Limitations de R

Pierre Lafaye de Micheaux

25 September 2020

Contents

1 Introduction

Crédit: quelques éléments tirés d'un talk donné par Trevor Hastie

1.1 Le Problème de l'Apprentissage Supervisé

Point de départ:

- Mesures d'une variable réponse Y (aussi parfois appelée variable dépendante, ou cible).
- Vecteur de mesures sur p prédicteurs X (aussi appelé entrées, régresseurs, covariables, features, variables indépendantes).
- Dans le **problème de régression**, Y est quantitative (e.g, prix, pression artérielle).
- Dans le **problème de classification**, Y prend ses valeurs dans un ensemble non ordonné fini (survécu/mort, chiffre 0–9, catégorie de cancer d'un prélèvement de tissu).
- On a des training data $(x_1, y_1), \ldots, (x_N, y_N)$, l'échantillon d'apprentissage. Ce sont des observations (exemples, instances) de ces variables.

Objectifs:

Sur la base d'un échantillon d'apprentissage, on voudrait:

- Prédire de façon précise des cas tests non vus.
- Comprendre quelles entrées affectent la sortie, et comment.
- Évaluer la qualité de nos prédictions et inférences.

1.2 Apprentissage Non-supervisé (Unsupervised Learning)

- Pas de variable réponse, juste un ensemble de prédicteurs (*features*) mesurés sur un ensemble d'échantillons.
- L'objectif est un peu flou – trouver des groupes d'échantillons qui se comportent de la même manière, trouver des features qui se comportent de manière similaire, trouver des combinaisons de features ayant le plus de variation.
- difficile de savoir si l'on fait bien.
- différent de l'apprentissage supervisé (supervised learning), mais peut être utile en tant qu'étape de pré-traitement pour l'apprentissage supervisé.

2 Différent Types de Jeux de Données

Statistique traditionnelle: des experts du domaine travaillent pendant 10 ans pour apprendre de bonnes features; ils amènent au statisticien un **petit jeu de données propre**.

Approche aujourd'hui: on part d'un **gros jeu de données** avec de nombreuses features, et on utilise un algorithme de machine learning pour trouver les bonnes. **Un énorme changement**.

2.1 Comment mesure t-on la taille d'un (gros) jeu de données?

Bit: 0/1

Bytes (octet): 1B $= 2^3 =$ 8 bits

Kilobytes: 1 KB $= 2^{10}$ B

Megabytes: 1 MB $= 2^{10}$ KB

Gigabytes: 1 GB $= 2^{10}$ MB

Terabytes: 1 TB $= 2^{10}$ GB

Petabytes: 1 PB $= 2^{10}$ TB

Note: $2^{10} = 1,024 \approx 1,000$

L'un des plus gros jeux de données en astronomie est le Sloan Digital Sky Survey (SDSS, http://www.sdss.org). Chaque nuit (depuis 2000), le téléscope SDSS produit 200 GB de données. Mais il n'est pas facile de trouver (et de télécharger, comme vous l'avez sans doute réalisé!) des jeux de données très massifs. Pour un très gros jeu de données, 100% libre et gratuit que vous pouvez télécharger au format CSV, voir https://www.gdeltproject.org/#downloading .

> "A single year of the GDELT GKG totals 2.5TB"

Pour ce cours, on ne traitera que des jeux de données de l'ordre de la dizaine de Gigabytes.

Qui plus est, aujourd'hui R peut "seulement" adresser jusqu'à 8 TB de RAM s'il tourne sur une machine 64-bit. Une telle infrastructures existe (voir, e.g., https://shop.dellemc.com/en-us/Product-Family/VxRail-Products/Dell-EMC-VxRail-Appliance/p/VCE-VxRail) mais elle est très chère (au moins $50K).

Pour aller plus loin, vous devez utiliser des outils dans les nuages (le cloud) fournis par Google, Amazon ou Microsoft par exemple.

> "We're particularly excited about the ability to use features like Google BigQuery's new Pearson correlation support to be able to search for patterns across the entire quarter-billion-record dataset in just seconds. (2014)"

Depuis 2018, on peut faire des régressions linéaires et multinomiales dans Google BigQuery (voir https://cloud.google.com/bigquery/docs/bigqueryml-analyst-start).

Voir aussi cet article très intéressant "Why Are We So Afraid of Petabytes?" (https://www.forbes.com/sites/kalevleetaru/2017/01/17/why-are-we-so-afraid-of-petabytes/#74e02e1c5875).

On ne couvre pas ces outils dans le cours car il faut avoir accès à des infrastructures spécialisés qui sont coûteuses.

2.2 Stratégies pour la Modélisation des Big Data

Une fois les données nettoyées et organisées, on se retrouve souvent avec une matrice massive d'observations.

