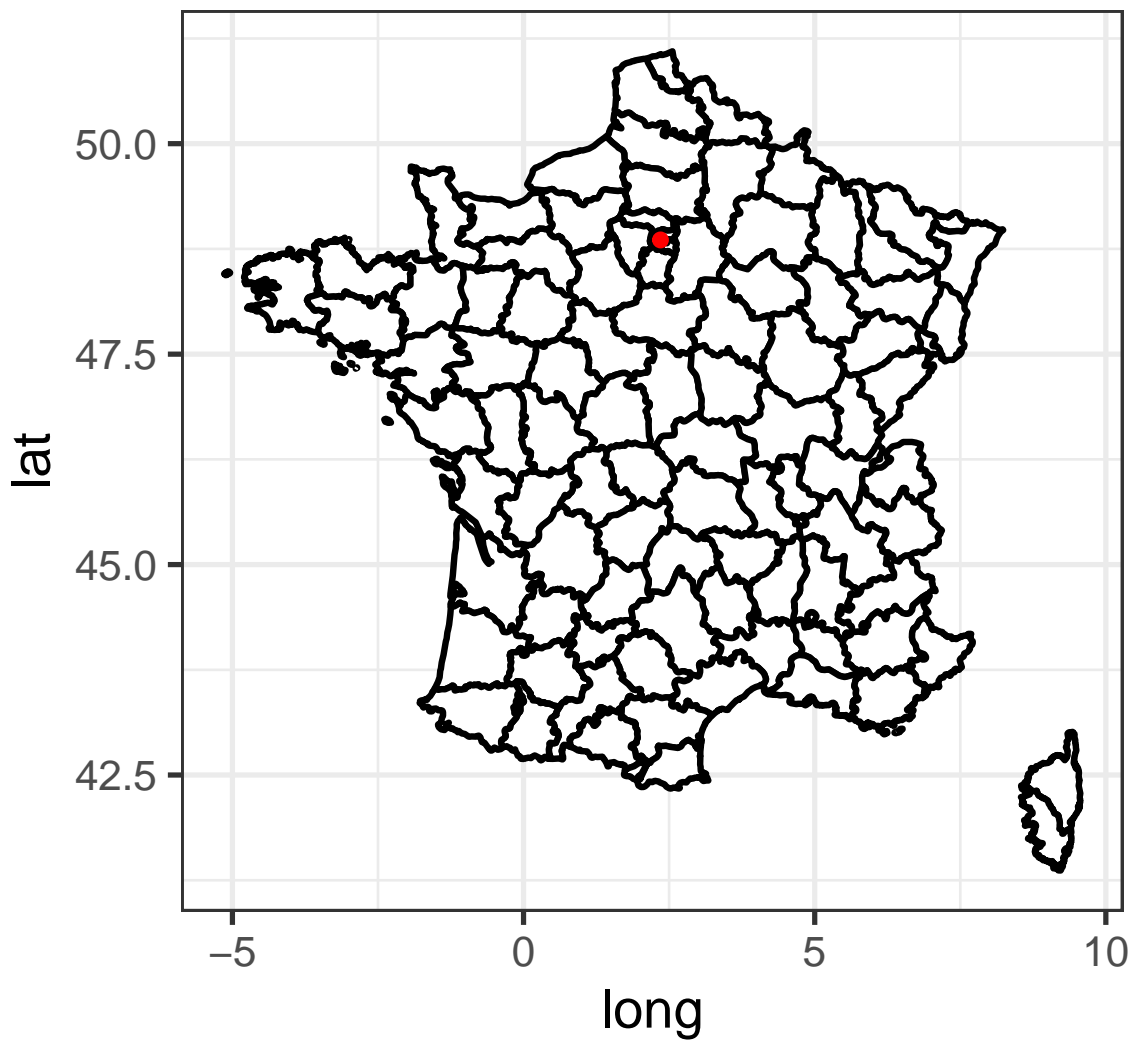
```
> us <- map_data("state")
> head(us)
      long      lat group order  region subregion
1 -87.46201 30.38968    1     1 alabama    <NA>
2 -87.48493 30.37249    1     2 alabama    <NA>
3 -87.52503 30.37249    1     3 alabama    <NA>
4 -87.53076 30.33239    1     4 alabama    <NA>
5 -87.57087 30.32665    1     5 alabama    <NA>
6 -87.58806 30.32665    1     6 alabama    <NA>
```

```
> ggplot(us)+aes(x=long,y=lat,group=group)+
+   geom_polygon(fill="white",color="black")+coord_quickmap()
```

Ajouts de points

```
> fr <- map_data("france")
> Paris <- tibble(long=2.351499,lat=48.85661)
> ggplot(fr)+aes(x=long,y=lat)+
+   geom_polygon(aes(group=group),fill="white",color="black")+
+   geom_point(data=Paris,color="red")+
+   coord_quickmap()
```



Géolocalisation

- Le package `tidygeocoder` propose de nombreux outils pour *géolocaliser* des lieux.
- Avec notamment la fonction `geocode` :

```
> library(tidygeocoder)
> tibble(ville=c("Paris", "Lyon", "Marseille", "Rennes")) |>
+   geocode(city=ville)
# A tibble: 4 x 3
  ville      lat long
<chr>    <dbl> <dbl>
1 Paris    48.9  2.32
```

2	Lyon	45.8	4.83
3	Marseille	43.3	5.37
4	Rennes	48.1	-1.68

7.2 Contours shapefile contours avec sf

Le package sf







- `ggplot2` : bien pour des cartes “simples” (fond et quelques points).
- *Pas suffisant* pour des **représentations plus complexes** (considérer une région comme un individu statistique).
- `sf` (*Simple Features*) permet de gérer des *objets spécifiques à la cartographie* : notamment les différents **systèmes de coordonnées** et **leurs projections en 2d** (latitudes-longitudes, World Geodesic System 84...)
- Fonds de carte au format *shapefile* (**contours** = **polygones**)
- Compatible avec `ggplot2` (verbe `geom_sf`).

Références

- <https://statnmap.com/fr/2018-07-14-initiation-a-la-cartographie-avec-sf-et-compagnie/>
- **Vignettes** sur le cran : <https://cran.r-project.org/web/packages/sf/index.html>
- Un *tutoriel* très complet (un peu technique) : <https://r-spatial.github.io/sf/articles/>
- Le chapitre <https://ggplot2-book.org/maps#sec-sf>

Le format shapefile

- Format de fichiers pour les *systèmes d'information géographiques (SIG)*.
- Permet de stocker la **forme**, la **localisation** et les **attributs** d'entités géographiques.
- Stocker sous la forme d'un *ensemble de fichiers*.

-  **departement.dbf**
-  **departement.lyr**
-  **departement.prj**
-  **departement.shp**
-  **departement.shx**
-  **departement.avl**

