R avancé

Laurent Rouvière

janvier 2024

Table des matières

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).
  - Co-responsable du Master MAS et responsable du parcours SDD-IA.
  - Consulting : énergie (ERDF), finance, marketing.

- *Prérequis* : niveau avancé en **R** - bases en statistique et programmation
- *Objectifs* :
  - Maîtriser les outils **R** modernes pour manipuler les données
  - Appréhender les environnements standards des bases de données dans **R**
  - Scrapper des données
  - Mettre en oeuvre des pipelines en modélisation/Machine Learning

Plan du cours

1. *Lire* et *manipuler* des données dans le tidyverse avec readr et dplyr
2. *Bases de données* SQL, JSON et API avec DBI et jsonlite
3. *Webscrapping* avec rvest
4. Créer des *pipelines de Machine Learning* avec tidymodels

2 Gérer des données

2.1 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 les 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).

2.2 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
```

2.3 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.sqli
      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

- *J*ava*S*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 esperluettes & : `&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                                 23 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                                    28 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

2.4 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.

2.5 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
```

2.6 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
```

3 Bases de données

3.1 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
MysSQL	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)
```

3.2 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*.

3.3 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.

3.4 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)
```

3.5 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 le types structurés, les différentes valeurs peuvent être de n'importe quel type, y compris un autre type structuré.
- Pour vérifier la validité d'un fichiers 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
```

3.6 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                      6
2  Saint-Georges Piscine       -1.674417        48.11239                     10
3       Musée Beaux-Arts       -1.674080        48.10960                      9
4         Bonnets Rouges       -1.665814        48.10685                     13
5      Charles de Gaulle       -1.677119        48.10511                     15
6              Colombier       -1.680594        48.10610                      6
7         Pont de Nantes       -1.684015        48.10202                      8
8               Oberthur       -1.661853        48.11355                     12
9    Auberge de Jeunesse       -1.681876        48.12088                     11
10    Croix Saint-Hélier       -1.662090        48.10305                      3
```

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
```