

# Introduction à R

Laurent Rouvière

21 mars 2023

## Table des matières

<b>1 Présentation du cours</b>	<b>2</b>
<b>2 Rstudio, quarto et packages R</b>	<b>10</b>
<b>3 Objets R</b>	<b>12</b>
<b>4 Gérer des données</b>	<b>19</b>
4.1 Importer des données . . . . .	19
4.2 Annexe : les fonctions read.table et read.csv . . . . .	23
4.3 Base de données avancées . . . . .	25
4.4 Fusion de tables . . . . .	28
4.5 Manipuler les données avec Dplyr . . . . .	30
4.6 Quelques fonctions utiles de tidyverse . . . . .	35
<b>5 Visualiser des données</b>	<b>37</b>
5.1 Graphes conventionnels . . . . .	37
5.2 Visualisation avec ggplot2 . . . . .	40
<b>6 Programmer en R</b>	<b>51</b>
6.1 Structures de contrôle . . . . .	51
6.2 Les fonctions map . . . . .	54
<b>7 Cartes</b>	<b>56</b>
7.1 ggmap . . . . .	56
7.2 Contours shapefile contours avec sf . . . . .	58
7.3 Cartes interactives avec leaflet . . . . .	64

<b>8 Quelques outils de visualisation dynamiques</b>	<b>69</b>
8.1 rAmCharts et plotly . . . . .	69
8.2 Graphes avec visNetwork . . . . .	72
8.3 Tableau de bord avec flexdashboard . . . . .	75

## 1 Présentation du cours

### Présentation

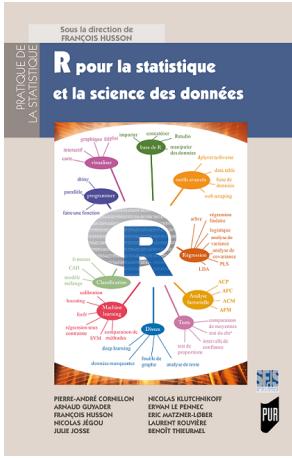
- *Enseignant* : Laurent Rouvière, laurent.rouviere@univ-rennes2.fr
  - *Recherche* : statistique non paramétrique, apprentissage statistique.
  - *Enseignement* : statistique et probabilités (Université, école d'ingénieur, formation continue).
  - *Consulting* : énergie (ERDF), finance, marketing, sport.
- *Prérequis* : bases en programmation, probabilités et statistique.
- *Objectifs* : comprendre et utiliser les outils R classiques en data science, 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/visualisationR.html](https://lrouviere.github.io/page_perso/visualisationR.html)
- *Tutoriel* : compléments de cours et exercices disponibles à l'url [https://lrouviere.github.io/TUTO\\_VISU\\_R/](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 scientits) 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**

#### **La problématique**

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

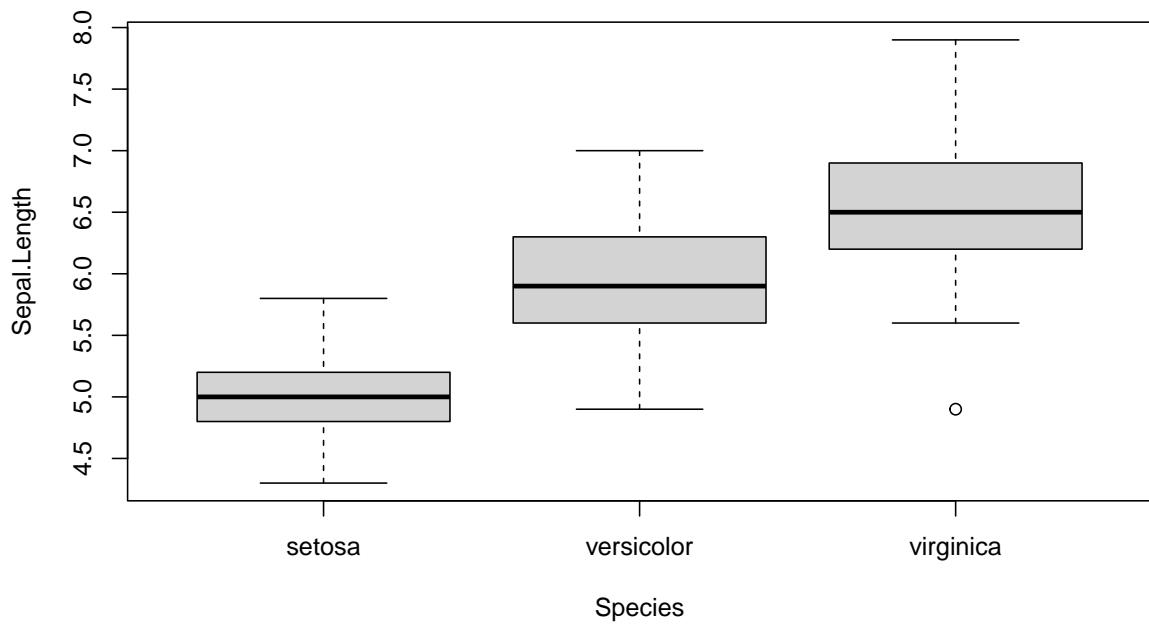
- **dplyr** est un package de **tidyverse** qui permet de faciliter la manipulation des données, notamment en terme de **syntaxe**.

```
> 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**.

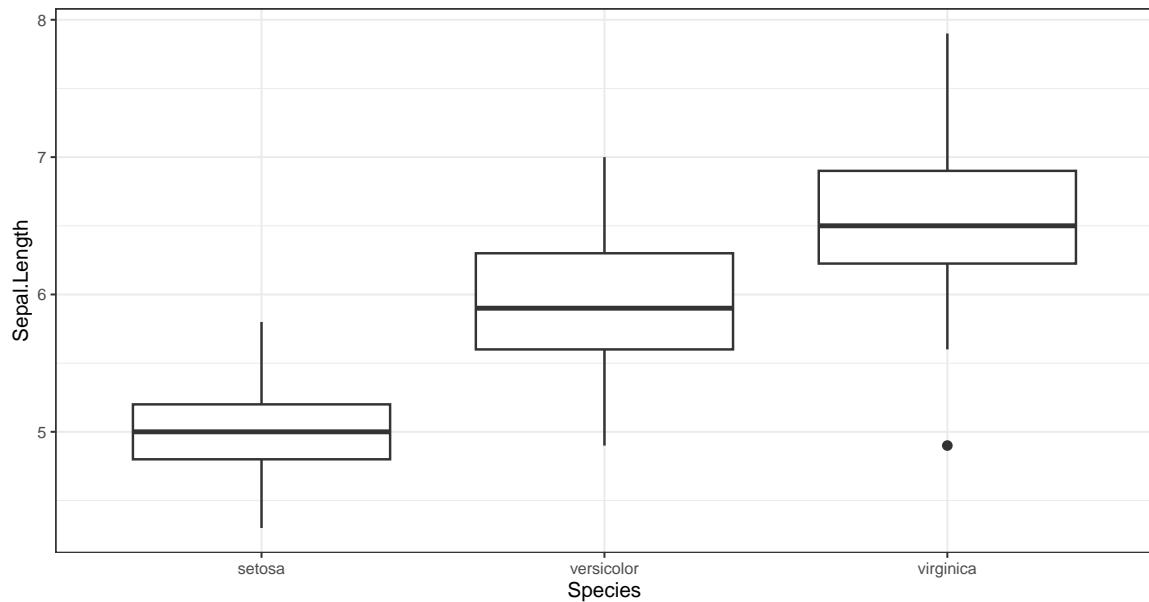
## Visualisation

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



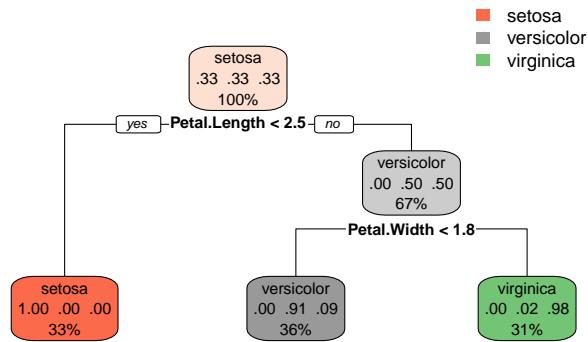
### Visualisation avec ggplot2

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



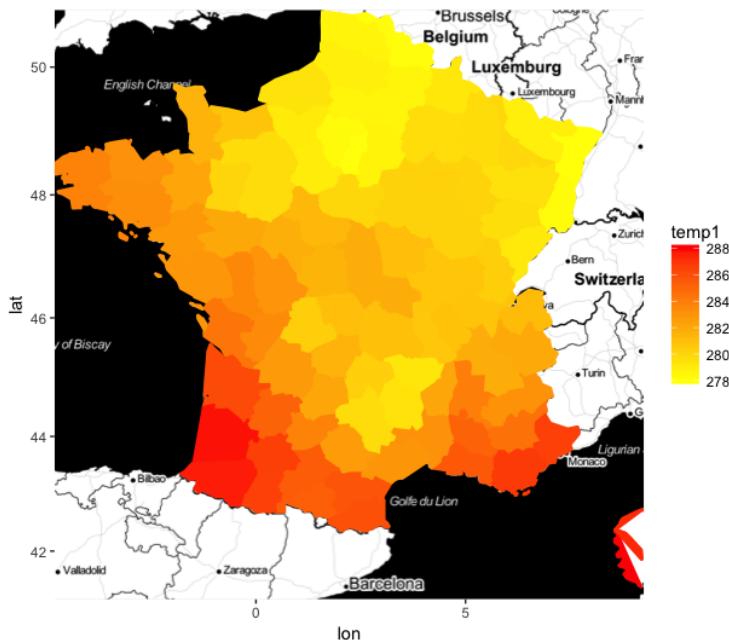
## Un modèle d'arbre

