Module 2 - Premières manipulations et analyses de données

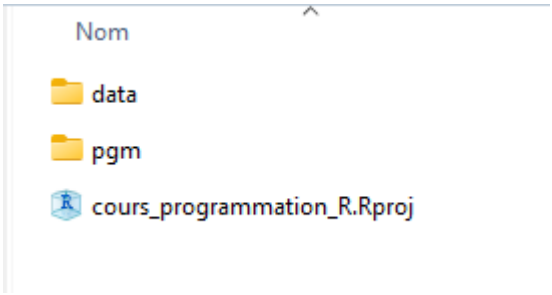
Dans ce module, nous allons apprendre à charger, manipuler et faire des analyses simples de données. Nous allons le faire avec les commandes basiques de R puis en utilisant deux librairies plébiscitées par les data-analysts, `dplyr` et `ggplot2`.

1 Travailler avec les projets R

Il est recommandé d'utiliser systématiquement les projets RStudio pour mener des projets statistiques avec R, car ils contribuent à rendre les traitements reproductibles. Le principe d'un projet RStudio est de rassembler tous les éléments de contexte propres à ce projet : espace de travail, historique de commandes, variables d'environnement, options de R Il permet aussi d'assurer un suivi précis des modifications de ces programmes (contrôle de version en association avec Git - non abordé ici).

L'avantage majeur d'un projet R est qu'il *portable*, facilement transmissible d'un utilisateur à un autre en rendant les scripts indépendant de l'arborescence de la machine : si vous avez un code `traitement.R` situé dans le même dossier que le fichier `.Rproj`, alors le chemin de ce code est `./traitement.R`, où que soit situé le dossier du projet.

En pratique, un projet RStudio prend la forme d'un fichier `.Rproj`, qui est placé habituellement dans le dossier qui contient les programmes. Pour créer un projet, il suffit de faire `File > New Project`. Créer un nouveau projet `cours_programmation_R.Rproj`. On peut organiser l'arborescence de ce projet avec un dossier pour les programmes, un dossier pour les données. Plus le projet est complexe, plus cette arborescence est à travailler (un dossier pour les sorties graphiques, un dossier pour les résultats, un dossier pour les tables intermédiaires...).



À noter qu'il est fortement conseillé de *désactiver* la sauvegarde du fichier `.RData` à la fermeture du projet. Pour ce faire, il faut aller dans `Tools > Global Options` et rester dans l'onglet `General`. Chercher `Save workspace to .RData on exit:`, et choisir l'option `Never`. De manière

générale, ne jamais sauvegarder l'espace de travail en .RData à la fermeture d'une sesion !

Au sein de chaque programme R, une bonne pratique est de commencer par charger toutes les librairies utiles pour la suite du programme, puis de dérouler les opérations dans un ordre logique, en les regroupant par type et en les commentant. Par exemple : import des données, manipulations, calculs, export des résultats.

2 Chargement de données

Vous pouvez créer un premier fichier R `Module2.R`.
Plusieurs commandes permettent de charger des données. Il est également possible dans le quadrant en haut à droit d'importer des datasets. En général les commandes sont du type `read.table` ou `read.csv`. Voici un exemple ci-dessous avec le fichier `PisaUS.csv` qui recensent des notes scolaires d'élèves

```
rm(list=ls()) # supprime tous les objets de l'environnement
gc() # garbage collector : force la libération de la mémoire

pisa_us<-read.table("PisaUS.csv",header = TRUE, sep=";",dec=",",
na="␣")
# lire les donnees

class(pisa_us)

head(pisa_us) # pour voir les premières lignes du data.frame()
```

3 Manipulation de données

Les données que l'on charge sont de type `data-frame` (appelé aussi tableau de donneés).
Une façon de bien comprendre la structure de nos données est d'utiliser la commande `str`

```
help(str)

str(pisa_us) #donne la structure de notre data-frame
```

Pour accéder à une des colonnes qui correspond à une variable étudiée on utilisera le caractère $ comme le montre l'exemple ci-dessous :

```
summary(pisa_us$MATH)
```