Lire des fichiers shapefile

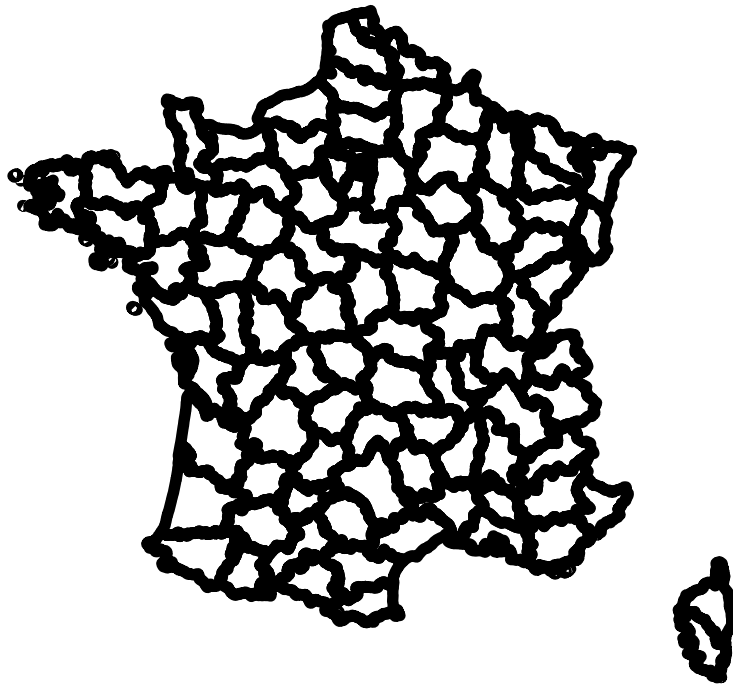
```
> library(sf)
> dpt <- read_sf("./data/dpt")
> dpt[1:5,3]
Simple feature collection with 5 features and 1 field
Geometry type: MULTIPOLYGON
Dimension: XY
Bounding box: xmin: 644570 ymin: 6290136 xmax: 1022851 ymax: 6997000
Projected CRS: RGF93 v1 / Lambert-93
# A tibble: 5 x 2
  NOM_DEPT geometry
  <chr> <MULTIPOLYGON [m]>
1 AIN (((919195 6541470, 918932 6541203, 918628 6540523, 91~
2 AISNE (((735603 6861428, 735234 6861392, 734504 6861270, 73~
3 ALLIER (((753769 6537043, 753554 6537318, 752879 6538099, 75~
4 ALPES-DE-HAUTE-PROVENCE (((992638 6305621, 992263 6305688, 991610 6306540, 99~
5 HAUTES-ALPES (((1012913 6402904, 1012577 6402759, 1010853 6402931,~
> class(dpt)
[1] "sf" "tbl_df" "tbl" "data.frame"
```

Cr  er un objet sf

```
> fr <- map_data("france") |> as_tibble()
> fr1 <- fr |>
+   select(long,lat) |>
+   split(fr$group) |>
+   map(as.matrix) |>
+   st_polygon() |>
+   st_sfc(crs=4326)
> fr1
Geometry set for 1 feature
Geometry type: POLYGON
Dimension: XY
Bounding box: xmin: -5.14209 ymin: 41.366 xmax: 9.562665 ymax: 51.09752
Geodetic CRS: WGS 84
POLYGON ((2.557093 51.09752, 2.579995 51.00298,...
```

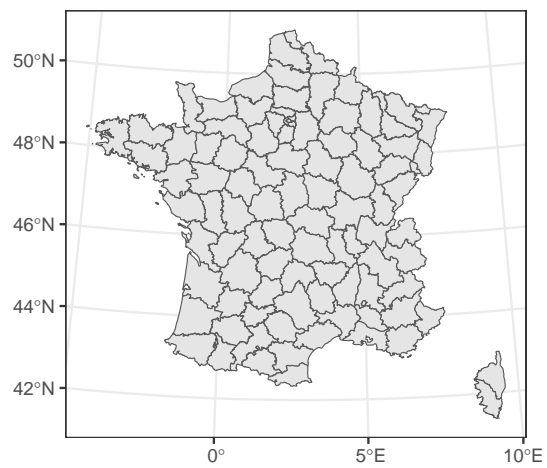
Visualisation avec plot

```
> plot(st_geometry(dpt))
```



Visualisation ggplot

```
> ggplot(dpt)+geom_sf()
```



Ajouter des points sur le graphe

- Définir des coordonnées avec `st_point`

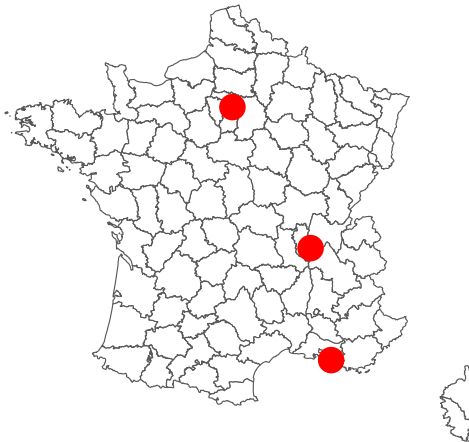
```
> point <- st_sfc(st_point(c(2.351462,48.85670)),  
+                 st_point(c(4.832011,45.75781)),  
+                 st_point(c(5.369953,43.29617)))
```

- Spécifier le *système de coordonnées* (4326 pour lat-lon)

```
> st_crs(point) <- 4326 #coord sont des long/lat  
> point  
Geometry set for 3 features  
Geometry type: POINT  
Dimension: XY  
Bounding box: xmin: 2.351462 ymin: 43.29617 xmax: 5.369953 ymax: 48.8567  
Geodetic CRS: WGS 84  
POINT (2.351462 48.8567)  
POINT (4.832011 45.75781)  
POINT (5.369953 43.29617)
```

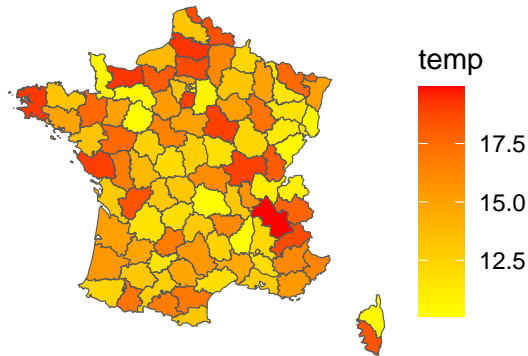
Étape ggplot

```
> ggplot(dpt) + geom_sf(fill="white")+  
+   geom_sf(data=point,color="red",size=4)+theme_void()
```



Colorier des polygones

```
> set.seed(1234)
> dpt1 <- dpt |> mutate(temp=runif(96,10,20))
> ggplot(dpt1) + geom_sf(aes(fill=temp)) +
+   scale_fill_continuous(low="yellow",high="red")+
+   theme_void()
```



Compléments : la classe geometry

- Une des forces de `sf` est la classe `geometry` qu'il propose.
- C'est cette classe qui conduit la représentation avec `plot` ou `geom_sf` :
 - `point` ou `multipoint` points pour localiser un lieu
 - `polygon` ou `multipolygon` contours pour représenter des frontières.
 - `linestring` ou `multilinestring` lignes pour représenter des fleuves, des routes...
- Quelques fonctions utiles :
 - `st_point` et `st_multipoint` : créer des points ou suite de points
 - `st_sfc` : créer une liste d'objets `sf`
 - `st_geometry` : extraire, modifier, remplacer, créer le geometry d'un objet
 - `st_crs` : spécifier le système de coordonnées d'un geometry
 - `st_cast` : transformer le type de geometry (passer d'un `MULTIPOINTS` à plusieurs `POINTS` par exemple)
 - ...
- Création d'un objet `point`


```

> b1 <- st_point(c(3,4))
> b1
POINT (3 4)
> class(b1)
[1] "XY" "POINT" "sfg"

```

- Création d'un objet **sfc** (liste avec des caractéristiques géométriques)

```

> b2 <- st_sfc(st_point(c(1,2)),st_point(c(3,4)))
> b2
Geometry set for 2 features
Geometry type: POINT
Dimension: XY
Bounding box: xmin: 1 ymin: 2 xmax: 3 ymax: 4
CRS: NA
POINT (1 2)
POINT (3 4)
> class(b2)
[1] "sfc_POINT" "sfc"

```

- Extraction, ajout, remplacement d'un **geometry**

```

> class(dpt)
[1] "sf" "tbl_df" "tbl" "data.frame"
> b3 <- st_geometry(dpt)
> b3
Geometry set for 96 features
Geometry type: MULTIPOLYGON
Dimension: XY
Bounding box: xmin: 99226 ymin: 6049647 xmax: 1242375 ymax: 7110524
Projected CRS: RGF93 v1 / Lambert-93
First 5 geometries:
MULTIPOLYGON (((919195 6541470, 918932 6541203,...
MULTIPOLYGON (((735603 6861428, 735234 6861392,...
MULTIPOLYGON (((753769 6537043, 753554 6537318,...
MULTIPOLYGON (((992638 6305621, 992263 6305688,...
MULTIPOLYGON (((1012913 6402904, 1012577 640275...
> class(b3)
[1] "sfc_MULTIPOLYGON" "sfc"

