Chapitre 4
La programmation avec R

1. Introduction

Précédemment, certains objets R ont été étudiés, à savoir : les vecteurs, les facteurs, les data frames, les matrices, etc. qui permettent le stockage en mémoire des données. Cependant, il existe des structures, essentielles à l'implémentation d'algorithmes, qui n'ont pas été abordées. Il s'agit d'outils à la base de la programmation, de la grammaire du langage, pour non seulement structurer mais aussi traduire en code le raisonnement du programmeur. Ainsi, ce chapitre portera sur les fondamentaux de la structuration du code avec R et traitera des structures de contrôle (les conditions et les boucles), des fonctions et de la programmation orientée objet (POO) notamment.

2. Les structures de contrôle

On distingue généralement et principalement deux catégories de structures de contrôle, celles qui servent à poser des conditions ou à tester une information d'une part et celles qui permettent la répétition d'une ou de plusieurs opérations d'autre part, que l'on appelle en programmation les boucles.

Mais avant d'y arriver, on va aborder un type de structure simple qui sert à grouper plusieurs instructions ayant ou non une suite logique.

2.1 Les structures de groupage d'instructions

2.1.1 Le point-virgule

Il permet d'écrire plusieurs instructions en une seul ligne de commande. Il est plus pratique de ne l'utiliser qu'en cas d'instructions silencieuses (notamment pendant l'initialisation de plusieurs objets) pour permettre une meilleure lisibilité du code. Ainsi, les instructions suivantes :

```
> # initialisation de plusieurs variables
> a <- 3
> b <- 5
> c <- b*b-a*c
> mu <- mean(1:12)
```

Peuvent être écrites comme ci-dessous :

```
> # initialisation de plusieurs variables
> a <- 3;b <- 5;c <- b * b - a * c ;mu <- mean(1:12)
```

2.1.2 Les accolades

R permet également de regrouper entre accolades une suite d'instructions pointant vers un résultat. L'exemple ci-dessous permet d'abord d'initialiser les variables **a** et **b** et de déterminer **c** :

```
> # détermination de c = a + b
> {
+    a <- 2.3
+    b <- 5.7
+    c <- a + b
+ }
> # ou une autre façon plus élégante
>  c <- {
+  a <- 2.3
+  b <- 5.7
+  a + b
+  }
> # afficher la valeur de c
> c
[1] 8
```

Remarque

L'utilisation d'accolades n'est vraiment pertinente que pour bien structurer la détermination de certains résultats impliquant plusieurs instructions intermédiaires, notamment lors de la construction de structures plus élaborées (les fonctions, etc.).

2.2 Les structures conditionnelles

2.2.1 La structure if...else

Pour les habitués de Microsoft Excel, la structure si…alors…sinon est équivalente à la fonction **SI()** de ce fameux tableur, avec R cette fonction prend plusieurs formes :

```
# la première forme if ... le si simple
if(" Poser une condition ici "){
" Placer Les instructions ici si vrai "
}
# la seconde forme if …else… le si …sinon
if( " Poser une condition ici "){
  " Placer les instructions ici si vrai "
}else{
 " Placer les instructions ici sinon "
}
# la troisième forme la fonction ifelse() pour un test vectoriel
ifelse(" Poser une condition sur un vecteur ici ",
"si vrai instruction ",
" sinon instructions ")
```

Le principe de fonctionnement est simple, il suffit de poser un test ou une condition, c'est-à-dire une opération à même de générer une valeur booléenne à savoir **TRUE** ou **FALSE**, ensuite placer à la suite une ou plusieurs instructions selon que le test est vrai ou faux. Ainsi, une instruction ne sera exécutée que si la valeur booléenne générée par le test est vrai (**TRUE**).

```
> # si vrai afficher le texte
> if(TRUE){
+   print(" Hey !!! ça fonctionne ! ")
+ }
[1] " Hey !!! ça fonctionne ! "
> # si faux rien ne s'affiche
> if(FALSE){
+   print("Hey !!! ça ne fonctionne pas ! ")
+ }
> # avec un test générant un booléen FALSE
> if(5<3){
+   print("rien ne s'affiche")
+ }
> # avec un test générant un booléen TRUE
> if(5>3){
+   print("5 est bien supérieur à 3")
+ }
[1] "5 est bien supérieur à 3"
```