On peut aussi accéder aux colonnes/lignes grâce aux crochets []

```
pisa_us[2,4] # deuxieme ligne, quatrieme colonne
pisa_us[6,3]<-12 #pour changer une donnee ponctuelle

pisa_us<-pisa_us[-2,] # pour supprimer la deuxieme ligne
pisa_us<-pisa_us[,-6] # pour supprimer la sixième colonne

pisa_us<-pisa_us[pisa_us[,3]>600,] # pour supprimer les lignes dont
# la valeur en 3eme colonne est inferieure ou egale a 600

pisa_us$GLCM <-NULL # pour supprimer une colonne correspondant
# a une variable
```

Pour ajouter des lignes (respectivement des colonnes) à un data-frame, nous pouvons utiliser la commande `rbind` (respectivement `cbind`).

```
#Utilisation de cbind

twocol<-data.frame(matrix(0,4838,2))
names(twocol)<-c("colonne1","colonne2")
twocol

rescol<-cbind(pisa_us,twocol)

rescol

#Utilisation de rbind

twolines<-data.frame(matrix(0,2,4))
names(twolines)<-names(pisa_us)
```

```
resline<-rbind(pisa_us,twolines)

resline
```

On peut également fusionner deux datasets en utilisant la commande `merge`. Il faut préciser une colonne commune pour fusionner les deux datasets comme on le voit avec l'exemple ci-dessous.

```
#Utilisation de merge
pisa_bis<-pisa_us[,c(1,3)]

names(pisa_bis)<-c("X","MATHBIS")
pisa_bis
pisa_bis$MATHBIS<-(pisa_bis$MATHBIS)*1.2

pisa_new<-merge(pisa_us,pisa_bis, by="X")
```

4 Manipuler les données avec `dplyr`

Créons un deuxième fichier R intitulé `Module2_dplyr_ggplot.R`.
Un package a été spécifiquement conçu pour la manipulation des tableaux de données. Il s'agit du package `dplyr`. Il facilite certaines actions sur les tableaux de données et est également efficace en termes de temps d'exécution. Dans cette section, nous allons voir les commandes essentielles de `dplyr`, notamment les principaux "verbes" ainsi que le `pipe`. Nous chargeons les packages `dplyr` et `tidyverse` qui englobe notamment `ggplot2`.
Dans cette section, nous utiliserons des données sur les états américains en 1975.

```
rm(list=ls())

install.packages("tidyverse")
install.packages("dplyr")
```

```
library(tidyverse)
library(dplyr)

df<-as.data.frame(state.x77)
```

Le verbe `slice` permet de sélectionner certaines lignes d'un tableau. On peut également le faire avec le verbe `filter` qui permet de sélectionner des lignes qui vérifient certaines conditions comme on le voit avec l'exemple ci-dessous.

```
#le verbe slice

slice(df,48)

slice(df,44:48)

#le verbe filter

filter(df, Area > 24000)

filter(df, Area > 24000 & Area <= 32000)
```

Le verbe `select` permet de sélectionner des colonnes/variables. Le verbe `relocate` permet de changer l'ordre des variables. Le verbe `rename` permet de renommer les variables.

```
#le verbe select

select(df, Population, Area)

select(df, -Population, -Area)

select(df, Population:Murder)

select(df, where(is.numeric))

#le verbe relocate (ordonner les variables)
relocate(df, Area, Life Exp) #erreur attention aux noms avec espace
```

```
relocate(df, Area, "Life␣Exp")

#le verbe rename

rename(df, "LifeExp" = "Life␣Exp")
```

On peut également réordonner les lignes selon certaines variables avec le verbe `arrange`. Par exemple on peut sélectionner les 3 plus petits états américains en combinant les commandes `arrange` et `slice`.

```
#le verbe arrange

arrange(df, Area) #ordre croissant
arrange(df, desc(Area)) #ordre decroissant

#combiner arrange et slice

slice(arrange(df, Area),1:3)
```

Le verbe `mutate` permet de créer des variables

```
#le verbe mutate

mutate(df,Areakm=Area/0.38610)
```

Attardons-nous à présent sur l'outil `pipe`. Cet outil permet d'enchaîner les commandes sur un même data-frame et de faire ce que l'on appelle un `pipeline`. Le symbole %>% peut se lire "puis". Voici un exemple ci-dessous.

```
#outil pipe pour un pipeline

df %>% filter(Population >5000)

df%>%mutate(Areakm=Area/0.38610)%>%select(-Area)%>%
filter(Areakm>400000)
```

```