```

7.3 Cartes interactives avec leaflet

Fonds de carte

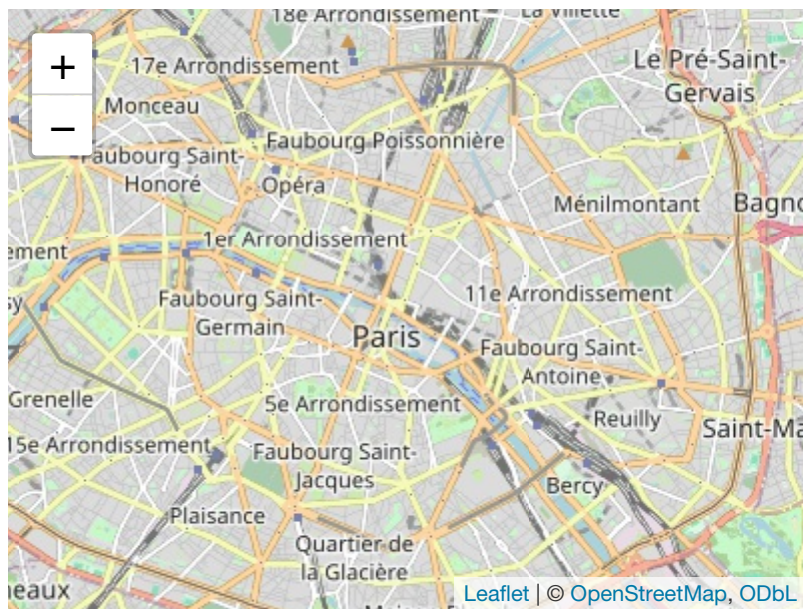
- **Leaflet** est une des librairies open-source JavaScript les plus populaires pour faire des **cartes interactives**.
- *Documentation*: <https://rstudio.github.io/leaflet/>

```
> library(leaflet)
> leaflet() |> addTiles()
```



Différents styles de fonds de carte

```
> Paris <- c(2.35222, 48.856614)
> leaflet() |> addTiles() |>
+   setView(lng = Paris[1], lat = Paris[2], zoom=12)
```



```
> leaflet() |>  
+   addProviderTiles(providers$Esri.NatGeoWorldMap) |>  
+   setView(lng = Paris[1], lat = Paris[2], zoom = 12)
```



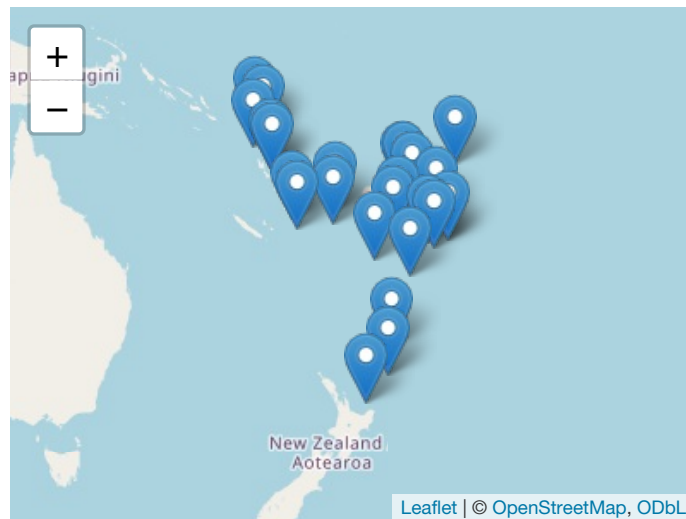
Avec des données

- Localiser 1000 séismes près des Fiji

```
> data(quakes)
> head(quakes)
      lat  long depth mag stations
1 -20.42 181.62   562 4.8        41
2 -20.62 181.03   650 4.2         15
3 -26.00 184.10    42 5.4         43
4 -17.97 181.66   626 4.1         19
5 -20.42 181.96   649 4.0         11
6 -19.68 184.31   195 4.0         12
```

Séismes avec une magnitude plus grande que 5.5

```
> quakes1 <- quakes |> filter(mag>5.5)
> leaflet(data = quakes1) |> addTiles() |>
+   addMarkers(~long, ~lat, popup = ~as.character(mag))
```

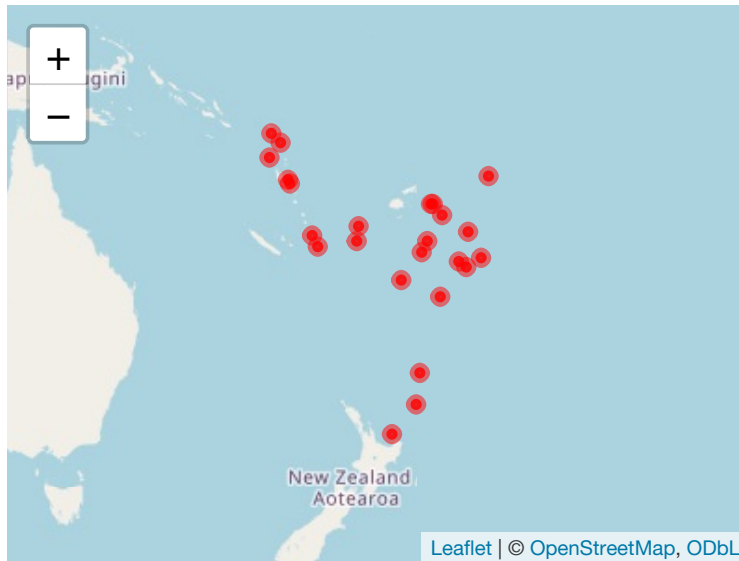


Remarque

La magnitude apparaît lorsqu'on **clique sur un marker**.

addCircleMarkers

```
> leaflet(data = quakes1) |> addTiles() |>  
+   addCircleMarkers(~long, ~lat, popup=~as.character(mag),  
+                   radius=3,fillOpacity = 0.8,color="red")
```



Des polygones utilisant sf

```
> states <- spData::us_states
> states
Simple feature collection with 49 features and 6 fields
Geometry type: MULTIPOLYGON
Dimension: XY
Bounding box: xmin: -124.7042 ymin: 24.55868 xmax: -66.9824 ymax: 49.38436
Geodetic CRS: NAD83
First 10 features:
```