- Si les données sont sparses (beaucoup de 0 ou de `NA`), les stocker en utilisant une méthode impliquant des matrices sparses.
- Si elles ne sont pas sparses, utiliser des bases de données distribuées et compressées. Plusieurs groupes développent des algorithmes rapides et des interfaces pour ces bases de données. Par exemple `H2O` (https://cran.r-project.org/web/packages/h2o/index.html) par H_2O interface R avec des versions très compressées des données en utilisant des implémentations basées sur JAVA de plusieurs outils de modélisation importants.

Nettoyer et organiser des données massives est une tâche difficile, très coûteuse en temps!

Les outils Linux sont très utiles dans ce contexte!

Vous devriez vous familiariser (e.g., utiliser `man cut`) au moins avec ces outils en ligne de commandes si votre bût est de travailler sur des projets de science en Big Data:

```
cut, head, sed, awk, diff, cat, tail, paste, sort, uniq, comm, tr, join
```

Nous aurons aussi besoin de `sponge`, qui peut être installé sur Debian comme suit:

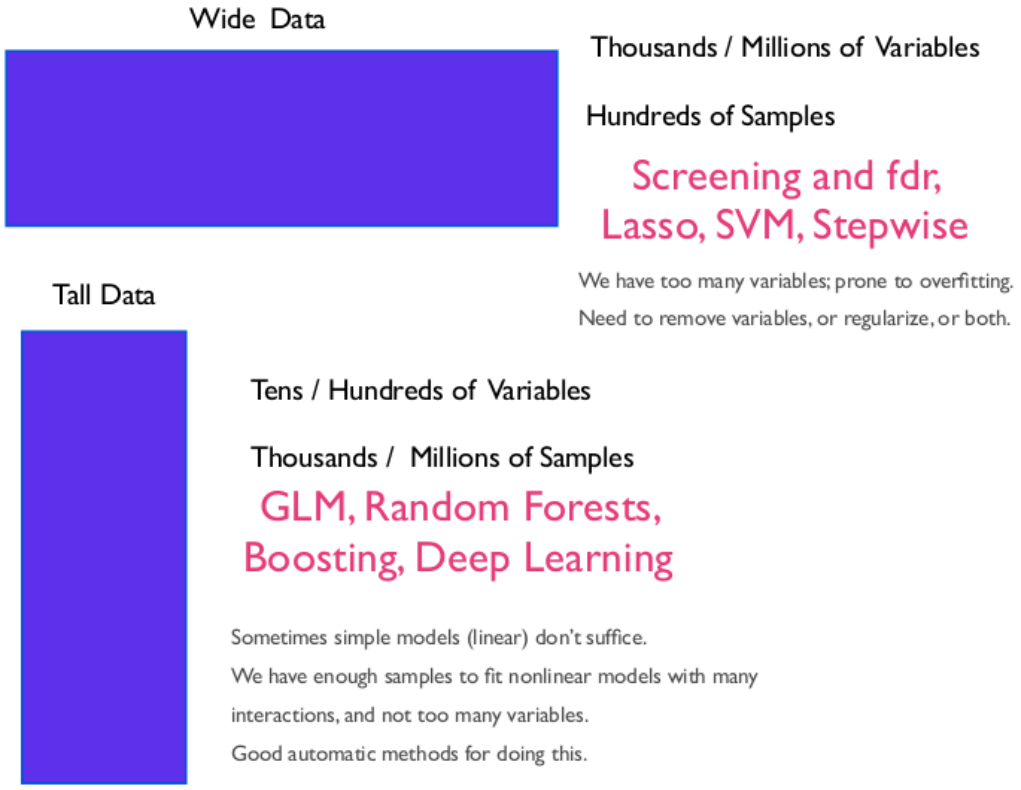
```
$ sudo apt-get install moreutils
```

```
sponge
```

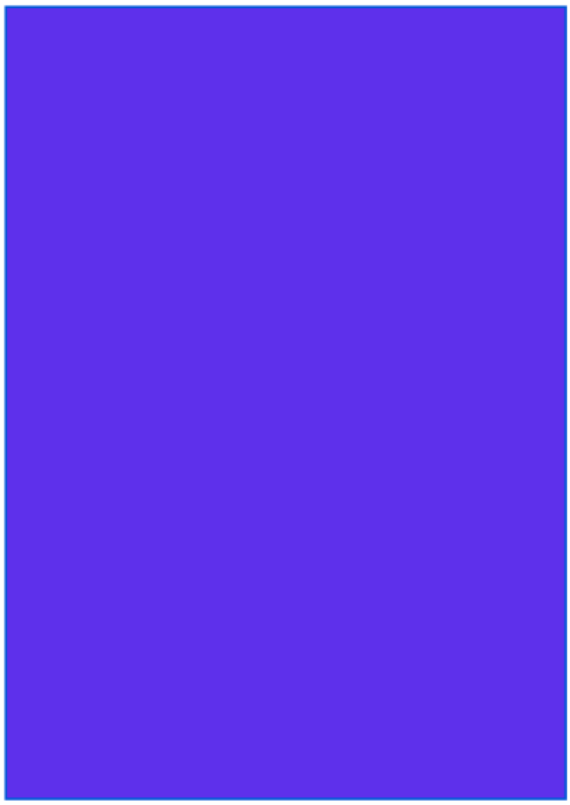
Note: si votre OS est Microsoft Windows, vous devriez installer une machine virtuelle avec Linux dedans (voir https://www.virtualbox.org/wiki/Downloads ou https://my.vmware.com/en/web/vmware/free#desktop_end_user_computing/vmware_workstation_player/15_0), ou même mieux, basculer complètement vers Linux!