```
> 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.

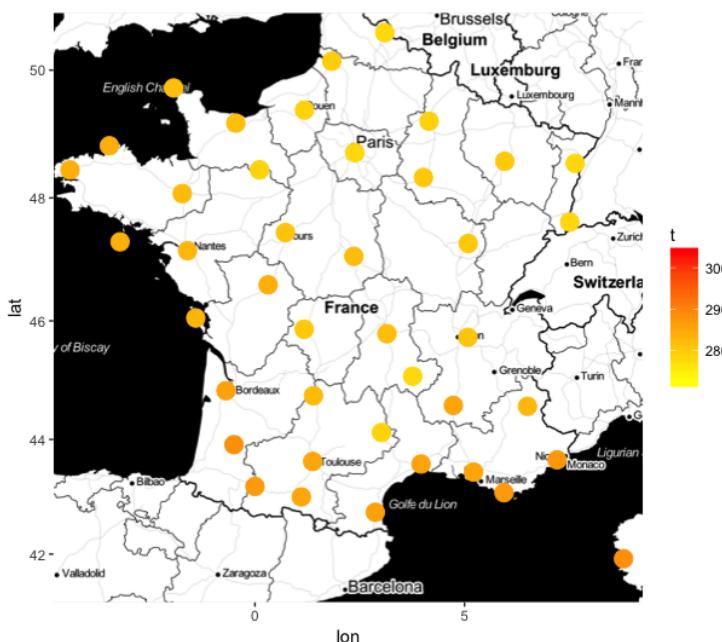


## Chargement des données + fond de carte

- 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

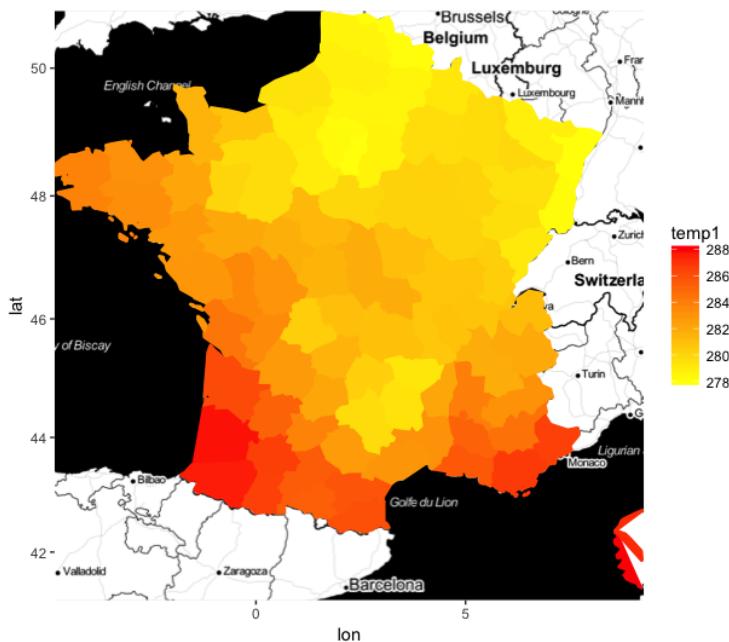
- Algorithme de *plus proche voisins* pour estimer la température sur toutes les longitudes et latitudes du territoires.

```
> 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.
- *Exemples :*
  - Graphiques descriptifs pour un jeu de données : [https://lrouviere.shinyapps.io/DESC\\_APP/](https://lrouviere.shinyapps.io/DESC_APP/)
  - Visualisation des *stations velib* à Rennes : <https://lrouviere.shinyapps.io/velib/>

## Dans cette partie

- 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

## 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 packages 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*.

## 3 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*  $\implies$  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
```

$\implies$  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 **differents 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 tibbles*

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

## 4 Gérer des données

### 4.1 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` :
 

```
> setwd("~/lectureR/Part1")
> df <- read_csv("data.csv",...)
```
- Changement du répertoire de travail
- 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_rrownames(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).

## 4.2 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
```

## 4.3 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*  $\Rightarrow$  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"
```

- $\Rightarrow$  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 3.40.1 [/Users/laurent/Google Drive/LAURENT/COURS/SNS/VISU/SLIDES/data/LEveloSTAR.sqlite]
# ... with more rows
```

	id	nom	latitude	longitude	etat
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

- 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)
# A tibble: 83 x 5
# ... with 73 more rows
```

	id	nom	latitude	longitude	etat
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

## API et JSON

- *JavaS*cript *O*bject *N*otation.
- 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 des espaces : `&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=nombremplacementsactuels",
+   "&facet=nombremplacementsdisponibles",
+   "&facet=nombrevelosdisponibles"
+ )
```

```
> ll <- jsonlite::fromJSON(url)
> tbl <- ll$records$fields |> as_tibble()
> tbl |> select(3:5)
# A tibble: 10 x 3
  nom          nombremplacementsactuels idstation
  <chr>                <int> <chr>
1 Sainte-Anne            24  5505
2 Saint-Georges Piscine    18  5509
3 Musée Beaux-Arts         16  5510
4 Bonnets Rouges           24  5514
5 Charles de Gaulle        24  5517
6 Colombier                  24  5519
```

7	Pont de Nantes	20	5520
8	Oberthur	29	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

## 4.4 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.

## **4.5 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 la 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          5.1         3.5          1.4         0.2  versicolor
2          4.9         3.0          1.4         0.3  versicolor
3          4.7         3.2          1.3         0.2  versicolor
4          4.6         3.1          1.5         0.2  versicolor
5          4.8         3.4          1.6         0.2  versicolor
6          4.8         3.0          1.4         0.2  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          1.4         0.2  setosa
2          4.4         3.4          1.3         0.2  setosa
3          4.3         3.0          1.1         0.1  setosa
4          4.4         3.2          1.3         0.2  setosa
5          4.3         3.1          1.5         0.1  setosa
6          4.4         3.0          1.4         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 5.1 1.6 3.5
#> #> 2 4.9 1.4 3.5
#> #> 3 4.7 1.4 3.3
#> #> 4 4.6 1.4 3.2
#> #> 5 5.0 1.5 3.5
#> #> 6 5.4 1.7 3.7

```

## 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
```

## 4.6 Quelques fonctions utiles de **tidyverse**

### Le package **tidyverse**

- 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
```

## 5 Visualiser des données

### 5.1 Graphes conventionnels

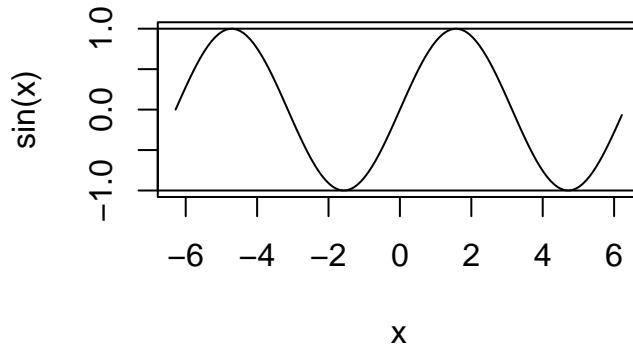
- *Visualisation* : cruciale à **toutes les étapes** d'une étude statistique.

