Calculs statistiques et mathématiques en R

Sophie Baillargeon, Université Laval

2019-03-04

Table des matières

Calculs statistiques

L'utilité première du logiciel R est la réalisation de calculs statistiques. Les principaux outils pour réaliser de tels calculs sont présentés ici. Cette section traite de fonctions permettant de manipuler des distributions de probabilité, de générer des nombres pseudo-aléatoires, de réaliser de tests statistiques et d'ajuster des modèles statistiques.

Distributions de probabilité

Le package `stats` de l'installation de base de R comprend, pour plusieurs distributions de probabilité, des fonctions R de calcul de :

- la fonction de densité (forme `dxxx` où `xxx` change selon la distribution),
- la fonction de répartition (forme `pxxx`) et
- la fonction quantile (forme `qxxx`).

La fiche d'aide ouverte par la commande `help(Distributions)` énumère toutes les distributions de probabilité offertes dans le package `stats`. Il existe aussi des fonctions relatives à d'autres distributions de probabilité dans des packages sur le CRAN (voir https://CRAN.R-project.org/view=Distributions pour découvrir ce qui est offert).

Ces fonctions sont utiles notamment pour calculer des valeurs critiques ou des seuils observés de tests d'hypothèses. Un exemple est présenté plus loin.

Fonction de densité

Les fonctions R implémentant des fonctions de densité ont un nom qui débute par le lettre `d` pour *density*. Leur premier argument est toujours un vecteur de valeurs en lesquelles calculer la fonction de densité. Les arguments suivants servent à spécifier les valeurs des paramètres de la distribution.

Dans le cas d'une variable aléatoire discrète, la fonction de densité est plus justement appelée fonction de masse. Il s'agit alors d'une probabilité, pour une variable aléatoire suivant une certaine distribution, de prendre une certaine valeur.

Exemple : Distribution binomiale

Soit X une variable aléatoire représentant le nombre de 6 obtenus lors de 5 lancés d'un dé. Cette variable aléatoire suit une distribution binomiale de paramètres $n = 5$ et $p = 1/6$, donc $X \sim Bin(5, 1/6)$.

Calculons $P(X = 2)$, soit la probabilité que la variable aléatoire X prenne la valeur 2.

```
dbinom(x = 2, size = 5, prob = 1/6)
```

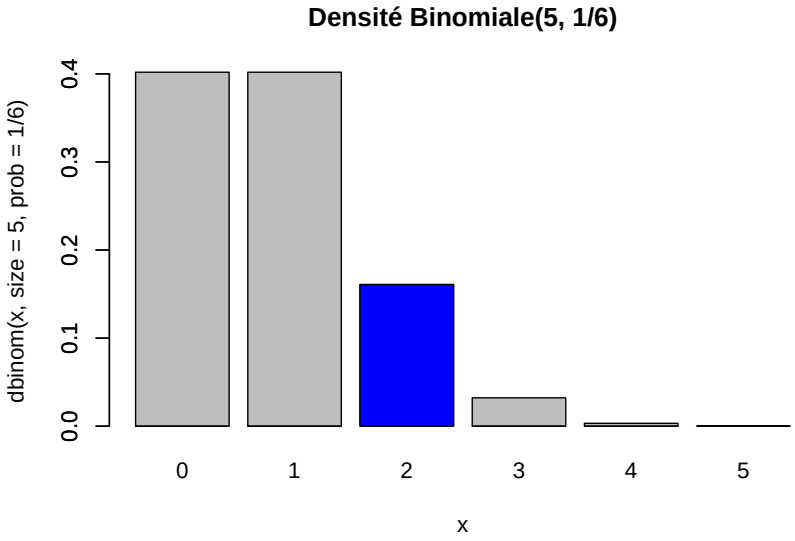
```
## [1] 0.160751
```

La fonction `dbinom` implémente donc la fonction de masse d'une distribution binomiale. Les arguments `size` et `prob` sont les paramètres n et p de la distribution, selon la notation utilisée ci-dessus.

Les fonctions de la famille `dxxx` peuvent calculer plusieurs valeurs de densité par un seul appel de la fonction, car celles-ci acceptent des valeurs d'arguments de longueur supérieure à 1. Ces fonctions travaillent donc de façon vectorielle, comme presque toutes les fonctions de calcul en R.