Exercice: Lire la documentation (en anglais) des outils présentés sous l'onglet "Cleaning Your Data" à l'adresse http://datalyptus.com/DataScienceBook/

2.3 Les Big Data Varient de Forme. Des Approches Différentes Sont Nécessaires



Tall and Wide Data



Thousands / Millions of Variables

Millions to Billions of Samples

Tricks of the Trade

Exploit sparsity
Random projections / hashing
Variable screening
Subsample rows
Divide and recombine
Case/ control sampling
MapReduce
ADMM (divide and conquer)
.
.
.

3 Problèmes de Gestion de Mémoire avec R

Dans ce cours, nous allons travailler avec des gros fichiers, dont la taille est de l'ordre de 50% de la *mémoire physique* disponible (mais pas plus qu'une dizaine ou centaine de GB).

Noter que par défaut, R charge les fichiers dans la mémoire (RAM signifie Random Access Memory) et donc si un fichier est plus volumineux que la RAM disponible, des problèmes peuvent survenir. Aussi, même si un fichier peut rentrer dans la mémoire, quand vous commencez à travailler sur celui-ci, vous pouvez être amenés à en créer des copies (ou à créer de larges matrices en sortie de certaines analyses) ce qui peut rapidement conduire à une saturation de la RAM.

Le point de cette discussion sur la mémoire dans cette première leçon est de déterminer si votre ordinateur a suffisamment de mémoire pour réaliser le travail que vous voulez faire. Si vous déterminez que les données sur lesquelles vous travaillez ne peuvent pas être stockées entièrement dans la mémoire pour une session donnée de R, alors vous aurez sans doute besoin de recourir à d'autres tactiques (nous y reviendrons).

Une bonne compréhension de l'organisation de la RAM et de comment R l'utilise est crucial lorsque l'on veut analyser des Big Data. (C'est l'objet de ce premier cours.)

Quelques références:

- https://bookdown.org/rdpeng/RProgDA/the-role-of-physical-memory.html

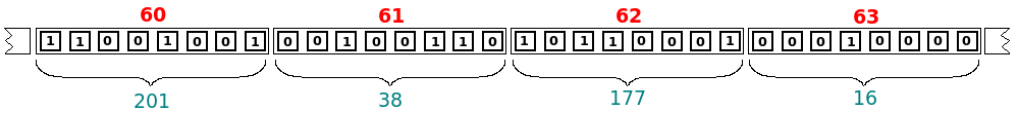
- http://www.matthewckeller.com/html/memory.html
- https://csgillespie.github.io/efficientR/hardware.html

3.1 Organisation de la RAM

La RAM est organisée comme une suite ordonnée de cases, et chaque case peut contenir un chiffre binaire: 0 ou 1. L'information contenue dans une case, la plus petite quantité d'information qui peut être contenue est appelée un *bit* (pour binary digit). À ce stade, il est bon de noter que l'information est en fait souvent organisée en *blocs* de 8 cases. Une autre unité a donc été introduite: le *byte* (octet), qui vaut 8 bits.

$$1\ byte = 8\ bits$$

Aussi, notez que chaque bloc est numéroté; ce numéro (d'un bloc de 8 cases) est appelé son *adresse mémoire*. Une adresse mémoire est donc un identifiant, qui désigne une zone de mémoire spécifique où les données (ou bien les instructions à exécuter) peuvent être lues et stockées. Cet identifiant est en général un entier (*integer*), souvent exprimé en notation hexadécimale (base $b = 16$) au lieu de la notation décimale usuelle (base $b = 10$).



Sur cette figure, chaque nombre en vert donne la représentation décimale de la valeur binaire du bloc au-dessus. Chaque nombre en rouge donne l'adresse (exprimée en notation décimale pour une meilleure compréhension) du bloc de 8 cases (ou 1 byte) situé en-dessous. Notez que ces adresses mémoire auraient aussi pû être écrites en notation hexadécimale, c'est-à-dire ici $60 = 3C$, $61 = 3D$, $62 = 3E$ et $63 = 3F$.