```

Une commande importante du package `dplyr` concerne les opérations groupées : `group_by`

```
newdf<-mutate(df,HLE=case_when(`Life Exp`>72 ~ 2,`Life Exp`>70 ~ 1,
TRUE ~ 0))

newdf%>%group_by(HLE)
```

La commande `group_by` crée des groupes selon une variable. Après avoir fait cela, les verbes `slice`, `mutate` ou `filter` vont prendre en compte que les lignes sont "groupées". Par exemple pour si on exécute la commande `slice(1)` il sélectionnera la première ligne de chaque "groupe". De même si l'on utilise la commande `mutate` comme ci-dessous, la commande créera une variable qui vaut la moyenne par "groupe".

```
newdf%>%arrange(`Life Exp`)%>%group_by(HLE)%>% slice(1)

newdf %>%
  group_by(HLE) %>%
  mutate(mean_Pop = mean(Population, na.rm = TRUE)) %>%
  select(`Life Exp`, HLE, mean_Pop)
```

La fonction `count` permet de compter le nombre de lignes partageant la même modalité pour une variable donnée :

```
newdf%>%count(HLE)
```

On peut utiliser la commande `ungroup()` pour supprimer les groupes créés.
On va maintenant apprendre à fusionner des datasets avec la librairie `dplyr`. On utilisera la commande `left_join` qui fusionne en prenant comme référence le premier dataset mis en argument ("fusion en se basant sur le dataset de gauche"). D'autres fusions sont possibles avec des commandes de la famille `join` de `dplyr`. N'hésitez pas à regarder la cheat sheet dédiée à cette librairie.

```r
us_arrest<-read.table("Data/USArrests.csv",
header = TRUE, sep=";",dec=".", na="␣")

df<-df%>%mutate(State=rownames(df))

FusionDF<-left_join(df,us_arrest, by ="State")
```

À noter qu'il existe un autre package similaire à `dplyr`, `data.table`, conçu pour manipuler rapidement et efficacement des données volumineuses. `data.table` offre des performances supérieures à `dplyr` mais a une syntaxe concise, moins lisible que celle de `dplyr` comme le montre l'exemple ci-dessous.

```r
# En dplyr

# Créer un data.frame
df <- data.frame(ID = 1:5, Value = c(10, 20, 30, 40, 50))

# Filtrer les lignes
df %>% filter(Value > 20)

# Ajouter une nouvelle colonne
df %>% mutate(NewValue = Value * 2)

# Résumer par groupe
df %>% group_by(ID) %>% summarize(MeanValue = mean(Value))

# En data.table
# Structure de type [i, j, by] avec
# i : sélectionne les lignes (similaire à filter())
# j : effectue des opérations sur les colonnes (comme mutate())
# by : regroupe les données pour des calculs (équivalent à group_by())

library(data.table)

# Créer un data.table
dt <- data.table(ID = 1:5, Value = c(10, 20, 30, 40, 50))
```

```
# Filtrer les lignes
dt[Value > 20]  # Équivalent de filter()

# Ajouter une nouvelle colonne
dt[, NewValue := Value * 2]  # Équivalent de mutate()

# Résumer par groupe
dt[, .(MeanValue = mean(Value)), by = ID]
# Équivalent de summarize() + group_by()
```

5 Premières analyses de données

Revenons sur le fichier `Module2.R` Pour tester des commandes d'analyses de données, notamment des statistiques descriptives, nous allons commencer par repartir à zéro en supprimant les variables créés et en important de nouveau les données.

```
rm(list=ls())

pisa_us<-read.table("Data/PisaUS.csv",header = TRUE,
sep=";",dec=",", na="␣")
```

La commande `summary` permet de résumer quelques statistiques descriptives.

```
summary(pisa_us) # resumer les donnees
```

On peut profiter de cette occasion pour introduire la fonction `apply` qui permet d'appliquer une même fonction à toutes les colonnes (avec MARGIN=2) ou toutes les lignes (avec MARGIN=1) d'un data-frame.

```
apply(pisa_us[,3:6], MARGIN=2,mean)
```