	GEOID	NAME	REGION	AREA	total_pop_10	total_pop_15
1	01	Alabama	South	133709.27 [km ²]	4712651	4830620
2	04	Arizona	West	295281.25 [km ²]	6246816	6641928
3	08	Colorado	West	269573.06 [km ²]	4887061	5278906
4	09	Connecticut	Northeast	12976.59 [km ²]	3545837	3593222
5	12	Florida	South	151052.01 [km ²]	18511620	19645772
6	13	Georgia	South	152725.21 [km ²]	9468815	10006693
7	16	Idaho	West	216512.66 [km ²]	1526797	1616547
8	18	Indiana	Midwest	93648.40 [km ²]	6417398	6568645
9	20	Kansas	Midwest	213037.08 [km ²]	2809329	2892987
10	22	Louisiana	South	122345.76 [km ²]	4429940	4625253

```

      geometry
1 MULTIPOLYGON (((-88.20006 3...
2 MULTIPOLYGON (((-114.7196 3...
3 MULTIPOLYGON (((-109.0501 4...
4 MULTIPOLYGON (((-73.48731 4...
5 MULTIPOLYGON (((-81.81169 2...
```

```

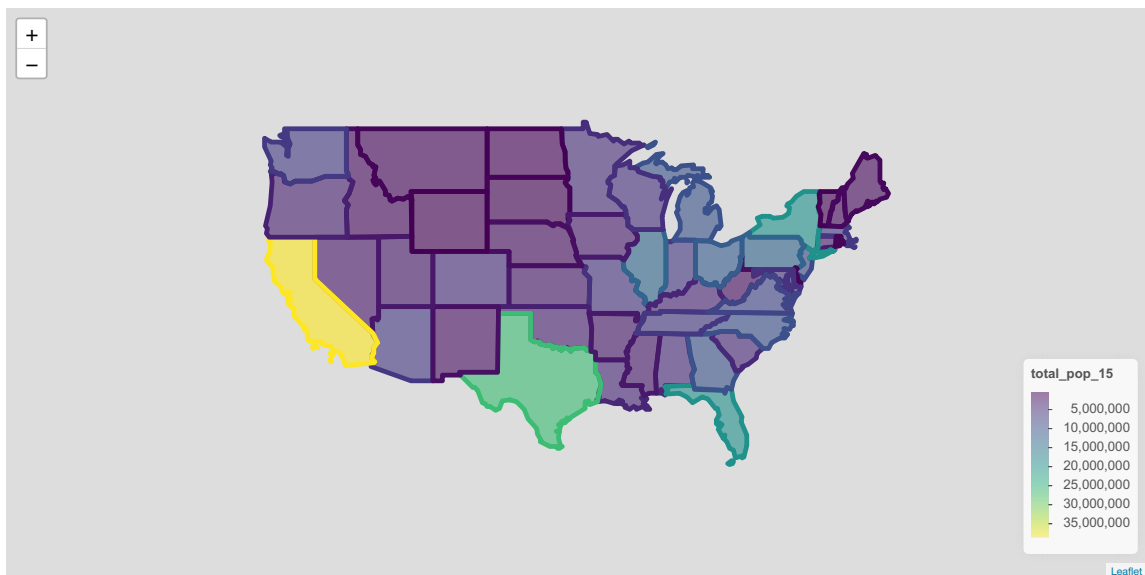
6 MULTIPOLYGON (((-85.60516 3...
7 MULTIPOLYGON (((-116.916 45...
8 MULTIPOLYGON (((-87.52404 4...
9 MULTIPOLYGON (((-102.0517 4...
10 MULTIPOLYGON (((-92.01783 2...

```

```

> pal1 <- colorNumeric(palette = c("viridis"),domain = states$total_pop_15)
> leaflet(states) |>
+   addPolygons(color=~pal1(total_pop_15),
+               popup=~str_c(as.character(NAME),
+                             as.character(total_pop_15),sep=" : "),
+               fillOpacity = 0.6,
+               opacity = 1) |>
+   addLegend(pal=pal1,value=~total_pop_15,position="bottomright")

```



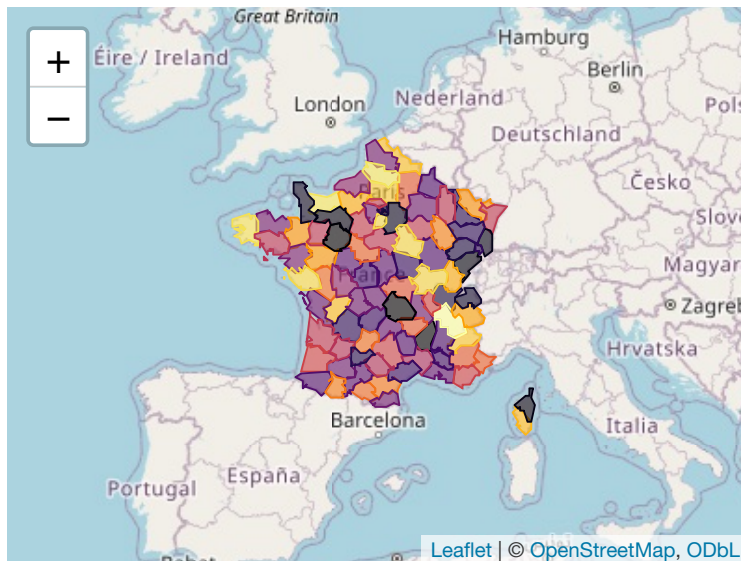
L'exemple des températures

```

> dpt2 <- st_transform(dpt1,crs=4326)
> pal2 <- colorNumeric(palette = c("inferno"),domain = dpt2$temp)
> leaflet() |> addTiles() |>

```

```
+ addPolygons(data = dpt2,color=~pal2(temp),fillOpacity = 0.6,
+             stroke = TRUE,weight=1,
+             popup=~paste(as.character(NOM_DEPT),
+                           as.character(temp),sep=" : "),
+             opacity = 1)
```



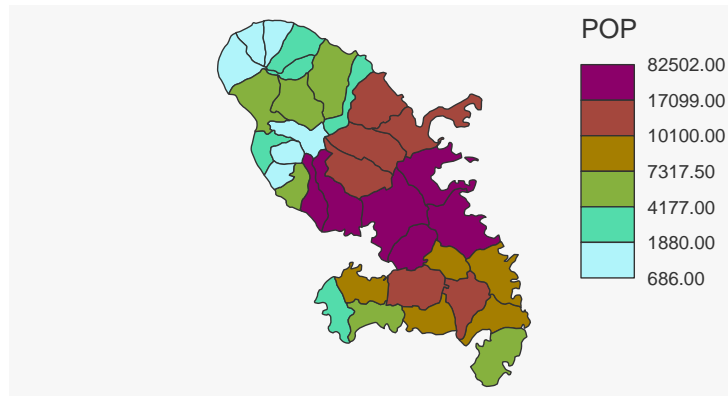
7.4 Autres packages carto

mapsf

```
> library(mapsf)
> mtq <- mf_get_mfq()
> mtq |> select(3,4,8) |> head()
Simple feature collection with 6 features and 2 fields
Geometry type: MULTIPOLYGON
Dimension: XY
Bounding box: xmin: 695444 ymin: 1598818 xmax: 717731 ymax: 1645182
Projected CRS: WGS 84 / UTM zone 20N
```

	LIBGEO	POP	geom
1	L'Ajoupa-Bouillon	1902	MULTIPOLYGON (((699261 1637...
2	Les Anses-d'Arlet	3737	MULTIPOLYGON (((709840 1599...
3	Basse-Pointe	3357	MULTIPOLYGON (((697602 1638...
4	Le Carbet	3683	MULTIPOLYGON (((702229 1628...
5	Case-Pilote	4458	MULTIPOLYGON (((698805 1621...
6	Le Diamant	5976	MULTIPOLYGON (((709840 1599...


```
> #hcl.pals(type="sequential")
> mf_map(x=mtq,var="POP",type="choro",pal="Hawaii")
```

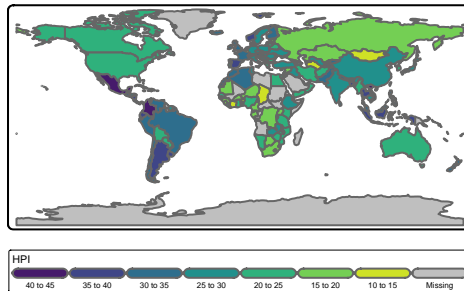


tmap

```
> library(tmap)
> #tmap_mode("view")
> data("World")
> World |> select(2,15,16) |> head()
Simple feature collection with 6 features and 2 fields
Geometry type: MULTIPOLYGON
Dimension: XY
Bounding box: xmin: -73.41544 ymin: -55.25 xmax: 75.15803 ymax: 42.68825
Geodetic CRS: WGS 84
```