Voici un exemple de code R permettant de représenter graphiquement la densité $Bin(5, 1/6)$ complète, en mettant en évidence la valeur calculée ci-dessus, soit $P(X = 2)$.

```
barplot(dbinom(0:5, size = 5, prob = 1/6),
        names.arg = 0:5, main = "Densité Binomiale(5, 1/6)",
        xlab = "x", ylab = "dbinom(x, size = 5, prob = 1/6)")
barplot(c(0, 0, dbinom(2, size = 5, prob = 1/6), 0, 0),
        col = "blue", add = TRUE)
```



Exemple : Distribution normale standard

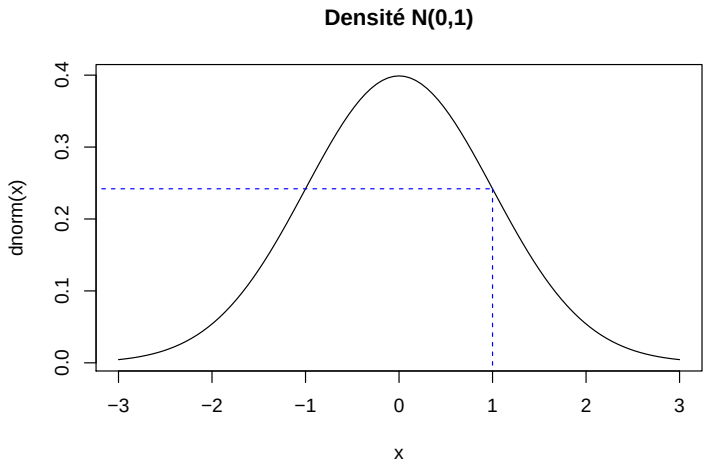
Supposons maintenant que X est une variable aléatoire continue de distribution normale standard, donc $X \sim N(0,1)$. La valeur de la fonction de densité pour cette distribution en la valeur $x = 1$, souvent notée $f_X(1)$, vaut :

```
dnorm(x = 1)
```

```
## [1] 0.2419707
```

Voici une représentation graphique de la densité complète dans laquelle la valeur calculée ci-dessus est mise en évidence.

```
curve(dnorm, xlim = c(-3, 3), main = "Densité N(0,1)")
segments(-4, dnorm(1), 1, dnorm(1), lty = 2, col = "blue")
segments(1, dnorm(1), 1, -1, lty = 2, col = "blue")
```



Ici, nous n'avons pas eu besoin de fournir des valeurs aux arguments de la fonction `dnorm` relatifs aux paramètres de la distribution, parce que nous avons utilisés leurs valeurs par défaut. Ces paramètres, pour la

densité $N(\mu, \sigma^2)$, sont représentés par les arguments `mean` $= \mu$ et `sd` $= \sigma$ (remarquez que l'argument de la fonction R représente l'écart-type, pas la variance).

Tout comme le premier argument, nommé x, les arguments des fonctions de la famille `dxxx` représentant des paramètres de la distribution acceptent aussi en entrée plus d'une valeur. Voici un exemple, qui permet de calculer la densité en $x = 1$ pour $X \sim N(\mu = -2, \sigma^2 = 1)$, $X \sim N(\mu = 0, \sigma^2 = 2.25)$ et $X \sim N(\mu = 1, \sigma^2 = 4)$ en un seul appel à la fonction `dnorm`.

```
dnorm(x = 1, mean = c(-2, 0, 1), sd = c(1, 1.5, 2))
```

```
## [1] 0.004431848 0.212965337 0.199471140
```

Fonction de répartition

La fonction de répartition d'une variable aléatoire X est définie par $F_X(x) = P(X \leq x)$. Il s'agit donc toujours d'une probabilité, d'où le `p` au début des noms des fonctions R implémentant des fonctions de répartition.

Exemple : Distribution normale standard

Prenons encore comme exemple la distribution normale standard. Nous avons donc $X \sim N(0, 1)$. Calculons la valeur de la fonction de répartition de cette variable aléatoire en $x = 1$.

```
pnorm(q = 1)
```

```
## [1] 0.8413447
```

Il s'agit de la valeur de la probabilité $P(X \leq 1)$.