Exercice: écrire un petit code en R pour convertir du décimal en hexadécimal.

3.2 Accéder à la Mémoire

Pour accéder à une zone donnée de la mémoire, R utilise (d'une façon transparente, cachée à l'utilisateur) ce que l'on appelle un *pointeur* (une quantité qui "pointe" vers la zone mémoire désirée). Un pointeur est une variable qui contient une adresse mémoire. À l'adresse contenue dans un pointeur donné, on peut trouver par exemple la valeur d'une donnée. Notez que chaque donnée a un type spécifique, comme `integer`, `double`, `character`, etc.

En R, un `integer` est codé sur 4 bytes, un `double` sur 8 bytes, un `character` sur 1 byte, un `logical` sur 4 bytes, un `complex` sur 16 bytes, pour ne citer que les types les plus communs de variables.

```
object.size((rep(1L, 10000))) / 10000  # 1L dit à R de créer l'entier 1
## 4 bytes
object.size((rep(1.0, 10000))) / 10000 # Ici, c'est le nombre réel 1.0
## 8 bytes
object.size(paste(rep("a", 10000), collapse = "")) / 10000 # Caractères
## 1 bytes
object.size((rep(TRUE, 10000))) / 10000 # Logiques
## 4 bytes
```

```
object.size((rep(1i, 10000))) / 10000 # Nombres complexes
## 16 bytes
```

Cela a certaines conséquences, par exemple quand vous devrez lire un fichier qui contient seulement des valeurs entières. (E.g., ce sera le cas pour le jeu de données de Netflix que nous verrons plus tard.)

```
cat(7L, file = "/tmp/integer.txt", sep = "") # On écrit 7 dans le fichier
# On compte le nombre d'octets dans ce fichier:
system("wc -c /tmp/integer.txt", intern = TRUE) # 1 byte ici
## [1] "1 /tmp/integer.txt"
x <- scan("/tmp/integer.txt", what = integer())
typeof(x) # x est un entier, et comme expliqué plus haut 4 bytes sont réservés
## [1] "integer"
```

Comme vous pouvez le voir, un entier à un seul chiffre (0–9) occupe un seul byte dans un fichier texte mais occupera 4 bytes de mémoire dans R. Ainsi, regarder la taille d'un fichier sur le disque n'est pas forcément une façon fiable de savoir combien de mémoire sera nécessaire à R pour le lire.

Exercice: trouvez un fichier de données sur internet, regardez sa taille sur le disque, importez le depuis R. Êtes-vous surpris(s)?

3.3 Accéder à la Mémoire

Qu'arrive t-il lorsque l'on tape ces instructions?

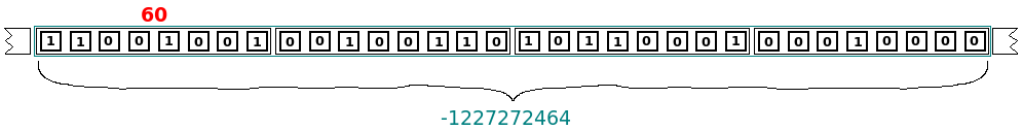
```
x <- 3L # crée la valeur 3, de type entier,
# ce qui est équivalent à:
x <- as.integer(3)
```

Étant donné ce que l'on vient d'expliquer, on peut supposer que, en même temps, un emplacement mémoire de 32 cases successives (4 bytes, de 8 bits chaque) est réservé par R (on dit alloué), et un pointeur est créé contenant l'adresse de la première de ces 32 cases.

Remarque: en fait, le pointeur ne doit pas seulement contenir l'adresse de la variable `x` , mais aussi son type (i.e., `integer`) pour savoir sur combien de cases la valeur dans cette variable est stockée. Pour cette raison, on parle de *pointeurs typés*. Quand un pointeur typé est incrémenté (i.e., quand on a besoin d'ajouter une unité à l'adresse qu'il contient), il n'est pas nécessairement incrémenté de 1, mais de la taille du type vers lequel il pointe.

3.4 Accéder à la Mémoire

Nous illustrons ci-dessous comment R stocke en mémoire un entier (signé).



Chaque petite case contient un chiffre binaire (0 ou 1). Le nombre en vert donne la représentation décimale de la valeur entière exprimée en notation binaire dans les quatre blocs (32 cases) du dessus. Le nombre rouge donne l'adresse (exprimée ici en base décimale) du premier bloc de 4 blocs mémoires au-dessous.