Mis à part des opérations logiques comme ci-dessus, il s'agit le plus souvent de tester la structure d'un objet notamment pour en déterminer la classe afin d'effectuer une suite d'instructions précises selon la classe. Les fonctions R permettant de réaliser de telles conditions sont par convention nommée selon le modèle **is.nom_de_la_classe()**. Les plus fréquemment utilisées sont **is.vector()**, **is.numeric()**, **is.string()**, etc. Pour obtenir la liste complète de ce type de fonctions très utiles lors de l'utilisation des structures conditionnelles, il faut procéder comme ceci :

```
> ls(pattern = "^is", baseenv())
 [1] "is.array"
 [2] "is.atomic"
 [3] "is.call"
 [4] "is.character"
 [5] "is.complex"
 [6] "is.data.frame"
 [7] "is.double"
 [8] "is.element"
 [9] "is.environment"
[10] "is.expression"
[11] "is.factor"
[12] "is.finite"
[13] "is.function"
[14] "is.infinite"
[15] "is.integer"
[16] "is.language"
[17] "is.list"
[18] "is.loaded"
```

En voici une illustration avec la fonction **is.numeric()** qui permet de tester si un objet est de classe numérique :

```
> v <- 100.0
> # tester si v est numérique
> if (is.numeric(v)) {
+    # si vrai en déterminer le logarithme
+    log10(v)
+ }
[1] 2
>
```

Toutefois, l'utilisation la plus répandue reste la seconde forme, **if…else** (si sinon), qui se prête plus à l'algorithmique :

```
> # exemple affichant la valeur absolue d'un nombre
> # aléatoirement généré compris entre -10 et 10
> a <- runif(1, -10, 10)
> if (a > 0) {
+    print(a)
+ } else {
+    print(-a)
+ }
[1] 5.802002
>
```

Sa forme la plus complexe consiste à imbriquer plusieurs structures **if…else**, comme l'illustre le code ci-dessous :

```
> # demander à l'utilisateur de saisir une valeur
> x <- scan(nmax = 1)
1:
7

Read 1 item
> ## tester si x est bien un nombre
> if (is.numeric(x)) {
+    msg <- paste(x, ", est un nombre")
+    ## structure if...else imbriquée
+    # testant la parité de x
+    if (x %% 2 == 0) {
+    cat(msg, "pair")
+    } else {
+    cat(msg,"impair")
+    }
+    # fin du test de parité
+ } else {
+   cat(x,"n'est pas un nombre")
+ } # fin du test du type de x
7 , est un nombre impair
```

La troisième forme, **ifelse**, est une fonction qui opère un test sur un vecteur de données contrairement aux formes précédentes qui ne testent que des scalaires :

```
> x <- c(-5, 7, 0, 9, -2)
> # valeur absolue du vecteur x
> ifelse(x >= 0, x, - x)
[1] 5 7 0 9 2
> # logarithme de x si possible
> ifelse(x > 0,
+       paste0("log(",x,") vaut ",log(x)),
+       paste0("log(",x,") n'est pas possible"))
[1] "log(-5) n'est pas possible"
[2] "log(7) vaut 1.94591014905531"
[3] "log(0) n'est pas possible"
[4] "log(9) vaut 2.19722457733622"
[5] "log(-2) n'est pas possible"
```

Naturellement, cette dernière forme accepte également l'imbrication comme les autres formes de condition, mais ne reste pertinente que pour tester des vecteurs.