- **R** Permet de créer un très grand nombre de type de graphes.
- On propose une (courte) présentation des *graphes classiques*,
- suivie par les graphes *ggplot*.

### 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

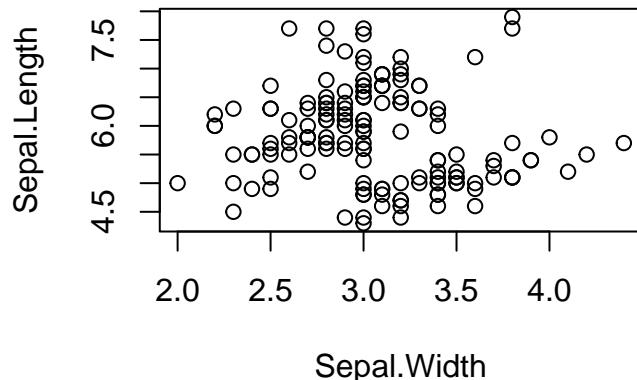
- **Histogramme** pour une variable *continue*, **diagramme en barre** pour une variable *qualitative*.
- **NUAGE DE POINTS** pour 2 variables continues.
- **Boxplot** pour une distribution continue.

### *Constat (positif)*

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

## Nuage de points sur un jeu de données

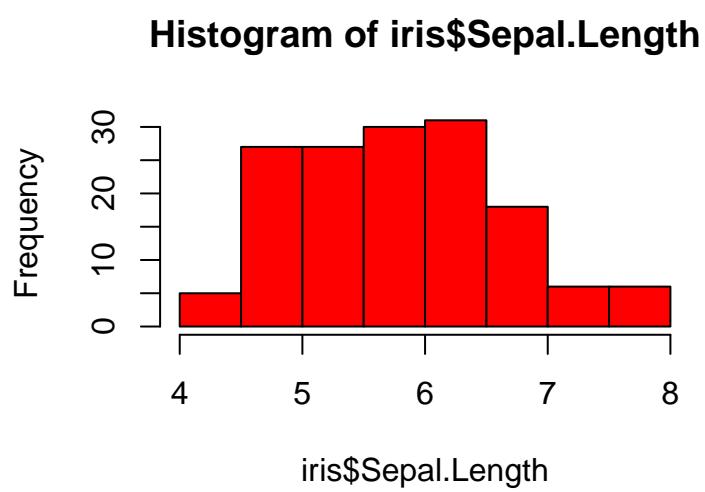
```
> plot(Sepal.Length~Sepal.Width,data=iris)
```



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

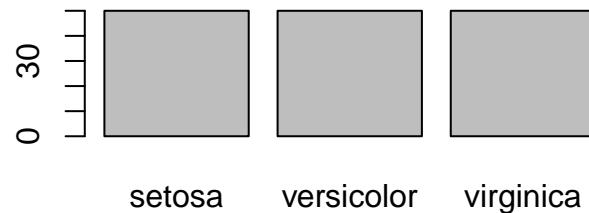
## Histogramme (variable continue)

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



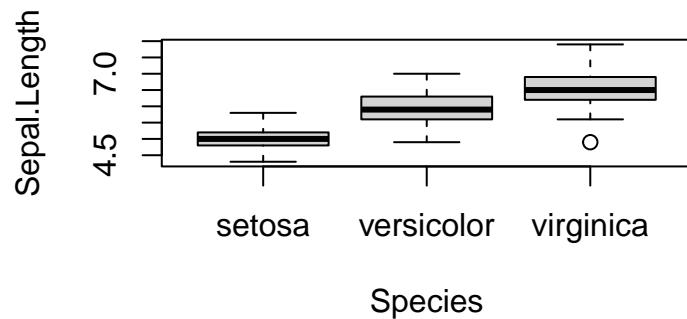
## Diagramme en barres (variable qualitative)

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



## Boxplot (distribution)

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



## 5.2 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 fait 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**

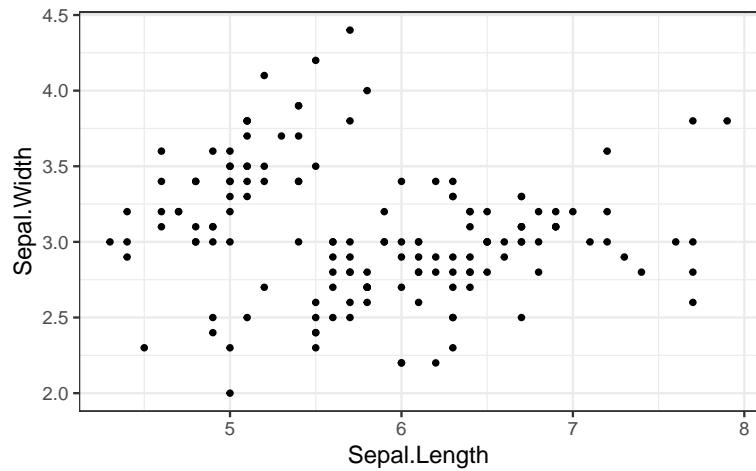
Les principaux *verbes* sont

- **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 **+**.

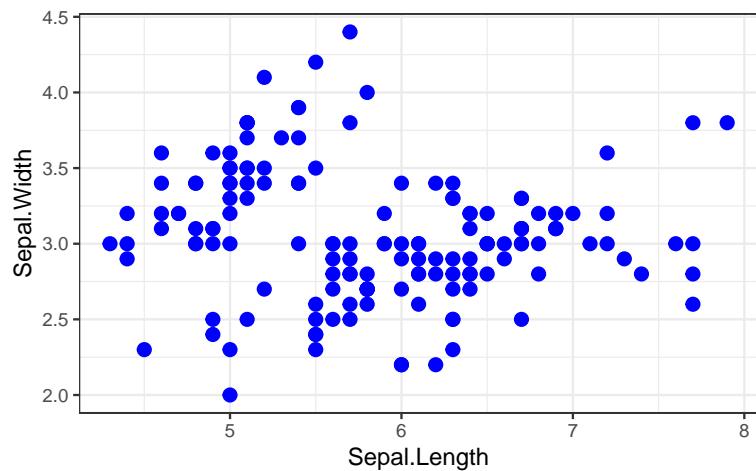
## Un premier exemple

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



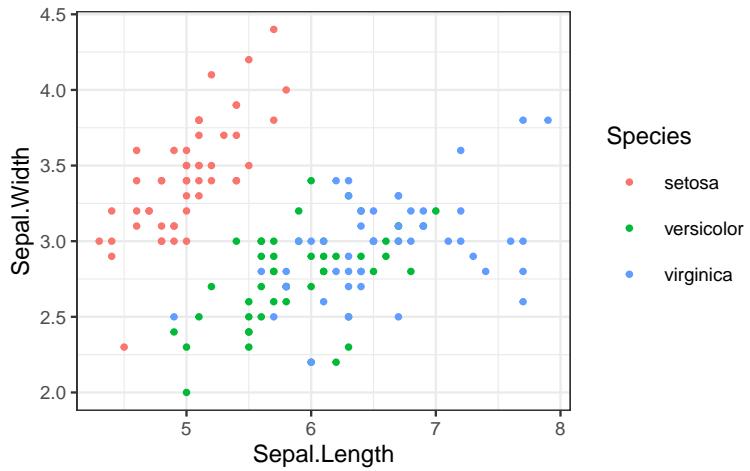
## Couleur et taille

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



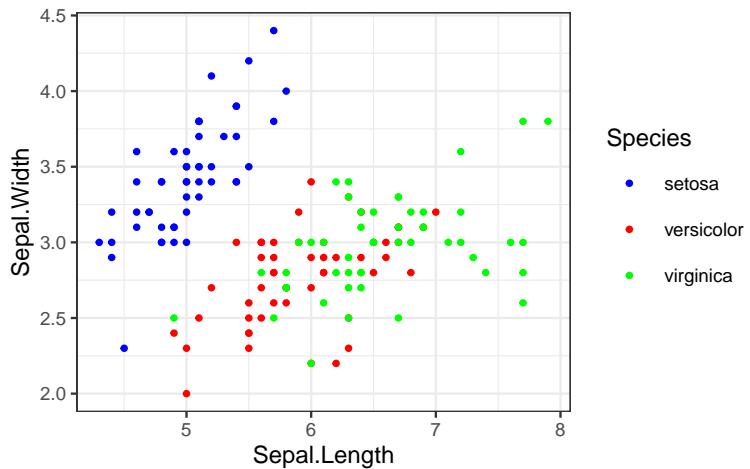
## Couleur avec une variable qualitative