Notez que la toute première case est utilisée pour spécifier le signe du nombre (1 pour négatif et 0 pour positif; donc négatif ici).

```
dec2bin <- function(x) paste(as.integer(rev(intToBits(x))), collapse = "")
dec2bin(1227272464)
## [1] "01001001001001101011000100010000"
```

Astuce: vous pouvez utiliser ce convertisseur en ligne pour convertir des nombres décimaux en nombres binaires: https://www.rapidtables.com/convert/number/decimal-to-binary.html

Exercice: Codez votre propre convertisseur en R entre binaire et décimal (et vice-versa).

3.5 Problèmes Causés par la Gestion en Mémoire des Entiers

Puisqu'un entier (signé) est codé sur 4 bytes, i.e., 32 bits, le plus grand entier qui peut être représenté est 2147483647. En effet, si le premier bit est réservé pour le signe, il y a 31 cases restantes disponibles, et donc 2^{31} arrangements possibles. En comptant 0, le plus grand entier que l'on peut atteindre est donc $2^{31} - 1$:

```
as.integer(2 ^ 31 - 1)
## [1] 2147483647
# ce qui correspond en R à:
.Machine$integer.max
## [1] 2147483647
```

Le résultat ci-dessous n'est donc pas tellement surprenant.

```
as.integer(2 ^ 31)
## Warning: NAs introduced by coercion to integer range
## [1] NA
```

Le nombre 2^{31} (et les nombres entiers plus grands) peut donc seulement être traité en R comme un double:

```
2 ^ 31
## [1] 2147483648
# une conversion automatique est opérée par R ici:
is.double(2147483648L)
## [1] TRUE
```

3.6 Problèmes Causés par la Gestion en Mémoire des Entiers

Votre nouvelle connaissance du fonctionnement de R devrait vous aider à comprendre la sortie ci-dessous.

```
46360 * 46360 # 46360 est stocké comme un double puisqu'on n'a pas écrit 46360L.
## [1] 2149249600
46360L * 46360L # 46360L est stocké comme un entier.
```

```
## Warning in 46360L * 46360L: NAs produced by integer overflow
## [1] NA
sum(1:304)
## [1] 46360
# sum(1:304) est stocké comme un integer.
sum(1:304) * sum(1:304)
## Warning in sum(1:304) * sum(1:304): NAs produced by integer overflow
## [1] NA
46360 ^ 2 # Le résultat est stocké comme un double.
## [1] 2149249600
sum(1:304) ^ 2 # Le résultat est stocké comme un double.
## [1] 2149249600
```

Le *Warning* ci-dessus vient du fait que `sum(1:304)` est un *integer*.

Noter que la fonction exposant (`^`) transforme ses arguments en réels et renvoie un nombre réel (un `double`).

Noter que la fonction `*()` peut renvoyer soit un *integer*, soit un *double*, dépendamment de ses entrées:

```
is.integer("*"(2L, 3L))
## [1] TRUE
is.integer("*"(2, 3))
## [1] FALSE
```

Un autre problème potentiel pour la gestion des Big Data est le suivant. Imaginez un énorme vecteur colonne contenant 2^{55} valeurs.

```
is.integer(2 ^ 55)
## [1] FALSE
2 ^ 55
## [1] 3.60288e+16
a <- 2^55; b <- a + 2; b == a
## [1] TRUE
```

Exercice: Quel est le problème potentiel ici?

3.7 Allocation Successive de la Mémoire

Le plus petit bloc de mémoire qui peut être alloué (réservé) par R est en fait de 8 bytes (=64 bits). La memoire est donc allouée dans R par groupes de 8 blocs **successifs** (consécutifs) de 8 cases chaque.

Dans l'instruction `x <- 3L`, il y a donc 64 cases réservées (d'1 bit chaque), desquelles les 32 premières sont utilisées pour stocker la valeur entière `+3` (et les dernières 32 ne sont pas utilisées). Toutes les 64 cases seraient utilisées pour allouer un double, avec l'instruction `x <- 3.0`, ou pour allouer deux entiers, avec l'instruction `x <- c(1L, 2L)`.

```
if (!"TRSbook" %in% installed.packages()) install.packages("TRSbook")
x <- c(8L, 9L)
x
## [1] 8 9
```

```
addr <- TRSbook::getaddr(x) # Récupère l'adresse de la première case
                            # du bloc de 64 cases où x est stocké
addr
## <vector: 0x55a259c393a8>
addr$zero[1] # adresse au format entier
## [1] 1505989544
TRSbook::writeaddr(addr, 6L) # Écrit l'entier 6 à cette adresse
x
## [1] 6 9
TRSbook::writeaddr(addr + 4L, 7L) # Un entier est codé sur 4 bytes, donc
                                  # incrémenter l'adresse par 4 pour aller à x[2].
x
## [1] 6 7
```