2.2.2 La structure switch()

Contrairement aux autres structures conditionnelles, cette structure est un peu particulière. R et les autres langages de programmation courants présentent plutôt la structure **switch()** comme un menu avec des choix ou sélections multiples de type clé-valeur qui prend obligatoirement en argument une clé et vérifie puis renvoie la valeur sur laquelle pointe la clé :

```
switch("Clé de sélection ",
       "sélection 1",
       "sélection 2",
       ".........",
       "sélection n",
       "sélection par dé faut ")
```

La clé de sélection est nécessaire à cette structure, elle permet d'indexer une des options ou sélections prévues. Elle peut être une chaîne de caractères ou numérique :

```
> # clé en charactère
> switch("b",
+     a = c(2.0,3.6,9,0,3),
+     b = "Bonjour le monde",
+     c = data.frame(1:4,LETTERS[1:4]),
+     "option par defaut")
[1] "Bonjour le monde"
> # clé numerique et affectation
> val <- switch(2,
+       a = c(2.0, 3.6, 9, 0, 3),
+       b = "Bonjour le monde",
+       c = data.frame(1:4, LETTERS[1:4]),
+       "option par defaut")
> val # afficher  val
[1] "Bonjour le monde"
```

Toutefois, en l'absence de correspondance c'est soit l'objet **NULL**, soit l'option ou sélection par défaut qui est retourné si cette dernière est prévue :

```
> # cas de clé inexistante
> val <- switch("d",
+       a = c(2.0, 3.6, 9, 0, 3),
+       b = "Bonjour le monde",
+       c = data.frame(1:4, LETTERS[1:4]),
+       "option par defaut")
> val
[1] "option par defaut"
```

Chapitre 6
Traitement du langage naturel

1. Positionnement du problème

L'importance des applications de *text mining* n'a cessé d'augmenter ces dernières années. L'émergence des réseaux sociaux a intensifié ce phénomène. La principale caractéristique des données textuelles, et qui les différencie des données semi-structurées (fichiers XML ou JSON) ou structurées (bases de données), réside dans la nécessité d'avoir à fouiller de façon non déterministe au sein de chaque item de données.

Les données textuelles peuvent être traitées à différents niveaux de profondeur :

– Identification de mots appartenant à des listes de mots (*bag of words*).

– Identification de chaînes de mots (phrases ou expressions).

– Identification d'éléments sémantiques (reconnaissance des mots par leur sens).

Une autre des caractéristiques, typique d'un texte, réside dans la proportion très faible entre le nombre de mots d'un texte et le nombre total de mots possibles dans une langue ou un jargon. Cela crée des structures de données très lacunaires, en particulier quand on cherche à représenter l'information sous forme de structure comme des tableaux (*sparse matrix*).

Dans ce chapitre, nous allons focaliser notre attention sur une méthode puissante, mais qui peut être un peu dissuasive quant à sa formulation : l'analyse sémantique latente.

Il existe de nombreuses autres techniques dans le contexte NLP, dont beaucoup sont des techniques très utiles, mais qui interviennent plutôt en amont des traitements de data sciences en extrayant des features. Celles-ci seront alors traitées de façon "habituelle".

2. Analyse sémantique latente et SVD

2.1 Aspects théoriques

Dénommée LSA (*Latent Semantic Analysis*) en anglais, l'analyse sémantique latente s'appuie sur la constitution d'une matrice comportant les valeurs d'une fonction particulière calculée à partir des occurrences des différents termes (mots) présents dans les différents documents.

Les documents peuvent être des textes, des mails, des tweets, des contributions de blog, des CV...

Chaque terme envisagé fait l'objet d'une ligne de cette matrice, et chaque colonne de la matrice correspond à un document.

À l'intersection des lignes et des colonnes se trouve le résultat du calcul d'une fonction. Ce résultat est d'autant plus grand que le nombre d'occurrences du terme est élevé dans le document et que le terme est rare en général. Les packages sont fournis avec différentes fonctions classiques, mais votre problème nécessitera peut-être que vous mettiez au point votre propre fonction. Une telle fonction doit toujours au moins refléter le fait qu'**un terme rare est plus significatif qu'un terme courant**.