Venons-en aux graphiques descriptifs. Nous abordons ici 3 types de graphes : les histogrammes, les boxplots (boîtes à moustache) et les plots.

```r
hist(pisa_us$MATH)

help(hist) # pour voir les differentes options

m=mean(pisa_us$MATH)
s=sd(pisa_us$MATH)

hist(pisa_us$MATH, freq=F, col="grey", main="Histogramme",
ylim=c(0,0.005), xlab = "Histogramme␣et␣approximation␣par␣une␣normale")
curve(dnorm(x,m,s), add=T, lwd=2) # ajout d'une courbe
# dnorme génère une densité d'une distribution normale
# de moyenne m et d'écart-type s.
# Cette fonction est appelée avec l'argument x, qui désigne
# la séquence de valeurs le long de l'axe des abscisses.
```

Pour retrouver les couleurs disponibles en R nous vous invitons à consulter le site `https://www.datanovia.com/en/fr/blog/liste-geniale-de-657-noms-de-couleur-dans-r/`

Pour les boîtes à moustache, on utilise la commande `boxplot`. Le boxplot affiche la médiane, le 1er et 3ème quartile $Q1$ et $Q3$ qui délimite la boîte. Les extrêmes correspondent à $Q1 - 1.5(Q3 - Q1)$ et $Q3 + 1.5(Q3 - Q1)$. Les points en dehors des extrêmes peuvent parfois être considérés comme des outliers.

```r
boxplot(pisa_us$MATH, horizontal = TRUE)

help(boxplot)

boxplot(pisa_us$READ,pisa_us$MATH,pisa_us$SCIE, horizontal = TRUE,
names=c("Read","Math","Science"), col=c("blue","red","green"))
```

Enfin nous utilisons la commande `plot` pour les graphes bivariés d'une variable y en fonction d'une variable x.

```r
plot(pisa_us$MATH,pisa_us$READ)
```

```
help(plot)

plot(pisa_us$MATH,pisa_us$READ, col="blue")
abline(lm(pisa_us$READ~pisa_us$MATH), col="red")
```

Nous pouvons profiter de ces graphes bivariés pour introduire les commandes `cor` et `pairs`, très utiles pour comprendre les relations entre nos différentes variables.

```
pairs(pisa_us[,3:5])
cor(pisa_us[,3:5])

cor(pisa_us$MATH,pisa_us$READ)
```

6 Les graphes avec `ggplot2`

Repassons sur `Module2_dplyr_ggplot.R` Une librairie qu'utilisent de nombreux data analysts/scientists est le package `ggplot2`. C'est un outil qui permet de réaliser des graphes de qualité et paramétrés finement. Dans cette section nous allons voir les principales options et quelques exemples. Commençons par charger la librairie (elle devrait être installée par le package `tidyverse`)

```
library(ggplot2) #facultatif ici
```

Dans la grammaire ggplot il y a 3 principaux éléments qui s'*additionnent* :

- Les données : `ggplot(df)` ou `df %>% ggplot()`
- L'aesthetics : `aes(x=...,y=...,color=...)` qui précise les variables à représenter ainsi que des options de couleur/taille/groupe...
- Le type de graph : `geom_point()`/`geom_line()`/ `geom_histogram()`...