3.8 Allocation Successive en Mémoire

Faisons la même chose pour des nombres réels (`double`).

```
x <- c(12.8, 4.5)
x
## [1] 12.8  4.5
addr <- TRSbook::getaddr(x) # Récupère l'adresse de la première case du
                            # bloc de 128 cases où x est stockée.
TRSbook::writeaddr(addr, 6.2)
x
## [1] 6.2 4.5
TRSbook::writeaddr(addr + 8L, 7.1) # Un double est codé sur 8 bytes.
x
## [1] 6.2 7.1
```

À ce stade, vous devriez avoir compris:

- comment la mémoire est organisée dans un ordinateur
- comment les nombres sont stockés dans la mémoire, de façon différente en fonction de leur type
- combien de mémoire chaque type de nombre (`integer`, `double`, etc.) occupe
- l'entier maximal qui peut être représenté avec R (`.Machine$integer.max`)

3.9 Taille d'un Objet dans R

On s'attend maintenant à ce que l'instruction `object.size(3L)` renvoie 8 bytes, mais ce n'est pas le cas.

```
object.size(3L) # (Sur un processeur 64 bit.)
## 56 bytes
object.size(3L) - 8
## 48 bytes
# Un objet "vide":
object.size(numeric())
## 48 bytes
```

En fait, chaque objet R contient (même lorsqu'on déclare un simple entier par `x <- integer(3)`) une entête (*header*) qui prend un peu de place dans la RAM (c'est-à-dire 48 bytes sur un processeur 64 bits) donné par l'instruction `object.size(numeric())`. Cet entête est utilisé pour sauver de l'information sur l'objet créé: son type (`integer`, `double`, `complex`, etc ..), sa longueur, etc ..

Note: Pour trouver sur quel type de processeur R est entrain de tourner, utiliser l'instruction suivante:

```
.Machine$sizeof.pointer
## [1] 8
```

La valeur 8 est retournée pour un processeur 64 bit et la valeur 4 pour un processeur 32 bit.

3.10 Taille d'un Objet dans R

L'allocation de mémoire dans R est faite de façon différente pour les petits et grands vecteurs d'entiers. Les petits vecteurs appartiennent à l'une de 6 classes, dépendamment de leur longueur (plus petite ou égale à 2, 4, 8, 12, 16 ou 32), et peuvent stocker des données de 8, 16, 32, 48, 64 ou 128 bytes respectivement. Puisqu'un entier utilise seulement 4 bytes, ces classes peuvent être utilisées pour stocker respectivement 2, 4, 8, 12, 16 et 32 entiers. Un vecteur d'entiers de longueur $n > 32$ utilisent un espace de taille jusqu'à $4n + 40$ si n est pair et $4(n + 1)$ si n est impair, auquel on ajoute une entête de 2 bytes sur un processeur 32 bit et de 48 bytes sur un processeur 64 bit.

```
N <- 50 ; V <- vector(length = 50)
for (L in 1:N) {
z <- sample(N, L, replace = TRUE)
V[L] <- object.size(z)
}
V - object.size(numeric()) # Sur un processeur 64 bit.
##  [1]   8   8  16  16  32  32  32  32  48  48  48  48  64  64  64  64 128 128
## [19] 128 128 128 128 128 128 128 128 128 128 128 128 128 128 136 136 144 144
## [37] 152 152 160 160 168 168 176 176 184 184 192 192 200 200
```

3.11 Composantes Partagées par Plusieurs Objets

Une subtilité de la taille d'un objet est que des composantes peuvent être partagées par plusieurs objets.

Cela peut avoir des conséquences importantes quand on doit travailler avec des gros objets!

Cela peut être vu grâce à la fonction `object_size()` du package `pryr` (mais pas avec la fonction `object.size()` du package `utils` que l'on a utilisé jusqu'ici).

```
if (!"pryr" %in% installed.packages()) install.packages("pryr")
# La fonction ":"() renvoie un vecteur d'entiers
x <- 1:1e6 # Le premier million d'entiers
object.size(x) # package utils: 4 bytes x 1M integers + 48 bytes pour le header
## 4000048 bytes
y <- list(x, x, x)
object.size(y) # le package utils est trompeur, comme on peut le voir ci-dessous!
```

```
## 12000224 bytes
3 * object.size(x)
## 12000144 bytes
pryr::object_size(x)
## 4 MB
pryr::object_size(y)
## 4 MB
```

L'objet `y` n'est pas trois fois plus gros que `x` car R est assez intelligent pour ne pas copier `x` trois fois; à la place il pointe juste vers le `x` existant. Cela signifie qu'il y a un ensemble de cases réservées en mémoire qui contiennent les valeurs de `x`. Cela signifie aussi que `x` et `y` contiennent la même adresse de (la première de) ces cases.

