Introduction

Interface RStudio

4 panneaux (position et onglets affichés changeables dans *View* > *Panes* > *Pane layout*)

- Source : onglets = scripts + visionneuse de tableaux
- Console :
  - Console : exécution des scripts, affichage des résultats
  - Terminal : terminal pour des commandes, notamment Git
- Environment, History, Connect…
  - Environment : liste des variables
  - History : historique des lignes de code exécutées
- Files, Plots, Packages…
  - Files : documents du répertoire de travail (*working directory*) actuel
  - Plots : affichage des graphiques
  - Packages : liste des *packages* installés (cf plus loin)
  - Help : documentation des fonctions et *packages*

Types de documents R

Le type d'un document est choisi à sa création.

Les 2 types les plus utiles :

- R script : script simple, exécution ligne par ligne ;
- R Markdown : document contenant du texte et des blocs de code (*chunks*), exécution par bloc ;
  - utilise la syntaxe Markdown pour le texte, possibilité d'ajouter des titres, du formatage, etc.
  - possibilité d'ajuster les paramètres des *chunks* : visibles ou non, résultats calculés ou non, taille des graphiques…
  - exportable (*Knit*) aux formats HTML et LaTeX notamment

Un script R simple est plus facile à manier ; un Markdown permet de séquencer son code et le commenter plus facilement, ainsi que de générer un document HTML propre à la fin.

Projets

Un projet R permet de désigner un dossier comme un espace de travail unique.

- pas besoin de chemin complet pour les documents ; on peut accéder à ceux du dossier uniquement avec leur nom
- garde en mémoire tous les onglets ouverts dans le panneau Source ; rouvrir le projet restaure les onglets présents à sa précédente fermeture

Conseils d'ordre général

Les erreurs s'affichent en rouge dans la console. Leur phrasé est généralement clair, et vous indique bien où est l'erreur dans le script.

Sinon, copier-coller le message d'erreur dans un moteur de recherche, et voir ce que d'autres utilisateurs en ont dit, marche très bien aussi.

La commande la plus importante en R est help() ou ?. Elle permet d'ouvrir la documentation d'une fonction, pour comprendre son fonctionnement et ses paramètres.

Objets en R et manipulations de base

Mode d'un objet

Les modes les plus importants ("types" des objets) :

- Integer : nombre entier
- Double : nombre décimal
- Numeric : nombre en général
- Logical : booléen (TRUE, FALSE)
- Character : chaîne de caractères (= texte)
- NA : valeur manquante (*not available*)

Opérations de base

- numeric :
  - +, -, *, /, ^, exp(), log(), factorial()…
  - %% pour le reste d'une division euclidienne (modulo)
  - round() pour un arrondi
- logical :
  - == pour test d'égalité, != pour inégalité, >, >=…
  - & pour AND, | pour OR, ! pour NOT
- character :
  - paste() pour concaténer
  - substr() pour une sous-chaîne de caractères
- typeof(), mode() pour déterminer le mode d'un objet

Vecteurs

L'équivalent R des arrays numpy (de dimension 1) en Python : liste organisée d'objets, pas forcément du même type.

Format pour la création : c(elem1, elem2, …)

Les opérations (somme, produit, etc.) se font, par défaut, élément par élément.

Les indices commencent à 1, contrairement à Python.

Création de vecteurs

- a:b : vecteur c(a, a+1, …) jusqu'au dernier élément inférieur ou égal à b (range en Python)
- rep() : vecteur répétant un élément plusieurs fois (cf documentation)
- seq() : vecteur allant d'une valeur à une autre (cf documentation)

Opérations sur des vecteurs

- length() : nombre d'éléments (len en Python)
- min(), max() : minimum ou maximum du vecteur
- which.min(), which.max() : indice du minimum ou du maximum
- sort() : rangement (ordre croissant par défaut)
- summary() : statistiques descriptives, dépendent du type des éléments
- crochets : sélection d'éléments, par indices ou par une condition
- head(), tail() : sélection des premiers ou des derniers éléments
- %*% : produit scalaire
- c(…, …) : concaténation – il suffit de mettre deux vecteurs dans un même nouveau vecteur

Matrices

L'équivalent des arrays numpy (de dimension 2) en Python : des données numériques uniquement, rangées en tableau.

Format pour la création : matrix(…, ncol = …, nrow = …)

Les opérations (somme, produit, etc.) se font, par défaut, élément par élément.