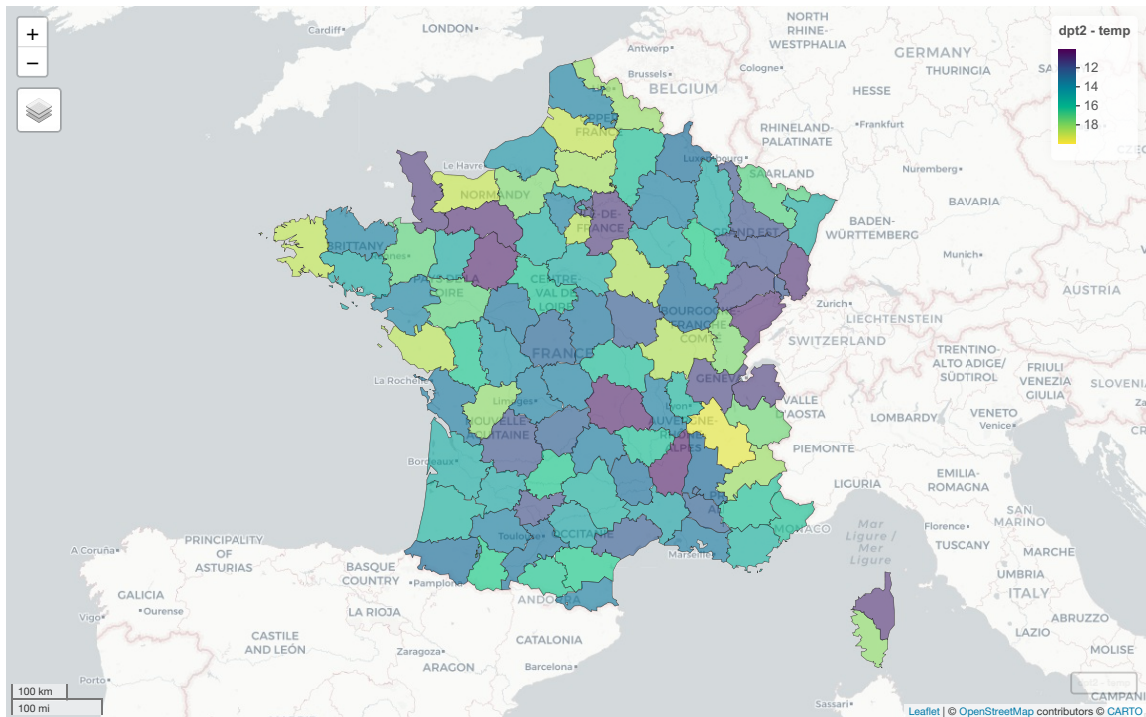
	name	HPI	geometry
1	Afghanistan	20.22535	MULTIPOLYGON (((61.21082 35...
2	Angola	NA	MULTIPOLYGON (((16.32653 -5...
3	Albania	36.76687	MULTIPOLYGON (((20.59025 41...
4	United Arab Emirates	NA	MULTIPOLYGON (((51.57952 24...
5	Argentina	35.19024	MULTIPOLYGON (((-65.5 -55.2...
6	Armenia	25.66642	MULTIPOLYGON (((43.58275 41...

```
> tm_shape(World) +
+   tm_polygons("HPI",
+               palette="viridis",
+               legend.is.portrait=FALSE,
+               legend.reverse=TRUE)
```



mapview

```
> library(mapview)
> mapview(dpt2,zcol="temp",
+         popup=leaflet::popupTable(dpt2, zcol = c("NOM_DEPT", "temp")))
```



Références

- **mapsf** : <https://cran.r-project.org/web/packages/mapsf/vignettes/mapsf.html>
- **tmap** : <https://cran.r-project.org/web/packages/tmap/vignettes/tmap-getstarted.html>
- *Tutoriel thinkr* :
 - <https://thinkr.fr/cartographie-interactive-comment-visualiser-mes-donnees-spatiales-de-maniere-dynamique-avec-leaflet/>
 - <https://thinkr.fr/cartographie-interactive-avec-r-la-suite/>

8 Quelques outils de visualisation dynamiques

Des packages R

- Graphiques classiques avec **rAmCharts**, **plotly** et **ggiraph**.

- Graphes avec `visNetwork`.
- Tableaux de bord avec `flexdashboard`.

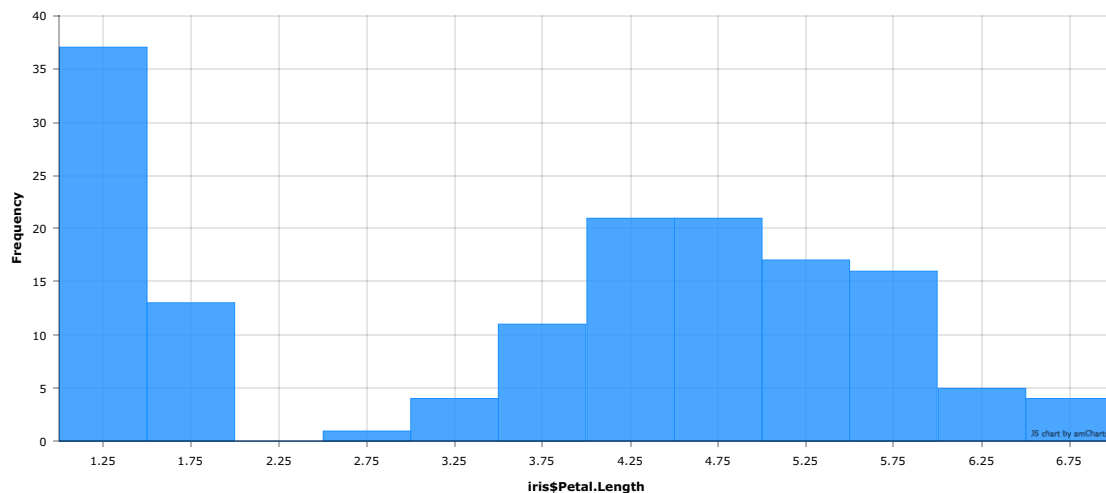
8.1 rAmCharts, plotly et ggiraph

rAmCharts

- *User-friendly* pour des graphes standards (nuages de points, séries chronologiques, histogrammes...).
- Il suffit d'utiliser la fonction **R** classique avec le préfixe `am`.
- *Exemples* : `amPlot`, `amHist`, `amBoxplot`.
- *Références* : https://datastorm-open.github.io/introduction_ramcharts/

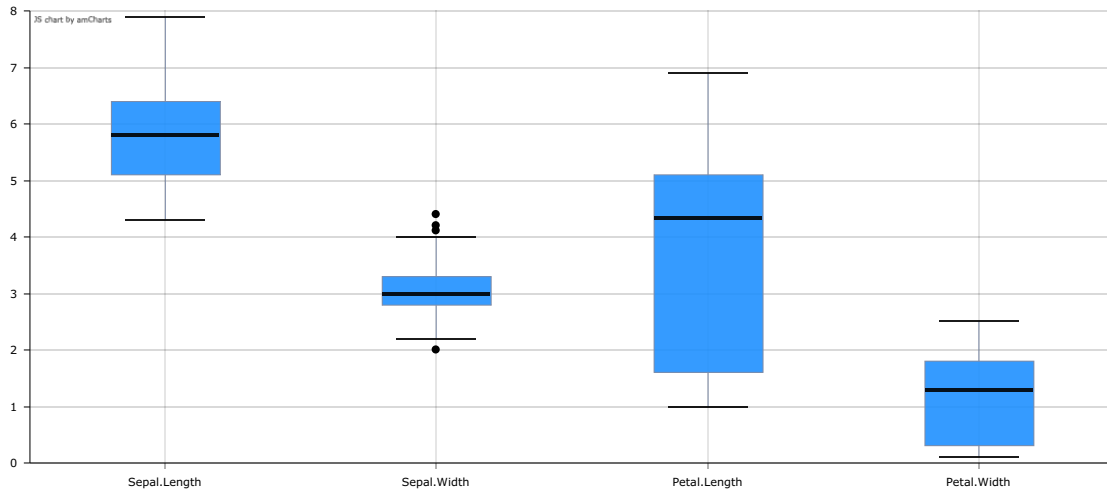
rAmCharts Histogramme

```
> library(rAmCharts)
> amHist(iris$Petal.Length)
```