Il est trompeur de regarder les tailles de `x` et de `y` individuellement. Si vous voulez savoir combien d'espace elles occupent ensemble, vous devez les appeler ensemble avec `object_size()`.

```
pryr::object_size(x, y)
## 4 MB
```

3.12 Modification en Place

```
x <- 1:10
x[5] <- 10
x
##  [1]  1  2  3  4 10  6  7  8  9 10
```

Au-dessus, une modification *en place* a lieu. Cela signifie que la cinquième valeur 5 est modifiée directement dans l'ensemble de cases mémoire originellement allouée par R pour créer `x`. Aucune copie de `x` n'est faite.

Cependant, ci-dessous, R crée une copie de `x` à un autre endroit, modifie la copie, et ensuite utilise le nom `x` pour pointer ver le nouvel emplacement.

```
x <- 1:10
y <- x
c(pryr::address(x), pryr::address(y)) # Même adresse, pas de copie
## [1] "0x55a25abecf48" "0x55a25abecf48"
x[5] <- 10
x
##  [1]  1  2  3  4 10  6  7  8  9 10
c(pryr::address(x), pryr::address(y)) # Une copie est faite
## [1] "0x55a258c51228" "0x55a25abecf48"
```

Si des gros objets sont impliqués (e.g., des matrices), cela pourrait créer un problème de mémoire!

3.13 Boucles

Les boucles `for()` dans R ont la réputation d'être lentes. Souvent, cette lenteur est la conséquence de la modification d'une copie au lieu d'une modification en place.

```
x <- data.frame(matrix(runif(1000 * 1e4), ncol = 1000))
medians <- vapply(x, median, numeric(1))
system.time({
for(i in seq_along(medians)) {
  x[, i] <- x[, i] - medians[i] # Chaque itération de la boucle copie le data frame
}
})
##    user  system elapsed
##   0.097   0.028   0.124
y <- as.list(x)
system.time({
for(i in seq_along(medians)) {
  y[[i]] <- y[[i]] - medians[i] # Toutes les modifications ont lieu en place
}
})
##    user  system elapsed
##   0.019   0.000   0.019
```

Les listes sont meilleures que les data frames dans ce cas!

3.14 Mémoire Totale Utilisée par R

La **taille totale** de la mémoire virtuelle allouée à R dans une session inclue:

- la mémoire utilisée pour stocker les valeurs des objets (leur contenu);
- la mémoire utilisée pour stocker les entêtes des objets.

Cette information peut être accédée avec la fonction `gc()`, pour laquelle `Ncells` représente le nombre de cellules (Cells) utilisées pour le header, et `Vcells` le nombre de blocs pour les valeurs. Vous pouvez aussi utiliser la fonction `pryr::mem_used()`.

```
gc()
##            used  (Mb) gc trigger  (Mb) max used  (Mb)
## Ncells   655051  35.0    1331881  71.2  1331881  71.2
## Vcells 21216475 161.9   61965284 472.8 51233991 390.9
pryr::mem_used()
## 206 MB
```

3.15 Mémoire Totale Utilisée par R

La quantité totale de mémoire disponible pour R dépend de plusieurs facteurs:

- la RAM physiquement présente sur l'ordinateur (en acheter plus au besoin, si vous pouvez vous le permettre);
- la RAM déjà utilisée à la fois par l'OS et par les autres logiciels en cours d'exécution sur le système (comme par exemple un navigateur internet);
- le type de processeur (32 ou 64 bits), puisque la RAM est limitée à 4 GB (1 GB= 1024 KB) pour des processeurs 32 bit (et elle est plus souvent proche de 3 GB ou 2 GB), mais elle est beaucoup plus importante pour des processeurs 64 bit.

Sur Linux:

```
$ gnome-system-monitor &
```

Depuis R:

```
# Pour Windows OS seulement:
# system('wmic OS get FreePhysicalMemory /Value') # Mémoire disponible libre:
# system('wmic OS get TotalVisibleMemorySize /Value')" # Mémoire totale disponible
# Sur Linux / MAC OSX:
( installed.RAM <- as.numeric(system("awk '/MemTotal/ {print $2}' /proc/meminfo",
                                     intern = TRUE)) ) # In KB
## [1] 16301728
( available.RAM <- as.numeric(system("awk '/MemFree/ {print $2}' /proc/meminfo",
                                     intern = TRUE)) ) # In KB
## [1] 6536928
( used.RAM <- installed.RAM - available.RAM ) # In KB
## [1] 9764800
system("free -h", intern = TRUE)
## [1] "               total        used        free      shared  buff/cache   available"
## [2] "Mem:            15Gi       6.9Gi       6.2Gi       1.3Gi       2.4Gi       7.0Gi"
## [3] "Swap:           27Gi       7.0Gi        20Gi"
```

Chaque octet de mémoire a une adresse. Dans un processeur 32 bit, une adresse est codée sur 32 bits, donc la mémoire adressable est limitée à 2^{32} adresses d'1 byte chaque (=4 GB). Sur un processeur 64 bit, une adresse est codée sur 64 bits, donc théoriquement, la mémoire adressable est "limitée" à 2^{64} bytes, un nombre énorme. En fait, elle est en général limitée par l'architecture du processeur. Cette information peut être obtenue du fabricant (elle est appelée `Max Memory Size`).