Création de matrices

- diag() : matrice diagonale

Opérations sur des matrices

- dim() : dimensions
- nrow(), ncol() : nombre de lignes ou de colonnes
- t() : transposée
- solve() : matrice inverse
- %*% : produit matriciel
- cbind() : concaténation par colonnes (côte à côte)
- rbind() : concaténation par lignes (l'une sur l'autre)
- rownames(), colnames() : noms des lignes ou des colonnes, s'ils existent
- crochets : sélection d'éléments, par indices ou par une condition

Listes

L'équivalent des dictionnaires en Python : des couples clé-valeur, ordonnés (contrairement à Python).

Format de création : list(clé1 = valeur1, clé2 = valeur2, …)

On ne peut pas faire d'opérations directement sur une liste, il faut en extraire les éléments ou utiliser lapply().

Opération sur des listes

- length() : nombre de couples clé-valeur
- crochets simples : sélection d'un couple clé-valeur par son indice ou par sa clé
- crochets doubles : sélection d'une valeur par son indice ou par sa clé
- $: sélection d'une valeur par sa clé
- append() : concaténation de listes (ou ajout de valeurs, par concaténation d'une autre liste contenant les nouvelles valeurs)
- names() : vecteur des clés

Facteurs

Comme des vecteurs, mais dont les valeurs sont considérées comme des modalités (*levels*) d'une variable qualitative.

Format de création : factor(vecteur)

On ne peut pas faire d'opérations directement sur un vecteur, il faut le transformer en vecteur avec as.vector() (cf plus loin).

Opérations sur des facteurs

- levels() : modalités du facteur
- summary() : effectif de chaque élément

Data frames

L'équivalent des data frames pandas en Python : des tableaux de données.

Format de création : data.frame(col1 = c(elem1, elem2, …), col2 = …)

Tous les éléments d'une seule colonne doivent être de même type, mais deux colonnes peuvent être de types différents.

Toutes les opérations sur des matrices (sauf transposée, inverse et produit matriciel) fonctionnent de la même façon sur des data frames.

Opérations sur des data frames

- summary() : statistiques descriptives pour chaque colonne
- $: sélection d'une colonne par son nom
- na.omit() : supprime les lignes contenant des NA

Identification de types et transtypage

Pour déterminer si un objet est d'un type, on utilise la fonction is.[type] : is.numeric(), is.na(), is.vector(), is.data.frame()…

Pour changer le type d'un objet, si c'est possible, on utilise la fonction as.[type] : as.character(), as.matrix(), as.factor()…

Chargement et sauvegarde de jeux de données

Jeux de données dans R

R de base, ainsi que plusieurs *packages*, disposent de jeux de données préexistants. On peut les charger avec data(*nom du jeu de données*).

Accès à des jeux de données externes

Les fonctions read.table(), read.csv() et autres permettent d'accéder à des documents externes depuis :

- un fichier sur l'ordinateur
- une URL

En pratique, pour un fichier local, on tape rarement les commandes à la main : dans *File > Import dataset*, on peut effectuer l'importation facilement. La ligne de commande est générée automatiquement et exécutée dans la console, il suffit de la copier-coller dans le script.

Notez qu'on utilise *From Text* pour un fichier CSV.

Pour une URL, il faudra écrire la ligne de commande à la main. Référez-vous à la documentation de read.table() et read.csv() pour les paramètres.

Fichiers RData

R a un format spécial : RData. Les fichiers RData se finissent par .rData, .RData, .rda…

Pour charger un fichier RData, on utilise load() avec le chemin vers le fichier.

On peut également sauvegarder un data frame au format RData avec save(), à laquelle on fournit les éléments à sauvegarder et le nom du nouveau fichier (avec file = …).

Graphiques

R base

R permet de faire des graphiques basiques. Les fonctions les plus utiles sont :

- plot(x, y, …) : trace y en fonction de x, avec d'éventuels paramètres :
  - type : type de tracé (points, ligne, points reliés…)
  - col : couleur du tracé
  - lty : type de ligne (continue, pointillée…)
  - pch : type de points (rond, croix, carré…)
- lines(x, y, …) : ajoute à un plot existant des lignes de coordonnées (x,y)
- points(x, y, …) : ajoute à un plot existant des points de coordonnées (x,y)
- title() : ajoute un titre à un plot existant.
- legend() : ajoute une légende à un plot existant
- boxplot() : trace un boxplot
- hist() : trace un histogramme
- barplot() : trace un diagramme en barres

L'instruction par(), avant un plot(), permet aussi de définir des paramètres graphiques (cf documentation).