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



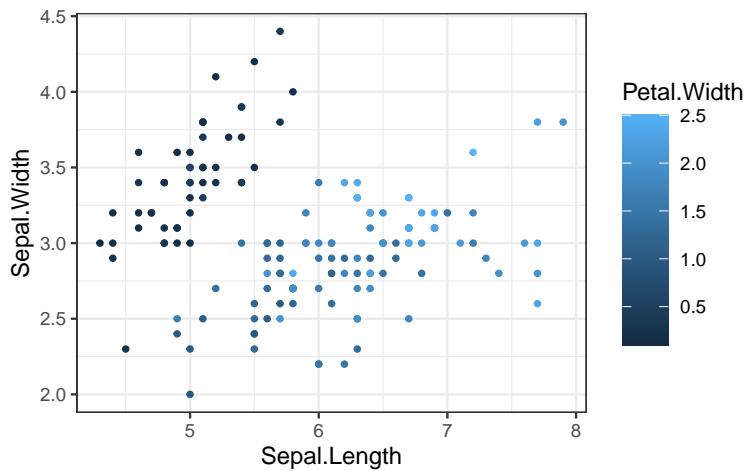
## Changer la couleur

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



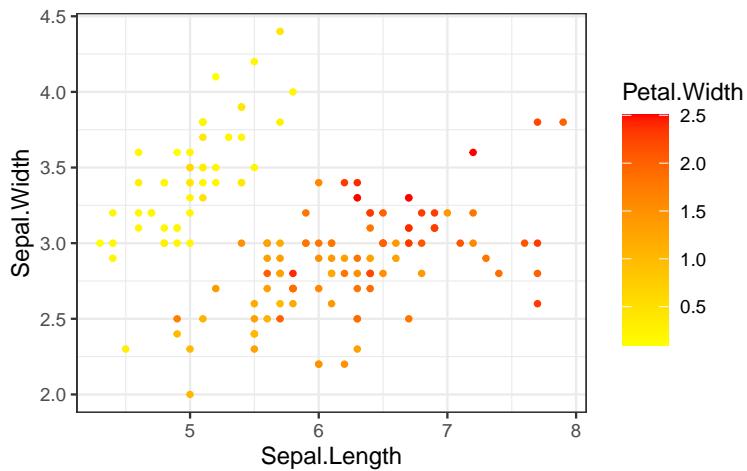
## Couleur avec une variable continue

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



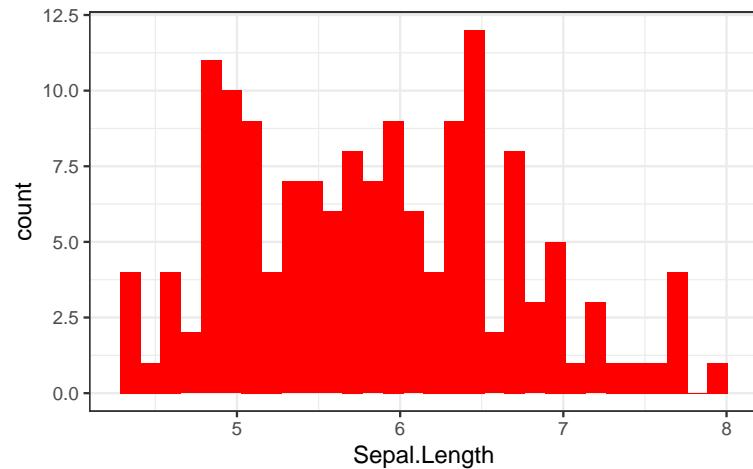
## Changer la couleur

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



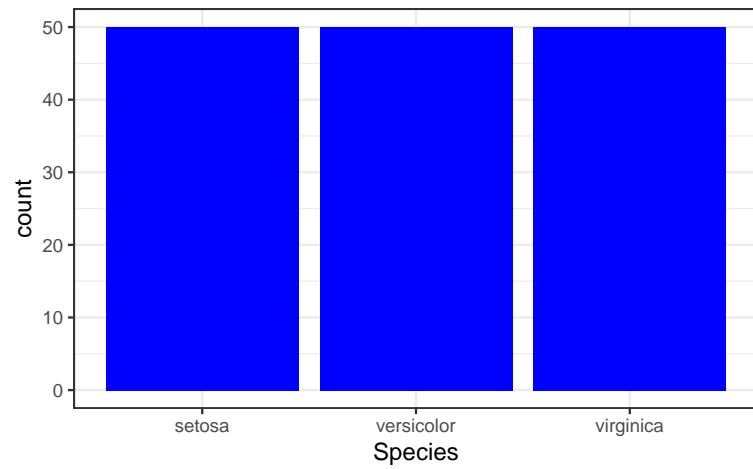
## Histogramme

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



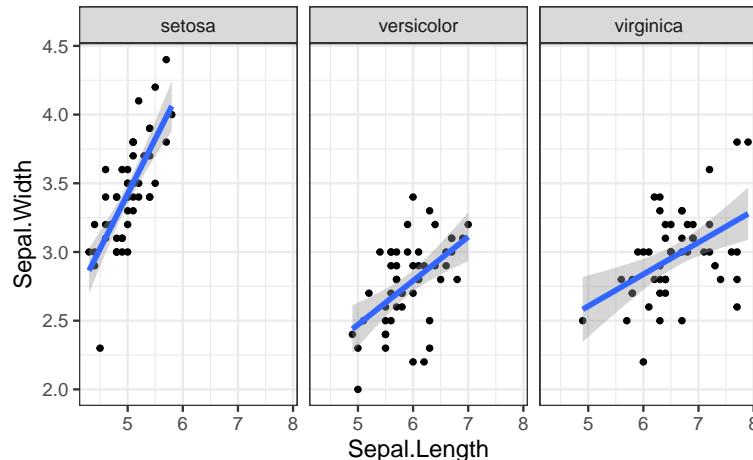
## Diagramme en barres

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



## Facetting (plus compliqué)

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



## Combiner **ggplot2** et **dplyr-tidyr**

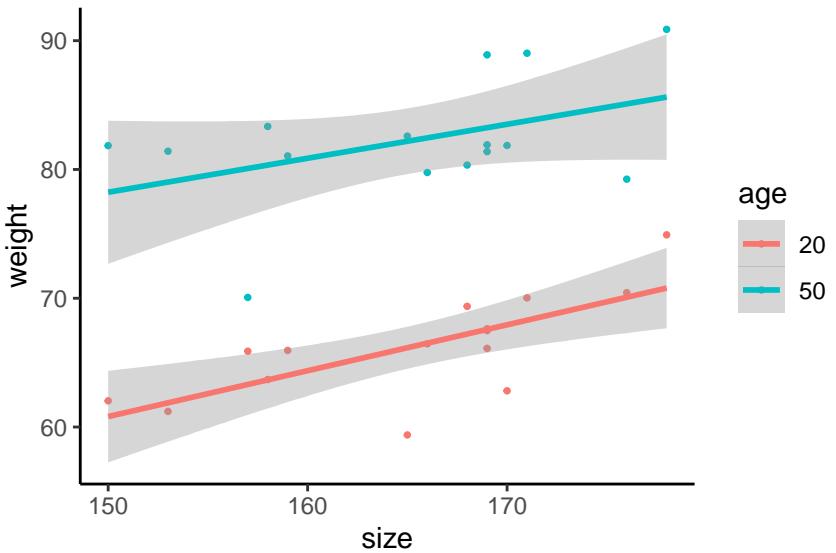
- Souvent important de *construire un bon jeu de données* pour obtenir **un bon graphe**.
- Par exemple

```
> head(df)
# A tibble: 6 x 3
  size weight.M weight.W
  <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**

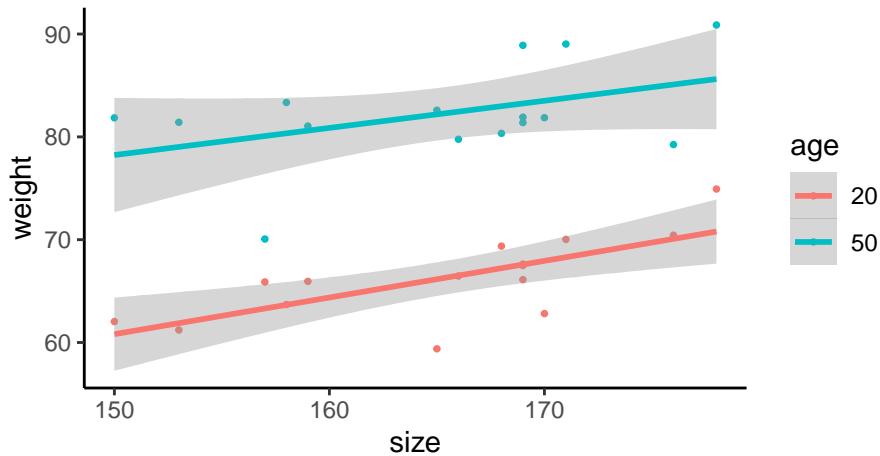
- Assembler les colonnes **weight.M** et **weight.W** en une colonne **weight** :



```
> 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.