Ne pas utiliser un processeur 32 bit pour traiter des big data!

Exercice: trouver la mémoie totale utilisée par R et celle disponible sur votre système.

3.16 Garbage Collection

Noter aussi que R alloue de la mémoire pour la création de gros objets, et vide la mémoire de ces objets (quand ils ne sont plus utilisés) via un procesus appelé *garbage collection*. Vous pouvez forcer le garbage collection avec la fonction `gc()` (parfois utile dans des boucles for).

Quand vous créez un gros objet, la mémoire réservée par R doit être contigue (elle ne peut pas être fragmentée en plusieurs blocs). Il est alors possible qu'il reste suffisamment de mémoire totale pour R, mais pas de "trou" assez grand pour faire rentrer les données dans un seul gros objet. C'était un gros problème pour les architectures 32 bit, mais plus vraiment pour les systèmes 64 bit.

Ne pas utiliser un processeur 32 bit pour traiter des big data!

3.17 Quelques Recommandations

Pour résumer:

- Utiliser Linux (ou MacOS), pas Microsoft Windows.

- Calculer **à l'avance** la taille (approximative) d'une matrice que vous prévoyez créer si vous anticipez qu'elle peut être très grande. Puisqu'un nombre réel utilise 8 bytes, une matrice de réels de taille $n \times p$ a besoin de $8np$ bytes de mémoire (sans l'entête).

Quelle sera la taille (en octets) d'une matrice avec 1,000 lignes et 10,000 colonnes?

```
M <- matrix(0.0, nrow = 1000, ncol = 10000)
prod(dim(M)) * 8
## [1] 8e+07
object.size(M) # In bytes
## 80000216 bytes
as.numeric(object.size(x)) / 1024 # En KB
## [1] 78242.78
as.numeric(object.size(x)) / 1024 ^ 2 # En MB
## [1] 76.40897
as.numeric(object.size(x)) / 1024 ^ 3 # En GB
## [1] 0.07461813
print(object.size(M), units = "MB")
## 76.3 Mb
```

- Calculer **à l'avance** la taille de l'objet R que vous obtiendrez après avoir lu un fichier, e.g., en utilisant l'outil en ligne de commande tel que `wc -c filename`. Si vous lisez accidentellement un jeu de données qui requiert plus de mémoire que celle disponible sur votre ordinateur, vous risque de geler votre session R (ou même votre ordinateur).
- Si vous devez travailler avec des matrices très volumineuses, un processeur 64 bit vous permettra d'allouer de gros blocs de mémoire. Comparer ceci avec un processeur 32 bit qui ne vous permettra pas en général d'allouer plus que 2 gigabytes. **Utiliser un ordinateur 64-bit, avec un OS 64 bit et une version 64 bit de R**.
- Si vous n'êtes pas capables de créer un gros objet en R, commencer par supprimer (en utilisant la fonction `rm()`) d'autres gros objets inutilisés (la fonction `pryr::object_size()` donne la taille de tels objets). Libérer de la mémoire avec la fonction `gc()` (probablement pas nécessaire cependant). Vous pouvez aussi fermer d'autres logiciels entrain de tourner sur votre ordinateur pour libérer un peu de mémoire, et en dernier recours acheter plus de mémoire physique. Une autre option serait de couper votre matrice en plusieurs sous-matrices et trouver un moyen d'effectuer votre analyse sur celles-ci, avant de recombiner les résultats. Nous y reviendrons.
- Lire `R> ?"Memory-limits"`
- Utiliser un package conçu pour stocker des objets sur disque dur plutôt que dans la RAM (`ff`, `filehash`, `R.huge`, ou `bigmemory`). Nous y reviendrons.
- Utiliser des **matrices sparses** et une librairie conçue pour travailler avec de telles matrices. Nous y reviendrons.
- Garder souvent un oeil sur l'utilisation des ressources avec la fonction unix `top` (ou `gnome-system-monitor`) pour voir en temps réel combien de RAM votre session R consomme.

Exercice: Ré-explorez les concepts vus jusqu'à présent. Fabriquez des matrices ou d'autres types d'objets R de taille croissante pour mieux appréhender les limites de votre ordinateur/système en termes de gestion de Big Data avec R. Fabriquez des exemples "tordus" et demandez à l'un de vos camarades s'il comprend vraiment l'origine du problème. Comprenez-vous ses exemples à lui (ou elle)?