Les exemples ci-dessous permettent de bien comprendre.

```
# Exemples de graphiques ggplot2

ggplot(newdf)+
```

```
    aes(x=Area,y=Population)+
    geom_point()

newdf %>% ggplot()+
    aes(x=Area, y=Population, color=`Life Exp`>70) +
    geom_point()

linreg<-lm(newdf$Population~newdf$Area)
PopulationPred=predict(linreg)
ggplot()+
    aes(x=newdf$Area,y=newdf$Population) +
    geom_point() +
    geom_line(aes(x=newdf$Area,y=PopulationPred), col="red") +
    theme_light()

newdf %>% ggplot() +
    aes(x=Area,y=Population) +
    geom_point() +
    geom_smooth(method="lm") + # ajoute une droite de régression linéaire
                               # avec intervalle de confiance
    theme_light()


newdf %>% ggplot() +
    aes(x=Area) +
    geom_histogram(fill="blue")

ggplot(newdf) +
    aes(y=Area,x=HLE)+
    geom_boxplot(fill="blue")+
    coord_flip()
# Attention ici la variable HLE est numérique !
# On la transforme en factor :
newdf <- newdf %>% mutate(HLE= as.factor(HLE))
ggplot(newdf) +
    aes(x=HLE,y=Area) +
    geom_boxplot() +
    theme_classic()
```

```
# Pour sauvegarder un graphique :
myplot <- newdf %>% ggplot() +
    aes(x=HLE,y=Area) +
    geom_boxplot() +
    theme_light()
print(myplot)
ggsave("myplot.png") # ou ggsave("myplot.pdf")

# Pour sauvegarder plusieurs graphiques à la suite dans un pdf :
pdf("myplot.pdf")
ggplot(mtcars, aes(wt, mpg)) + geom_point() + theme_classic()
ggplot(mtcars, aes(wt, mpg)) + geom_point() + theme_light()
ggplot(mtcars, aes(wt, mpg)) + geom_point() + theme_void()
ggplot(mtcars, aes(wt, mpg)) + geom_point() + theme_dark()
ggplot(mtcars, aes(wt, mpg)) + geom_point() + theme_linedraw()
ggplot(mtcars, aes(wt, mpg)) + geom_point() + theme_bw()
dev.off()
```

Pour combiner plusieurs graphes, on utilisera la librairie cowplot. Je vous invite à regarder le site `https://wilkelab.org/cowplot/articles/plot_grid.html` pour plus de détails.

```
library(cowplot)

g1<-ggplot(newdf)+aes(x=Area,y=Population)+geom_point()+
ggtitle("Graphe␣1")

g2<-ggplot(newdf)+aes(y=Area,x=HLE)+geom_boxplot(fill="blue")+
coord_flip() + ggtitle("Graphe␣2")

plot_grid(g1,g2)
```

Le monde des possibles est immense avec `ggplot2` donc on ne pourra pas en faire le tour en quelques heures. Vous pouvez utiliser la cheat sheet de `ggplot2` pour vous aider et consulter par exemple : `https://larmarange.github.io/analyse-R/graphiques-bivaries-ggplot2.html` Pour vous faciliter la vie avec `ggplot2` il est possible d'utiliser l' `add_in Esquisse` qui vous permet de faire des graphiques "à la main" et donne le code R associé.

```r
install.packages("esquisse")
```

À savoir qu'il existe également un autre package très utilisé dans la communauté R pour réaliser des graphiques, le package `plotly`. Il est basé sur la bibliothèque JavaScript Plotly.js et permet de créer des **visualisations interactives** dans R (zoom, affichage de détails au survol, sélection dynamique de sous-ensembles de données...) qui peuvent notamment être intégrées dans des rapports HTML (avec Quarto ou R Markdown) ou des tableaux de bord (avec Shiny). Les graphiques `ggplot2` peuvent être convertis en graphiques interactifs avec `ggplotly()`.

7 Exercice - Module 2

Rappel : les exercices sont à faire en Quarto (ou Rmarkdown) et doivent être rendu dans le même fichier. Vous devez donc compléter votre fichier initié à la fin du module 1 `Exercice_R_NOM_PRENOM.qmd`.

1. Importer les données PisaFR et observer si la structure des données est la même que PisaUS.
2. (Sans utiliser `dplyr` ni `ggplot2`) Supprimer la colonne GLCM. Supprimer toutes les observations dont la somme des notes en français (READ) et en maths (MATH) est inférieure à 1000, ainsi que celles des étudiants dont les notes en sciences (SCIE) sont inférieures à 500. Effectuer un histogramme pour vérifier que votre filtre a bien fonctionné pour la dernière condition.
3. Refaire la question 2 en utlisant les commandes des packages `dplyr` et `ggplot2`
4. Manipuler les données en utilisant les commandes slice, filter, select, relocate, rename, arrange, mutate, group_by et le pipe.
5. Supprimer toutes les données et recharger de nouveau `PisaFR`. Réaliser quelques statistiques descriptives et graphiques descriptifs en variant les commandes. Commenter. Que faut-il retenir de ce jeu de données ?