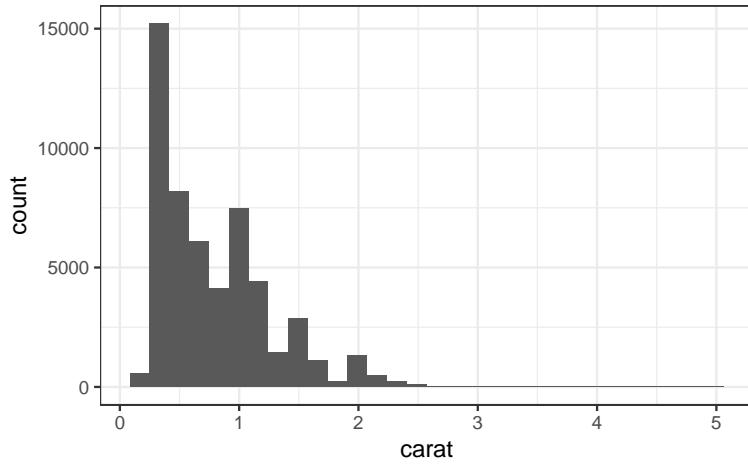
```
> geom_histogram(..., stat = "bin", ...)

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

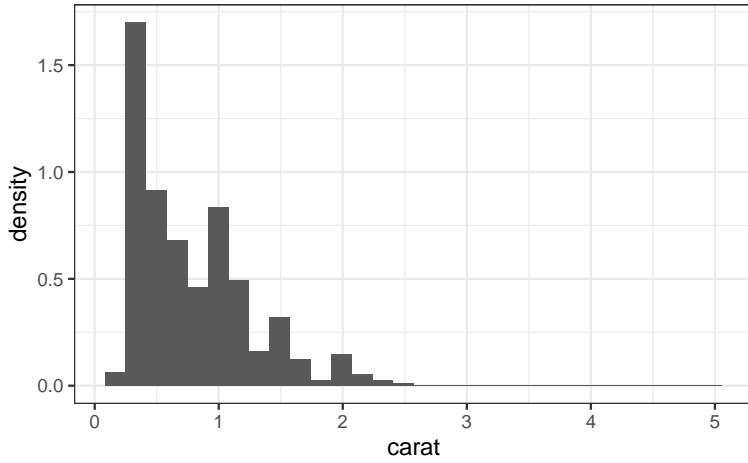
density
density of points in bin, scaled to integrate to 1
...
```

### Visualiser une autre statistique

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



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

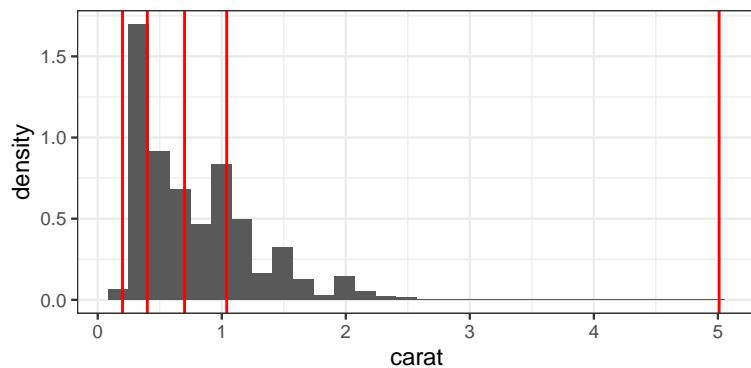


```
> #ou
> #ggplot(diamonds)+aes(x=carat,
> #                           y=..density..)+
> #   geom_histogram()
```

### **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")
```

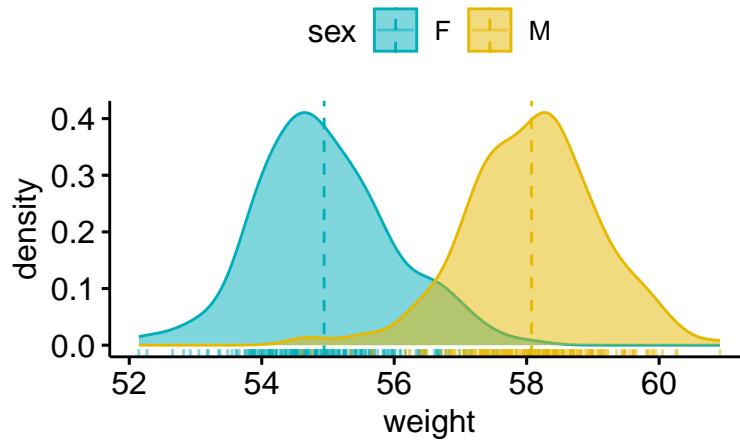


## ggpubr

- Permet de faire des graphes ggplot relativement simples avec une *yntaxe 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)
```

## 6 Programmer en R

### 6.1 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 {
+   expre2
+   ...
+ }

```

- *Exemple :*

```

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

```

## switch

- *Syntaxe :*

- *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
```

## 6.2 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
  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** :

```
> tb12 <- tibble(age=runif(5,20,50),taille=runif(5,150,180))
> map2_dfc(tb1,tb12,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(tb1,tb12,~mean(rbind(.x,.y)))
# A tibble: 1 x 2
  age   taille
  <dbl>  <dbl>
1 36.7   162.
```

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

```
> map2_dfc(tb1,tb12,\(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** :

```
> tb1 %>% .[1,1]
+ tb1 |> .[1,1]
Error: function '[' not supported in RHS call of a pipe
```

- 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
```

## 7 Cartes

### Introduction

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

### 7.1 ggmap

#### Syntaxe

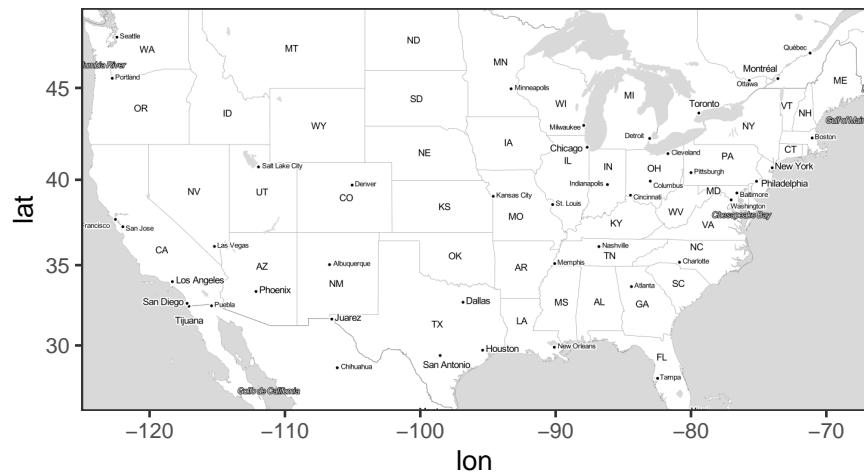
- Proche de *ggplot*...
- Au lieu de  

```
> ggplot(data)+...
```
- on utilise  

```
> ggmap(backgroundmap)+...
```