GGplot

GGplot est un *package* (cf plus loin) qui permet de faire de vrais, beaux graphiques. Il est recommandé dès qu'on veut faire des graphiques pour un rapport.

Un graphique GGplot se construit par couches, selon la syntaxe suivante :

ggplot(*jeu de données*) + aes(x = *variable en x*, y = *variable en y*, …) +

geom_[*type de graphique*]

Couche aes : variables utilisées pour le(s) graphique(s)

Tous les paramètres dans aes() sont des variables.

- x, y : variable en x et/ou en y
- color : variable donnant la couleur du tracé
- fill : variable donnant la couleur de remplissage
- shape : variable donnant la forme des points
- size : variable donnant la taille des points
- linetype : variable donnant le type de ligne

Couche geom_ : type(s) de graphique(s)

Tous les paramètres dans les geom_ sont des constantes.

- geom_point() : nuage de points
- geom_line() : ligne
- geom_bar() : diagramme en bâtons
- geom_histogram() : histogramme
- geom_boxplot() : boîte à moustaches
- geom_smooth() : lissage/régression sur le geom_ précédent

Couche scale *(optionnel)*

Permet de changer les palettes de couleur du graphique. Voir des exemples sur R Graph Gallery.

Couche facet *(optionnel)*

Permet d'arranger plusieurs graphiques en grille. Voir des exemples sur R Graph Gallery.

Couche labels *(optionnel)*

Permet de rajouter titre, sous-titre, annotations, etc.. Voir des exemples sur R Graph Gallery.

Références R Graph Gallery

R Graph Gallery est un site qui rassemble des exemples de graphiques avec divers *packages*, y compris R base, et les lignes de code qui permettent de les obtenir. Je le recommande pour s'inspirer de graphiques existants.

Packages

Un *package* est l'équivalent d'une bibliothèque Python : un ensemble de fonctions et de jeux de données qu'on peut charger pour les utiliser.

Pour utiliser un *package*, il faut d'abord l'installer avec install.packages(*"nom du package"*).

Il y a ensuite 2 façons d'utiliser les fonctions :

- charger le *package* avec library(*nom du package*)
  - permet d'utiliser librement toutes les fonctions du *package*
  - si des fonctions du *package* ont le même nom que des fonctions existantes avant le chargement, les précédentes sont écrasées au profit des nouvelles
- utiliser les fonctions individuellement avec *nom du package*::*nom de la fonction* (ex : dplyr::filter, pour la fonction filter() du *package* dplyr)
  - évite de charger un *package* entier pour une seule fonction
  - permet de spécifier, en cas d'homonymes, de quel *package* vient une fonction

Structures de contrôle et fonctions

Structures de contrôle

Remarque importante sur la syntaxe :

- la condition se met toujours entre parenthèses
- l'intérieur d'une boucle est défini par des accolades
- une boucle peut tenir sur une seule ligne ; il n'y a pas besoin d'accolades dans ce cas

Boucle if (si… sinon…)

```
if (condition){
        instructions
  } else if {
        instructions
  } else {
        instructions
}
```

Boucle for (pour … allant de… à…)

```
for (condition){
        instructions
}
```

Boucle while (tant que…) : pareil que la boucle for

Boucle repeat (faire en boucle…)

```
repeat{
        if (condition d'arrêt) break
        instructions
}
```

Fonctions

Les fonctions R sont les mêmes que les fonctions Python : on les définit pour pouvoir les utiliser plus tard.

Comme en Python, le return n'est pas obligatoire.

Syntaxe

```
nom de la fonction = function(param 1, param 2, …) {
        instructions
        return(résultat)
}
```

Avertissements et erreurs

- avertissement : warning(*message d'avertissement*)
- erreur : stop(*message d'erreur*)

La fonction continue après un avertissement mais s'arrête après une erreur.

Aléatoire et distributions de probabilité

Tirage d'un échantillon

La fonction sample() permet de tirer *k* éléments d'un vecteur à *n* éléments. Le tirage peut être avec ou sans remise, et avec ou sans vecteur de probabilités (cf documentation).