La fonction qui mesure le plus couramment l'importance d'un terme dans un document en fonction du corpus se nomme TF-IDF (*Term Frequency-Inverse Document Frequency*). Elle s'appuie sur la loi de Zipf qui traite de la fréquence des mots dans un texte et dont l'interprétation théorique fait appel à la notion d'entropie.

L'utilisation de l'entropie est un excellent signe d'objectivité. C'est également une bonne nouvelle concernant le nombre d'hypothèses empiriques à vérifier sur la qualité du corpus.

Pour autant, vous devrez envisager sérieusement de concevoir votre propre fonction si votre problème est un peu particulier.

Remarque

Quand on dispose de plusieurs corpus et de plusieurs "fonctions d'importance", il peut être extrêmement efficace d'effectuer des analyses croisées sur ces différents aspects en ne retenant que peu de features à chaque croisement.

Les features obtenues sont parfois complémentaires, relativement indépendantes et hautement interprétables.

Dans de nombreux cas d'implémentation, les termes considérés sont des mots qui ont été démunis de leurs caractéristiques contextuelles (marques du pluriel et de genre, majuscules de début de phrase, marques de conjugaison...).

2.1.1 SVD : généralités

La matrice X est décomposée au travers d'une technique qui se nomme "décomposition en valeurs singulières" (SVD).

Cette technique possède de nombreux usages en data science, notamment la réduction de dimensions.

On obtient la décomposition suivante : $X = U\Sigma V^T$

Les matrices U et V sont des matrices orthogonales et donc telles que $U^TU = I$ et $V^TV = I$, le carré de leur déterminant est égal à 1.

La matrice Σ est une matrice diagonale (donc tous les termes qui ne sont pas sur la diagonale sont nuls). Notez que l'on obtient toujours une décomposition et qu'elle est unique.

Remarque

Remarque pour les matheux : une matrice orthogonale est toujours semblable à une matrice composée d'une diagonale de matrices de rotations planes et de 1 et de -1. Par "semblable", on entend qu'il existe une matrice de changement de base qui peut la transformer de cette façon. Dans le cas d'une matrice orthogonale, la matrice de changement de base est elle-même orthogonale !

2.1.2 Une justification de la décomposition SVD

Considérons la décomposition $X = U\Sigma V^T$..

Imaginez que la matrice X à décomposer ait pour dimension 10000 termes x 100000 documents et que la matrice diagonale Σ soit une matrice carrée de 5000x5000.

La matrice U aurait comme dimension 10000 x 5000 et la transposée de V aurait comme dimension 5000 X 100000.

Donc notre matrice X comporterait 10^9 cellules, alors la somme du nombre de cellules des trois matrices serait de : 575.10^6.

Dans un tel cas de figure, on a réduit le volume des données de 50 % sans perdre aucune information. Ceci est courant quand la matrice X est très lacunaire (sparse matrix : avec de nombreux zéros, qui ne sont d'ailleurs pas encodés pour gagner de la place).

En fait, nous verrons plus loin que l'application de cette méthode nous procure une autre opportunité de gain de volume. En effet, nous pourrons supprimer les lignes et les colonnes les moins significatives des matrices résultant de la décomposition sans avoir perdu beaucoup d'information.

2.1.3 SVD dans le contexte LSA

Au sens fonctionnel, dans ce contexte, chaque **ligne** de la matrice U est un **terme** et chaque **colonne** de la matrice U peut être interprétée comme étant un **topique**, c'est-à-dire une **combinaison de termes** (mots) avec un coefficient de pondération exprimant le poids de chaque terme dans le document.

La matrice Σ est une matrice diagonale dont chaque valeur correspond au poids du topique au regard de l'ensemble des topiques du corpus des documents étudiés.

La matrice V^T nous procure des lignes de topiques avec des colonnes représentant les documents.