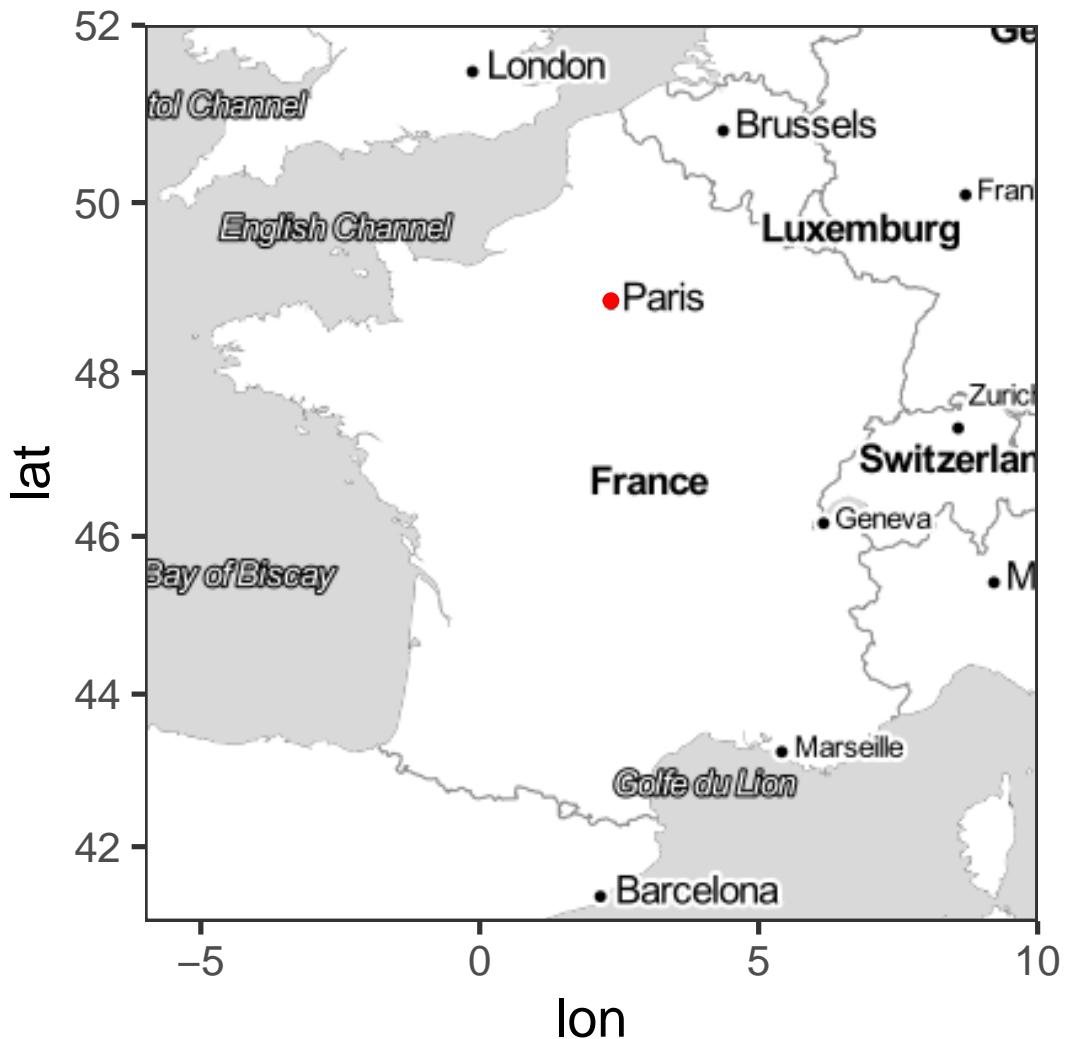
#### Fonds de carte ggmap

```
> library(ggmap)
> us <- c(left = -125, bottom = 25.75, right = -67, top = 49)
> map <- get_stamenmap(us, zoom = 5, maptype = "toner-lite")
> ggmap(map)
```



## Ajouts avec ggplot

```
> fr <- c(left = -6, bottom = 41, right = 10, top = 52)
> fond <- get_stamenmap(fr, zoom = 5,"toner-lite")
> Paris <- data.frame(lon=2.351499,lat=48.85661)
> ggmap(fond)+geom_point(data=Paris,aes(x=lon,y=lat),color="red")
```



## 7.2 Contours shapefile contours avec sf

### Le package sf

- *Ggmap* : bien pour des cartes “simples” (fond et quelques points).
- *Pas suffisant* pour des **représentations plus complexes** (colorier des pays à partir de variables).

- `sf` 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 `ggplot` (verbe `geom_sf`).

## Références

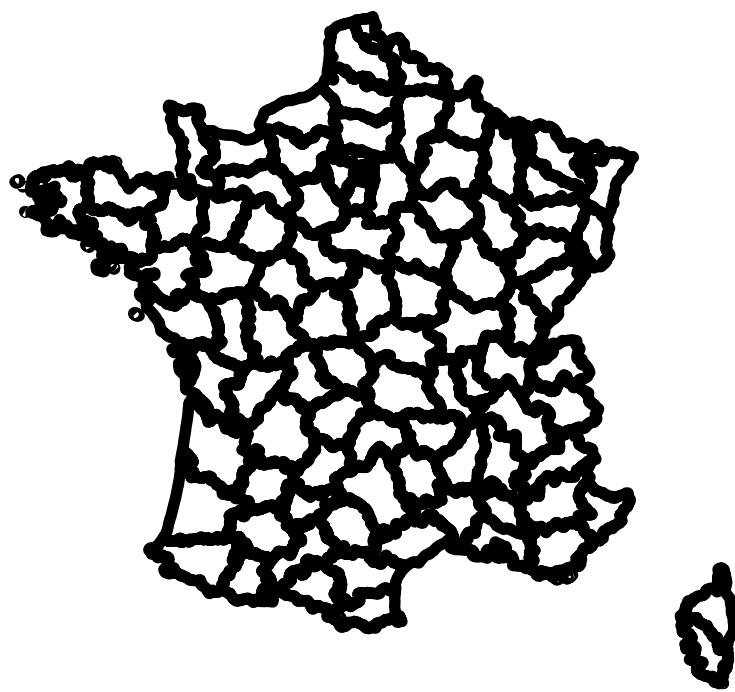
- <https://statnmap.com/fr/2018-07-14-initiation-a-la-cartographie-avec-sf-et-compagnie/>
- Vignettes sur le cran : <https://cran.r-project.org/web/packages/sf/index.html>
- Un *tutoriel* très complet (un peu technique) : <https://r-spatial.github.io/sf/articles/>

## Exemple

```
> 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, ~
```

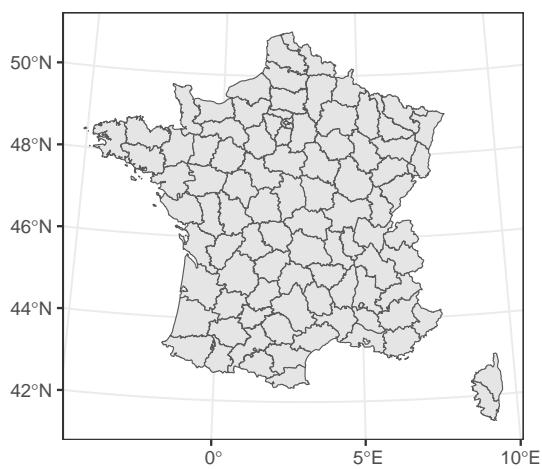
## Visualisation avec plot

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



### Visualisation ggplot

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



## Ajouter des points sur le graphe

- Définir des coordonnées avec *st\_point*

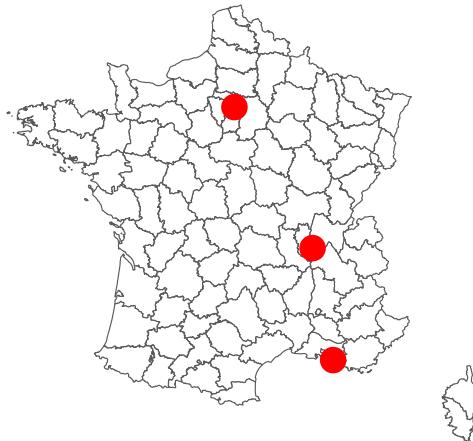
```
> point <- st_sf(st_point(c(2.351462,48.85670)),  
+                  st_point(c(4.832011,45.75781)),  
+                  st_point(c(5.369953,43.29617)))
```

- 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)
```

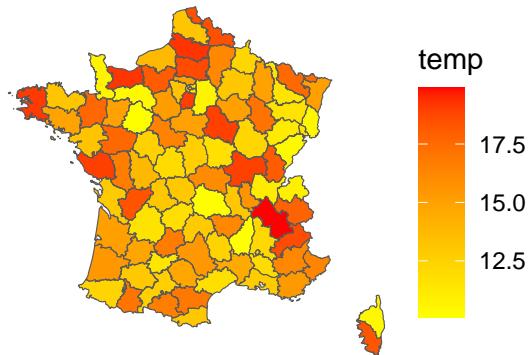
## Étape ggplot