Graines

Pour fixer l'aléatoire, on utilise l'instruction set.seed(*n*) où *n* est un nombre entier de votre choix. Toute génération aléatoire après un set.seed() sera la même à chaque exécution.

Distributions de probabilité

Chaque loi de probabilité (normale, uniforme, binomiale…) dispose de plusieurs fonctions. Le nom d'une fonction est constitué d'une lettre indiquant son usage (r = random, q = quantile, p = probabilité) et du nom abrégé de la distribution souhaitée.

Par exemple, runif() génère un nombre aléatoire selon une loi uniforme ; qnorm() donne un quantile de la loi normale.

Types de fonctions (1ère lettre)

- d- : donne la densité
- p- : donne la fonction de distribution
- q- : donne un quantile
- r- : génère un nombre aléatoire

Distributions

- -binom : loi binomiale
- -pois : loi de Poisson
- -unif : loi uniforme
- -norm : loi normale
- -exp : loi exponentielle

Régression linéaire, régression logistique

Régression linéaire

La fonction lm() permet d'effectuer une régression linéaire avec une variable à expliquer quantitative, et des variables explicatives quantitatives et/ou qualitatives.

Le résultat est un objet de type liste qui contient, entre autres : coefficients estimés, valeur prédites, R^2, résidus…

Syntaxe : lm(*y ~ var 1 + var 2 +…*, data = *jeu de données*)

En particulier :

- le premier argument est un objet de type formule ; la variable à expliquer à gauche du tilde, les variables explicatives à droite, séparées par des + (pour un modèle additif)
- pas besoin de dollars dans la formule, le nom des colonnes (variables) suffit
- si toutes les variables sont utilisées dans la régression, on peut écrire *y ~ .* (un point = tout) ; si on veut exclure une variable, on peut écrire *y ~ .- var* (le moins = sans cette variable)
- pour des interactions, on utilise deux points : *y ~ var 1 : var 2*

Régression logistique

Une régression logistique se fait avec la fonction glm(). Cette fonction permet d'évaluer des modèles linéaires généralisés, donc de changer la composante aléatoire (loi de Y sachant X) et la fonction de lien (lien entre espérance de Y et combinaison linéaire des X).

Syntaxe pour une régression logistique :

glm(*y ~ var 1 + var 2 +…*, data = *jeu de données*, family = "binomial")

Mêmes remarques que pour la régression linéaire.

Fonctions utiles avec des objets de classe lm et glm

Les objets renvoyés par lm() et glm() sont des listes, il est intéressant d'explorer avec $ tout ce qu'elles contiennent.

Les fonctions suivantes peuvent également être utiles :

- summary() : résume les résultats de la régression
- residuals() : résidus de la régression
- step() : lance une approche pas à pas (*stepwise algorithm*) pour sélectionner le meilleur modèle au vu d'un critère (par défaut l'AIC ; le paramètre *k* permet d'ajuster ça, cf documentation) (attention, selon la direction, *forward* ou *backward*, le premier argument est différent)
- AIC(), BIC() : calcule l'AIC ou le BIC du modèle
- predict() : prédit Y_i à partir d'une nouvelle valeur X_i et du modèle (cf documentation de predict.lm())
- plot() : trace divers graphiques des résidus, selon la valeur du paramètre *which* :
  - which = 1 : résidus vs valeurs prédites
  - which = 2 : Q-Q plot des résidus

Package FactoMineR

FactoMineR a été en partie développé par des chercheurs de l'Agro, donc tout le monde dans le couloir des maths aime l'utiliser. Il est également incontournable dans le domaine de l'analyse factorielle.

Tous les détails sur le *package* sont disponibles sur le site dédié : http://factominer.free.fr/index_fr.html

Méthodes classiques

- PCA() : analyse en composantes principales (ACP)
- CA() : analyse factorielle des correspondances (AFC)
- MCA() : analyse des correspondances multiples (ACM)
- HCPC() : classification hiérarchique sur composantes principales
  - prend en entrée un résultat d'ACM (attention au paramètre ncp)
  - récupère les coordonnées sur les composantes principales
  - effectue une CAH puis une consolidation k-means
  - rend une liste contenant les résultats de la classification

Descriptions de résultats

- catdes() : caractérisation des modalités d'une variable qualitative par les autres variables
- dimdesc() : caractérisation des dimensions d'une analyse factorielle par les autres variables
- descfreq() : caractérisation des lignes d'un tableau de contingence par les modalités en colonnes