On peut **diminuer les dimensions du problème en éliminant les topiques ayant le plus faible poids**, ce qui enlève les lignes correspondantes de U et de V et les lignes et les colonnes correspondantes de la matrice diagonale Σ (toutes les dernières lignes, car on a classé la matrice diagonale en fonction des valeurs des poids).

2.1.4 Interprétation

Pour mieux percevoir le sens de tout cela, il faut intégrer l'idée que la matrice U est composée de plusieurs vecteurs (colonnes) qui représentent nos topiques (c'est-à-dire nos concepts) et que l'on a donc fabriqué **un nouvel espace dont les axes sont les topiques** (ceux que l'on a conservés tout en perdant peu d'information).

On peut **représenter chaque document dans ce nouvel espace** de plus petite dimension et plus adapté à la topologie réelle des topiques/concepts sous-jacents.

La nouvelle dimension du problème étant raisonnable et cet espace étant porteur de sens, il devient envisageable d'utiliser des algorithmes classiques de classification ou d'apprentissage sur ce nouvel espace.

Remarque

En mathématiques, on nomme les vecteurs (colonnes) de U, vecteurs singuliers et les valeurs de la diagonale de Σ valeurs singulières.

Remarque

La décomposition SVD minimise la distance euclidienne des documents avec les vecteurs singuliers. Le carré de la valeur singulière correspondant à un vecteur singulier est parfois perçu comme une "énergie" latente. Pour choisir quels vecteurs on va éliminer, on peut appliquer une méthode de Pareto 80/20 à cette énergie. Autrement dit, on somme successivement les carrés des valeurs singulières de la plus grande à la plus petite et, quand cette somme atteint 80 % de la somme totale des carrés, on considère avoir trouvé l'indice du dernier vecteur singulier. Cette mécanique élimine tous les vecteurs colonnes qui suivent (nous reverrons cette méthode dans un autre contexte de réduction de dimension au chapitre Feature Engineering). Le seuil de 80 % est indicatif, c'est un paramètre comme un autre du modèle qu'il conviendra d'optimiser en fonction de la qualité des résidus et des performances attendues.

SVD capte les "corrélations" linéaires, donc en cas de problème moyennement linéaire, la question est de savoir s'il faut transformer ses features ou la variable expliquée avant de les utiliser pour revenir à un problème plus linéaire. Quand le contexte n'est franchement pas linéaire, il est possible de basculer sur l'algorithme **Isomap**.

D'autres types de décompositions de matrices existent et ont des usages similaires :

– "CUR matrix decomposition" qui donne un résultat non unique et approximé, mais qui est parfois plus facilement interprétable.
– "Eigen decomposition" qui est la méthode de référence que l'épouse de l'auteur (Mme Ghislaine Laude-Dumez) utilisait déjà dans ses recherches concernant la **constitution de clusters en utilisant des distances et des similarités linguistiques dans ses travaux de doctorat en 1986**... ses travaux nous ayant inspirés depuis lors et jusqu'à aujourd'hui. Nous en aborderons un des prérequis dans le chapitre Feature Engineering quand nous parlerons de PCA (*Principal Component Analysis* : analyse en composantes principales).

2.1.5 Alternative non linéaire, Isomap (MDS, géodésique, variété, manifold)

Isomap s'appuie sur l'exploration "géodésique" d'une variété (*manifold*) en utilisant la mécanique de positionnement multidimensionnel (MDS).

Remarque

MDS, que nous avons abordé indirectement quand nous avons précédemment parlé de "similarité", permet d'aborder une relation de proximité qui peut s'appuyer sur des distances dont la topologie ne nous est pas forcément bien connue et maîtrisée.

L'idée d'une exploration géodésique est simple : au lieu de considérer l'espace dans lequel est noyé l'objet que l'on veut explorer et de considérer que la ligne droite est le chemin le plus court, on accepte la topologie de l'espace sur lequel on se déplace. Considérez cet exemple : sur la terre le chemin le plus court opérationnel entre deux points éloignés est un arc de cercle, et pas la corde qui relie ces deux points en passant par le sol !