```
> 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.
- 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 `sf` (simple feature)

```

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

```

- Création d'un objet **sfc** ("liste" d'objets sf)

```

> b2 <- st_sf(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:* here

```
> 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("Stamen.Toner") |>
+   setView(lng = Paris[1], lat = Paris[2], zoom = 12)
```



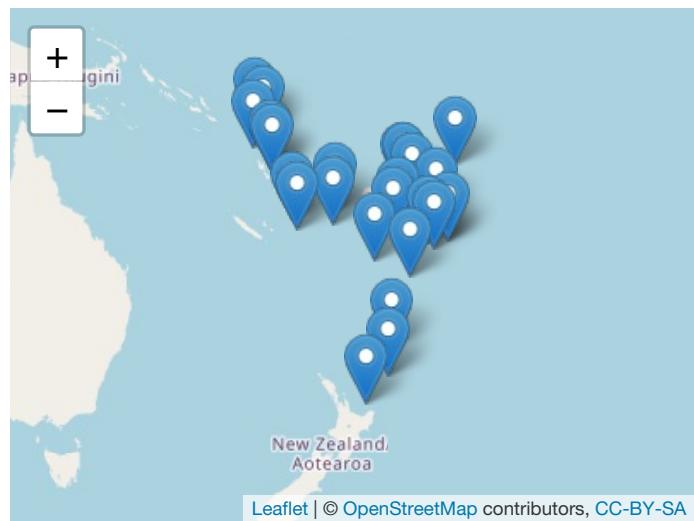
## Avec des données

- 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))
```

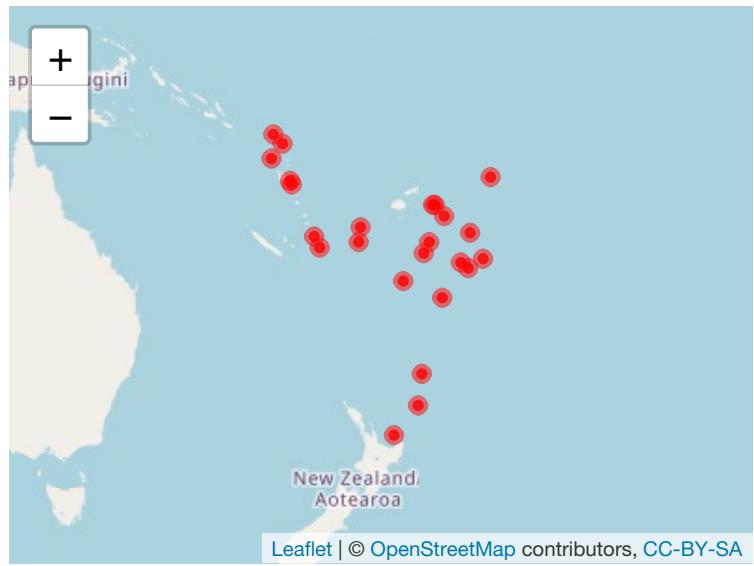


### *Remarque*

La magnitude apparaît lorsqu'on clique sur un marker.

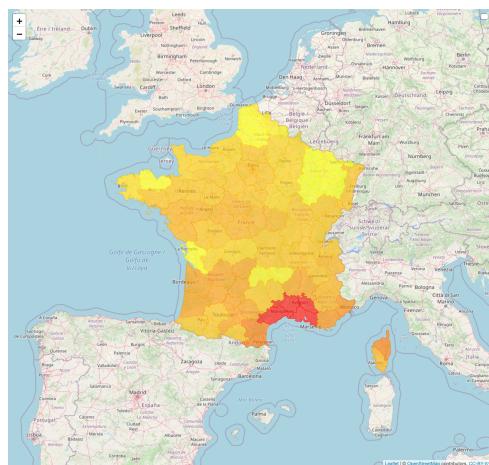
### **addCircleMarkers**

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



### Colorier polygones en combinant leaflet et sf

```
> leaflet() |> addTiles() |>
+   addPolygons(data = dpt2,color=~pal1(t_prev),fillOpacity = 0.6,
+               stroke = TRUE,weight=1,
+               popup=~paste(as.character(NOM_DEPT),
+                           as.character(t_prev),sep=" : "))
```



## 8 Quelques outils de visualisation dynamiques

### Des packages R

- Graphiques classiques avec *rAmCharts* et *plotly*.
- Graphes avec *visNetwork*.
- Tableaux de bord avec *flexdashboard*.