rAmcharts Boxplot

```
> amBoxplot(iris)
```

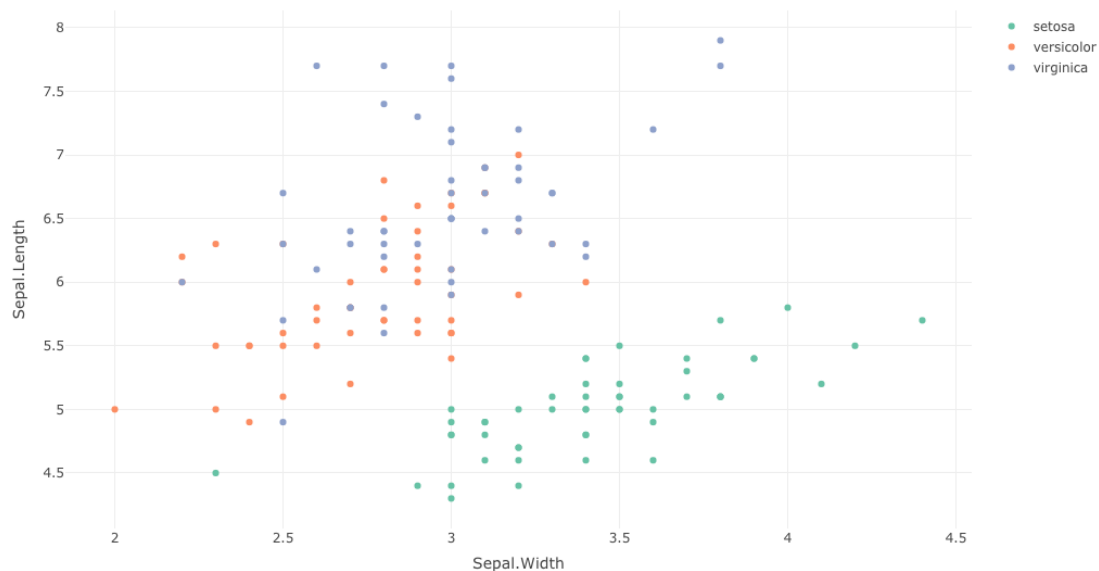


Plotly

- Package **R** pour créer des *graphes dynamiques* à partir de la librairie open source Javascript *plotly.js*.
- La syntaxe se décompose en *3 parties* :
 - données et variables (`plot_ly`) ;
 - type de représentation (`add_trace`, `add_markers...`) ;
 - options (axes, titres...) (`layout`).
- Références: <https://plot.ly/r/reference/>

Nuage de points

```
> library(plotly)
> iris |> plot_ly(x=~Sepal.Width,y=~Sepal.Length,color=~Species) |>
+   add_markers(type="scatter")
```



Plotly boxplot

```
> iris |> plot_ly(x=~Species,y=~Petal.Length) |> add_boxplot()
```

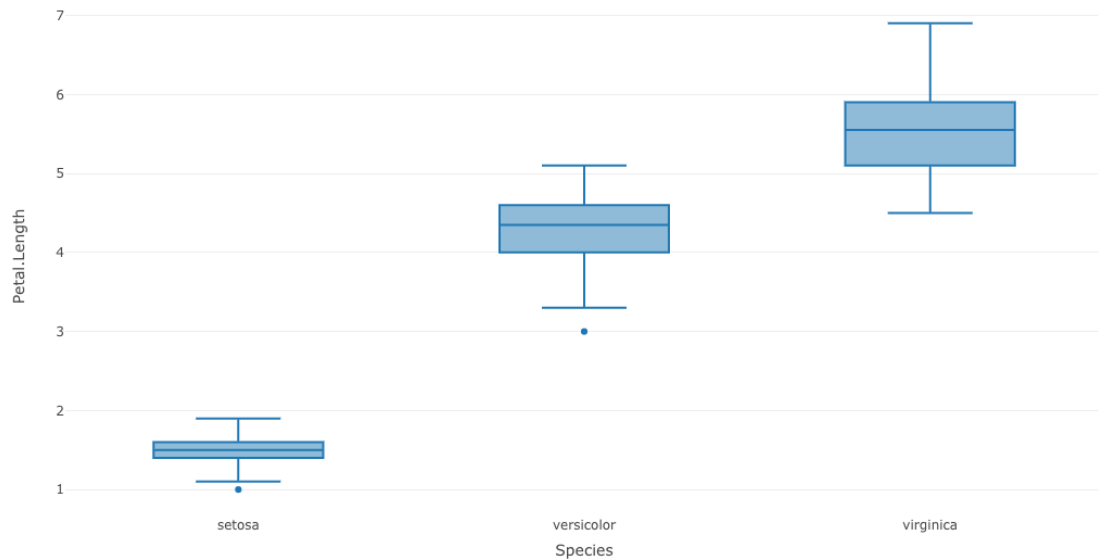
ggiraph

- Extension de `ggplot2` pour des graphes *dynamiques* et *interactifs*.
- Basé sur les fonctions `ggplot2` avec ajout du suffixe `_interactive`.
- *Documentation* : <https://www.ardata.fr/ggiraph-book/>

Le principe

- Création du graphe :

```
> library(ggiraph)
> p <- ggplot(data=iris) +
+   aes(x=Sepal.Length,y=Sepal.Width,
+       tooltip=Petal.Width,data_id=Species)+
+   geom_point_interactive(size=1,hover_nearest=TRUE)
```

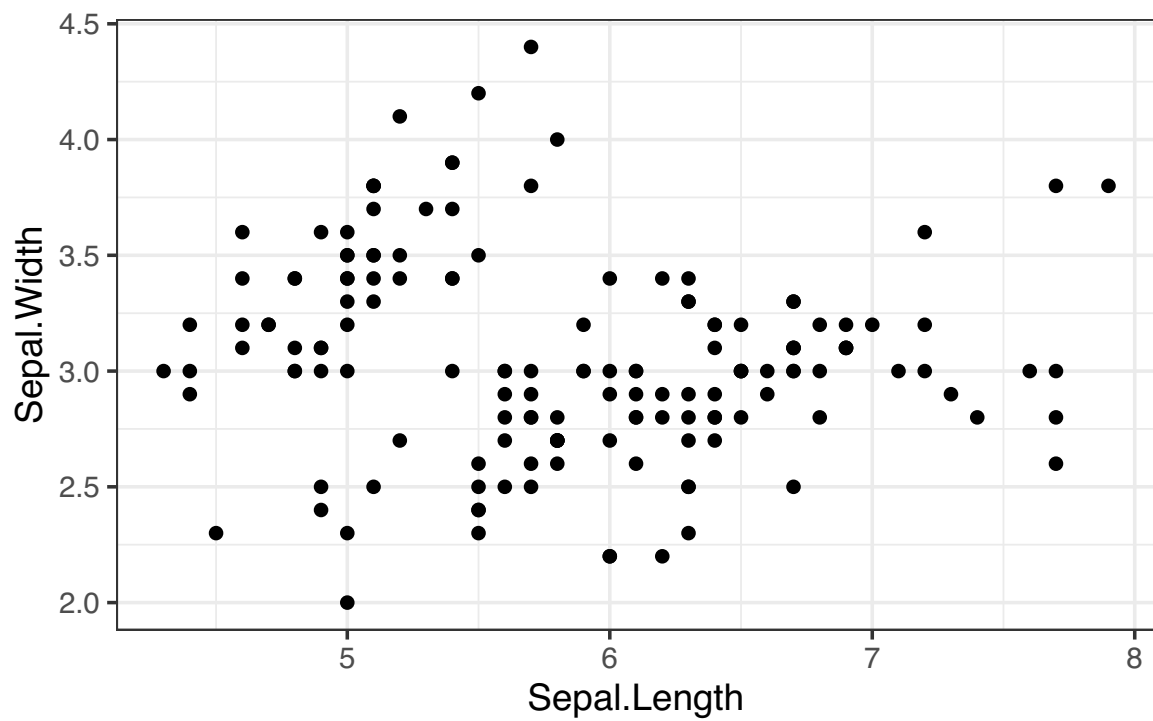


Interprétation

- `data_id=Species` : identifie les points de la même espèce que celui où se trouve la souris.
- `tooltip=Petal.Width` : affiche la largeur de pétale du point où se trouve la souris.