Une variété (manifold) s'apparente à une surface noyée dans une dimension plus grande, comme l'est par exemple la surface de la Terre dans l'espace 3D l'entourant. Cette notion permet d'appréhender des propriétés topologiques non évidentes : par exemple la variété nommée ruban de Möbius est une surface ne comportant qu'une seule face !

En vous posant une question comme : "comment appliquer la méthode du gradient sur les géodésiques d'une variété comme le ruban de Möbius ?", **vous accédez à la véritable nature du problème du data scientist** qui aimerait un monde linéaire et qui découvre des mondes complexes quand le nombre de dimensions augmente.

2.2 Mise en pratique

2.2.1 Initialisation

Pour cet exemple pratique, nous avons déposé six fichiers textes en anglais (.txt), libres de droits, dans un répertoire de notre ordinateur.

Le volume total de ce mini corpus est de 2.4 Mo.

L'encodage des fichiers est UTF-8. Pour transformer un fichier texte manuellement en UTF-8, il suffit d'un éditeur qui permette cette option au moment de la sauvegarde (exemple : WordPad).

Nous allons utiliser un package LSA qui encapsule la décomposition SVD, mais il est possible d'utiliser d'autres packages de décomposition SVD.

```
rm(list = ls())
library(lsa)                          # latente sémantique analyse
getwd()
                                      # fichiers in /classiques
dir <- paste(getwd(),"classiques", sep ="/")
```

Le chemin où se trouve le fichier se construit par agrégation entre le chemin courant et le nom du sous-répertoire.

En traitement du langage naturel, il est courant d'éliminer certains mots avant les traitements à effectuer. La raison la plus courante est que ces mots ne sont pas significatifs, mais il existe beaucoup d'autres bonnes raisons de le faire (par exemple éliminer des noms d'entreprises, de personnes...).

On nomme "stop words" ces paquets de mots. Les packages vous fournissent des stop words pour diverses langues, mais les différents laboratoires gèrent leurs propres paquets de mots à éliminer suivant les contextes et les langues.

```
data(stopwords_en)                  # stopword anglais
stopwords_en[1:5]
```

```
[1] "a"      "about"  "above"  "across" "after"
```

Le package **`lsa`** nous permet de fabriquer une matrice des termes de nos fichiers qui comporte tous les mots du corpus **sauf les stop words**.

```
                                    # tous les mots
M_mots= textmatrix(dir, stopwords=stopwords_en,
                   stemming=FALSE,
                   minGlobFreq=0)
dim(M_mots)
M_mots.df <- data.frame(M_mots[,])
```

```
[1] 14208     6
```

La matrice est d'un format propre au package, qui comprend de nombreux attributs de listes, nous verrons sa manipulation plus loin.

Pour jeter un œil rapide sur le contenu de cette matrice de 14 208 lignes de termes et de 6 colonnes de documents, nous avons créé un **`data.frame`** provisoire qui ne servira à rien d'autre. Sous RStudio vous pouvez visualiser facilement ses 1000 premières lignes.

Il existe de nombreux algorithmes pour nettoyer les mots de telle façon que des mots comme "loves", "loved"... deviennent une même entité.

Attention, le résultat n'est alors pas constitué de mots, mais de **radicaux (stem)**. Par exemple, à un mot important de notre jeu de données, happy, correspond l'expression "happi" qui est son *stem*.

```
                                    # les radicaux/stem
M= textmatrix(dir, stopwords=stopwords_en,
              stemming=TRUE,
              minGlobFreq=0)
dim(M)
```

```
[1] 9321    6
```

C'est encore beaucoup de radicaux (9321). Nous allons paramétrer notre code pour éliminer des scories, la valeur du paramètre est empirique et nécessite quelques tâtonnements.