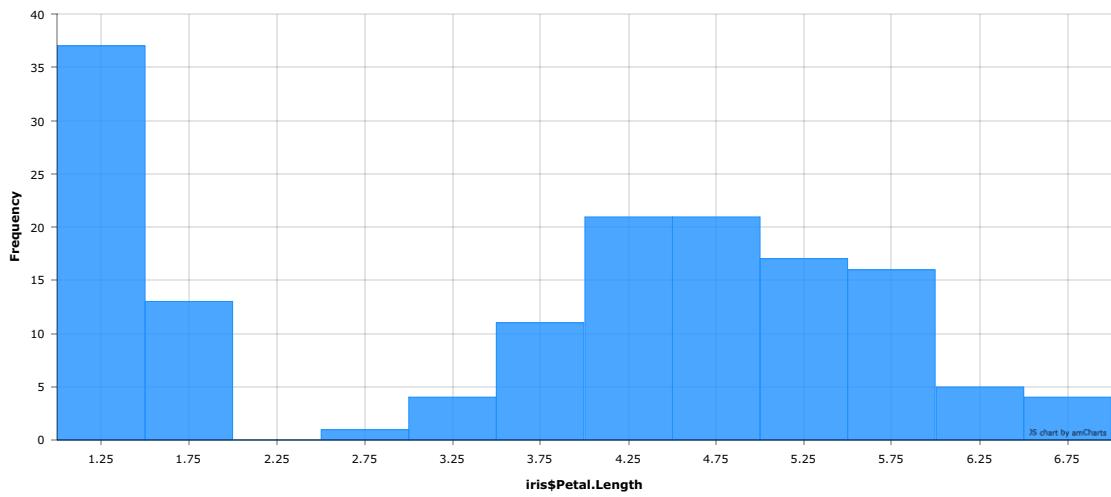
#### 8.1 rAmCharts et plotly

##### rAmCharts

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

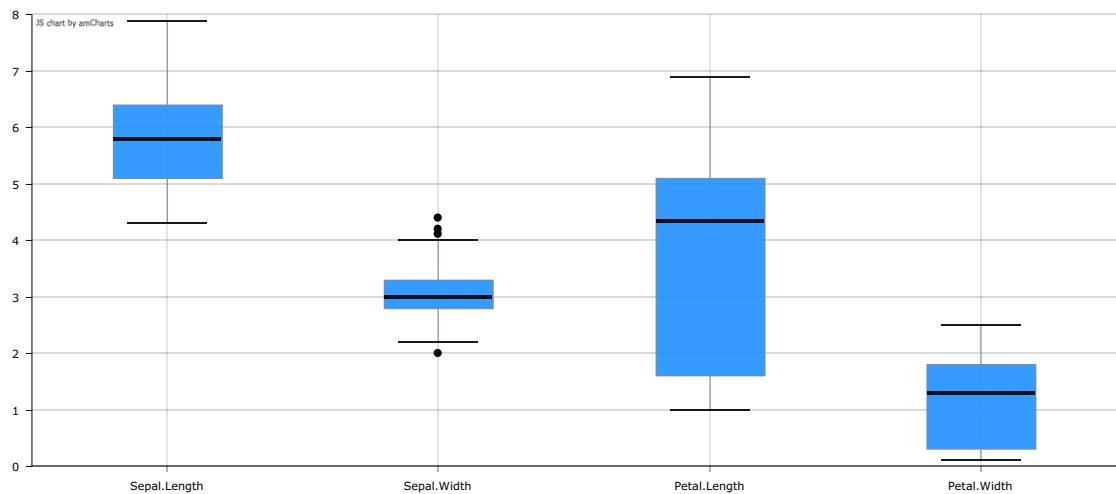
##### rAmCharts Histogramme

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



## rAmcharts Boxplot

```
> amBoxplot(iris)
```



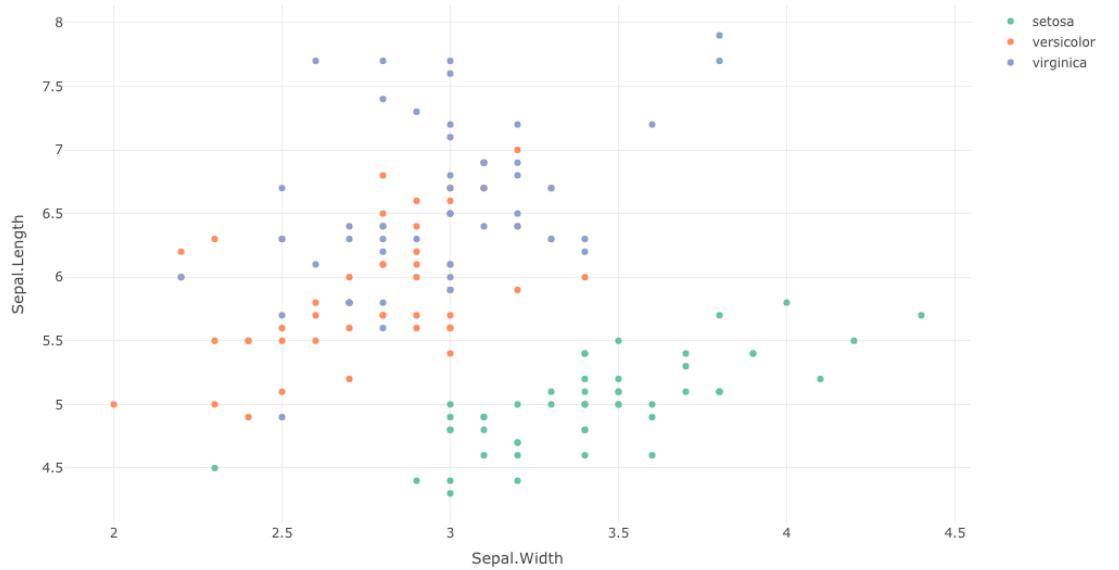
## Plotly

- Package **R** pour créer des *graphes interactifs* 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/>

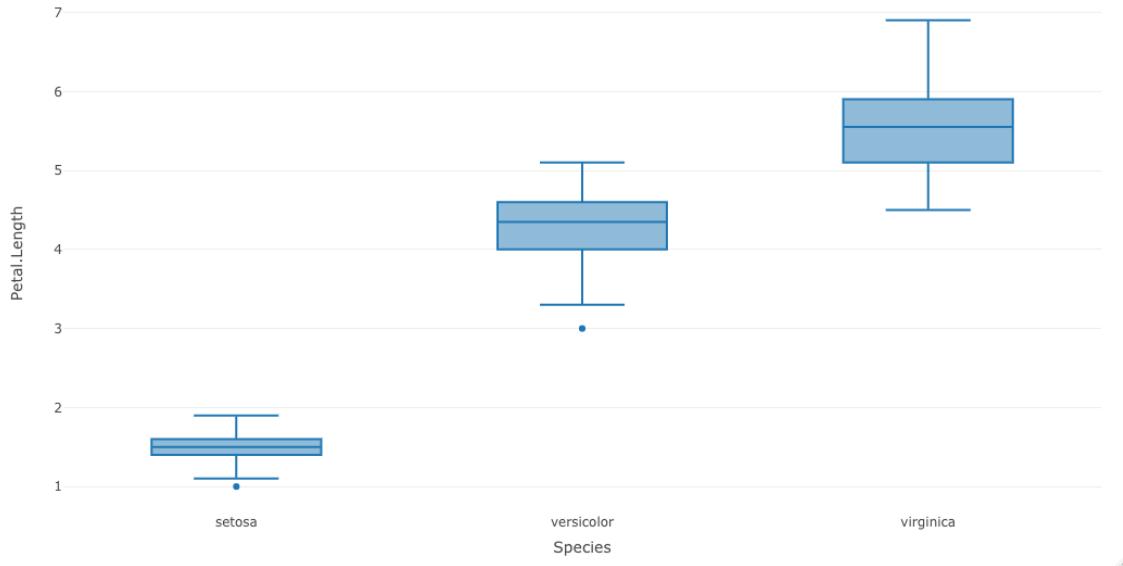
## Nuage de points

```
> 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()
```



## 8.2 Graphes avec visNetwork

### Connexions entre individus

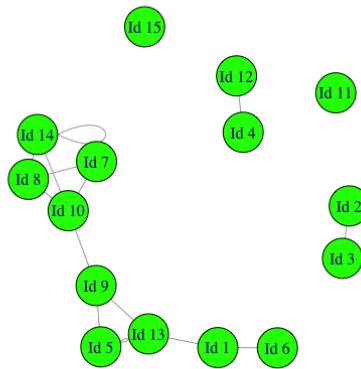
- De nombreux jeux de données peuvent être visualisés avec des *graphes*, notamment lorsque l'on souhaite étudier des *connexions* entre individus (génétique, réseaux sociaux, système de recommandation...)
- Un individu = *un nœud* et une connexion = *une arête*.

```
> 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>

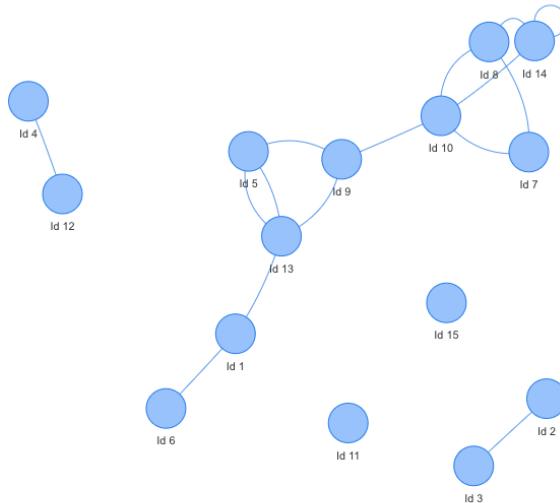
```
> library(igraph)
> net <- graph_from_data_frame(d=edges, vertices=nodes, directed=F)
> plot(net,vertex.color="green",vertex.size=25)
```



## Graph dynamique : le package visNetwork

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

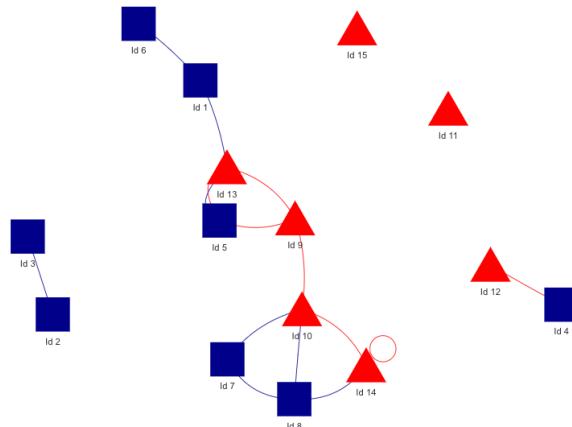
```
> library(visNetwork)
> visNetwork(nodes,edges)
```



...inf

## Nodes color

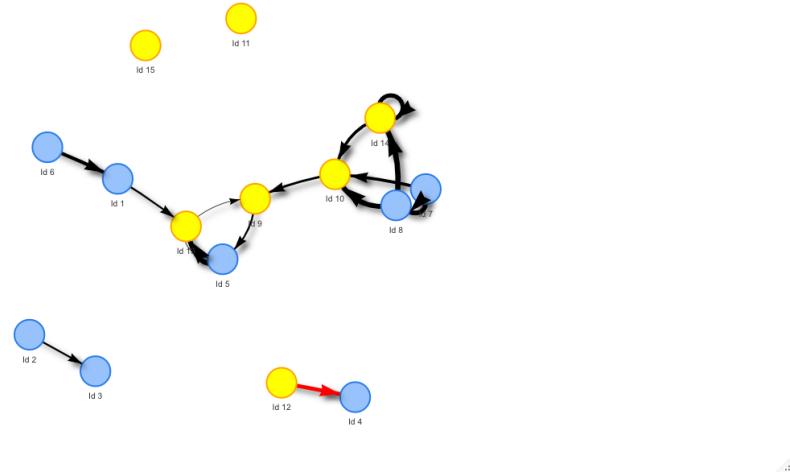
```
> nodes$group <- c(rep("A",8),rep("B",7))
> visNetwork(nodes,edges) |>
+   visGroups(groupname = "A", color = "darkblue",
+             shape = "square") |>
+   visGroups(groupname = "B", color = "red",
+             shape = "triangle")
```



...inf

## 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