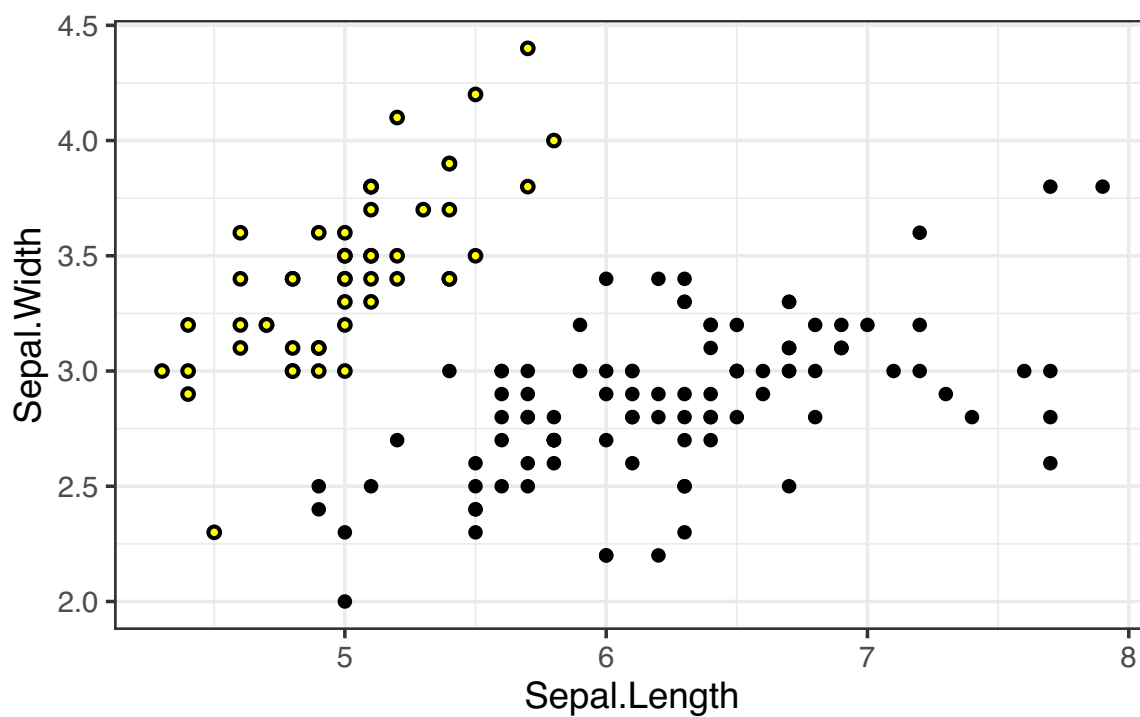
- Visualisation avec la fonction `girafe`

```
> girafe(ggobj=p)
```



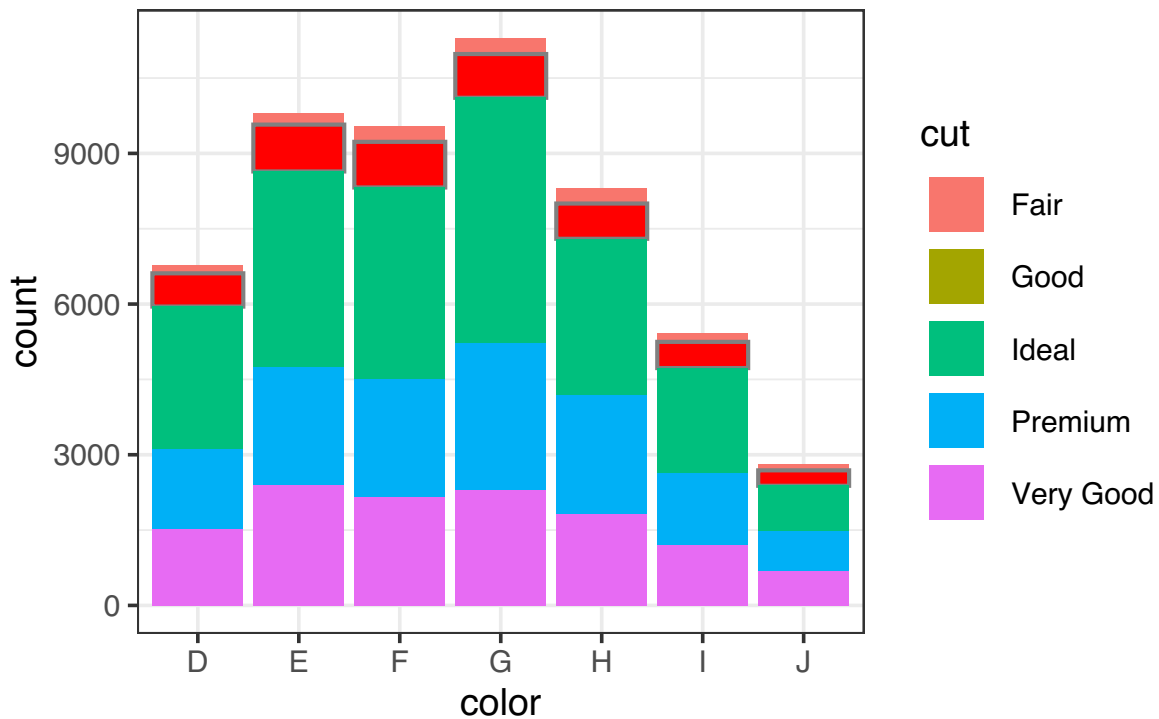
Améliorer avec du css

```
> preselection <- iris$Species[1]
> girafe(ggobj = p,
+   options=list(
+     opts_hover(css="fill:blue;stroke:black;stroke-width:2px;"),
+     opts_selection(
+       css="fill:yellow;stroke:black;stroke-width:1px;",
+       selected=preselection,type="multiple",only_shiny = FALSE)))
```

Autre exemple

```
> p1 <- ggplot(
+   data = diamonds,
+   mapping = aes(x = color, fill = cut, data_id = cut)
+ ) +
+   geom_bar_interactive(
+     aes(tooltip = sprintf("%s: %.0f", fill, after_stat(count))),
+     size = 3
+   )
> girafe(ggobj = p1,
+   options=list(
+     opts_selection(
+       selected="Good",
+       only_shiny = FALSE
+     )
+   )
+ )
```



8.2 Graphes avec visNetwork

Connexions entre individus

- De nombreux jeux de données peuvent être visualisés avec des *graphes*, notamment lorsque l'on souhaite étudier des *connexions* entre individus (génétique, réseaux sociaux, système de recommandation...)
- Un individu = *un nœud* et une connexion = *une arête*.

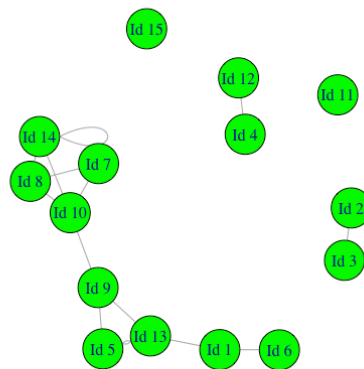
```
> set.seed(123)
> nodes <- data.frame(id = 1:15, label = paste("Id", 1:15))
> edges <- data.frame(from = trunc(runif(15)*(15-1))+1,
+                     to = trunc(runif(15)*(15-1))+1)
> head(edges)
  from to
1    5 13
2   12  4
3    6  1
4   13  5
```

```
5  14 14
6   1 13
```

Graphe statique : le package **igraph**

- *Références* : <http://igraph.org/r/>, <http://kateto.net/networks-r-igraph>

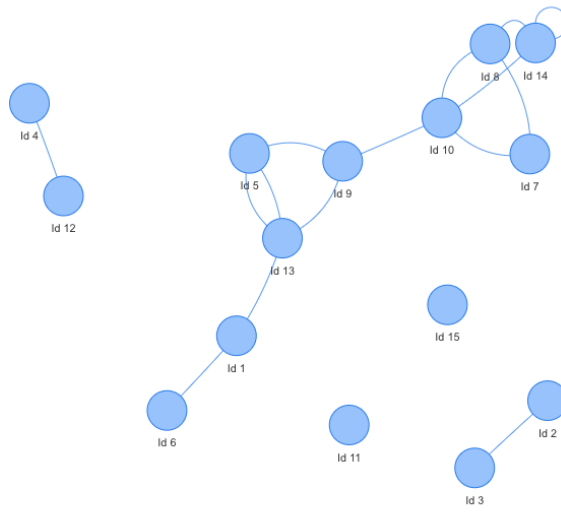
```
> library(igraph)
> net <- graph_from_data_frame(d=edges, vertices=nodes, directed=F)
> plot(net, vertex.color="green", vertex.size=25)
```



Graph dynamique : le package **visNetwork**

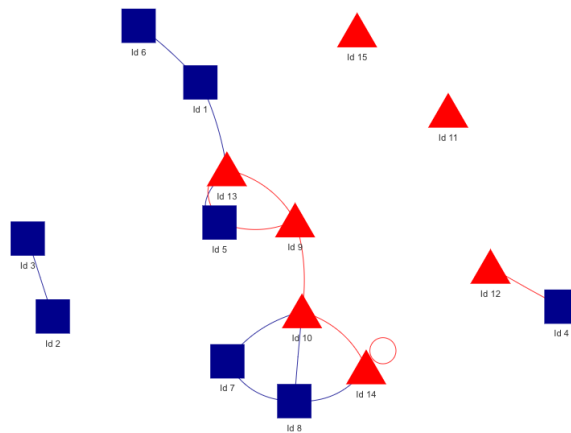
- *Référence* : <https://datastorm-open.github.io/visNetwork/interaction.html>

```
> library(visNetwork)
> visNetwork(nodes, edges)
```



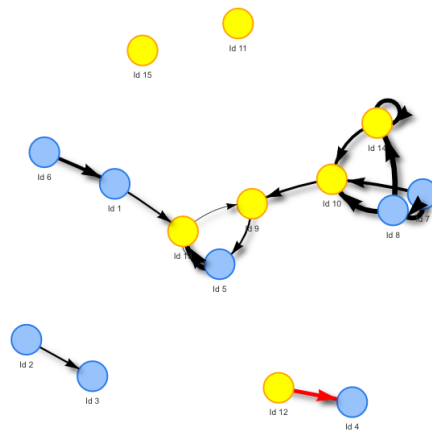
Nodes color

```
> nodes$group <- c(rep("A",8),rep("B",7))
> visNetwork(nodes,edges) |>
+   visGroups(groupname = "A", color = "darkblue",
+             shape = "square") |>
+   visGroups(groupname = "B", color = "red",
+             shape = "triangle")
```



Edeges width

```
> edges$width <- round(runif(nrow(edges),1,10))
> visNetwork(nodes,edges) |>
+   visEdges(shadow = TRUE,
+     arrows =list(to = list(enabled = TRUE)),
+     color = list(color = "black", highlight = "red"))
```



8.3 Tableau de bord avec flexdashboard

- Juste un outil... mais *un outil important* en science des données
- Permet *d'assembler des messages importants* sur des données et/ou modèles
- *Package* : flexdashboard
- *Syntaxe* : simple... juste du Rmarkdown
- *Référence* : <https://rmarkdown.rstudio.com/flexdashboard/>

Header

```
---
title: "My title"
output:
  flexdashboard::flex_dashboard:
    orientation: columns
    vertical_layout: fill
    theme: default
---
```

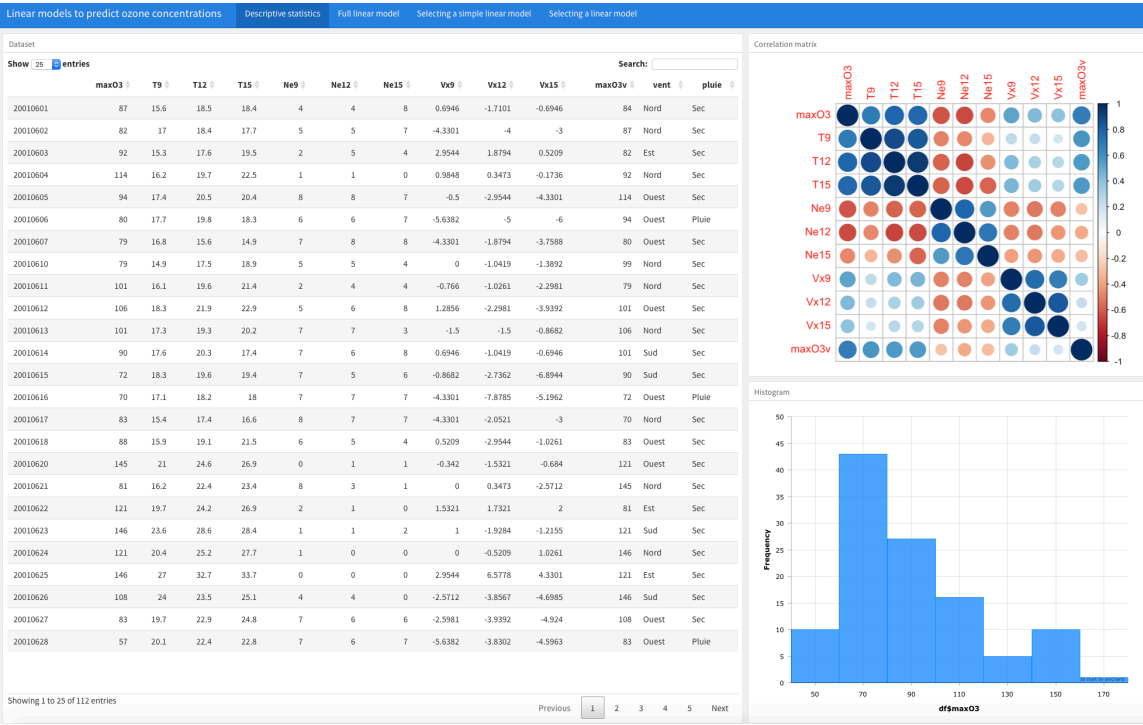
- Le thème par défaut peut être remplacé par d'*autres thèmes* (cosmo, bootstrap, cerulean...) (voir [ici](#)). Il suffit d'ajouter

```
theme: yeti
```

Flexdashboard | code

```
Descriptive statistics
=====
Column {data-width=650}
-----
### Dataset
```{r}
DT::datatable(df, options = list(pageLength = 25))
```
Column {data-width=350}
-----
### Correlation matrix
```{r}
cc <- cor(df[,1:11])
mat.cor <- corrplot::corrplot(cc)
```
### Histogram
```{r}
amHist(df$max03)
```
```

Flexdashboard | dashboard



Correlation matrix

Histogram