Le premier argument des fonctions de la famille `pxxx` ne se nomme pas `x`, il se nomme plutôt `q`. Cette lettre réfère au mot quantile et souligne le lien entre les fonctions de répartition et les fonctions quantiles. Les arguments suivants des fonctions de la famille `pxxx` permettent de spécifier les valeurs des paramètres de la distribution, comme pour les fonctions de la famille `pxxx`.

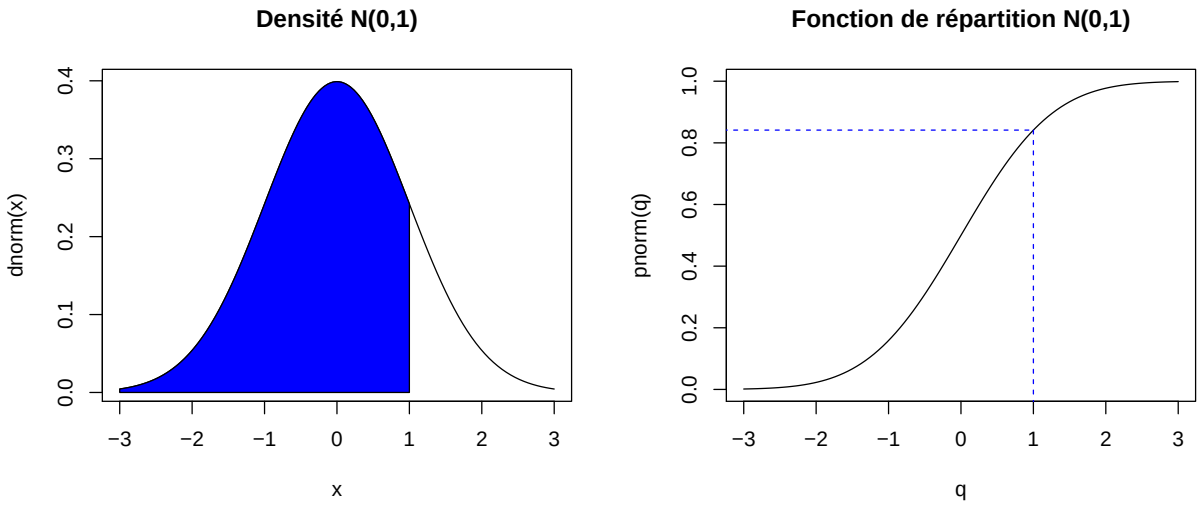
Voici une illustration graphique du lien entre la fonction de densité et la fonction de répartition pour la distribution normale standard.

```
par.default <- par(mfrow = c(1,2))

curve(dnorm, xlim = c(-3, 3), main = "Densité N(0,1)")
x <- seq(-3, 1, length = 100)
polygon(c(x, 1, -3), c(dnorm(x), 0, 0), col = "blue")

curve(pnorm, xlim = c(-3, 3), main = "Fonction de répartition N(0,1)", xname = "q")
segments(-4, pnorm(1), 1, pnorm(1), lty = 2, col = "blue")
segments(1, pnorm(1), 1, -1, lty = 2, col = "blue")

par(par.default)
```



Fonction quantile

La fonction quantile est l'inverse généralisé de la fonction de répartition. Les fonctions R implémentant des fonctions quantile ont un nom qui débute par le lettre `q` pour *quantile*.

Exemple : Distribution normale standard

Pour clore l'exemple de la distribution normale standard, voyons de quoi à l'air la fonction quantile de cette distribution.

Premièrement, calculons la valeur de la fonction quantile en un point, disons en $p = 0.8413447$.

```
qnorm(p = 0.8413447)
```

```
## [1] 0.9999998
```

Il s'agit de la valeur x pour laquelle $P(X \leq x) = 0.8413447$, où $X \sim N(0, 1)$.

Le premier argument d'une fonction de la famille `qxxx` se nomme `p`. Cette notation peut nous aider à nous rappeler que cet argument représente une probabilité et accepte donc seulement des valeurs entre 0 et 1. Encore une fois, les arguments suivants des fonctions de la famille `qxxx` permettent de spécifier les valeurs des paramètres de la distribution.

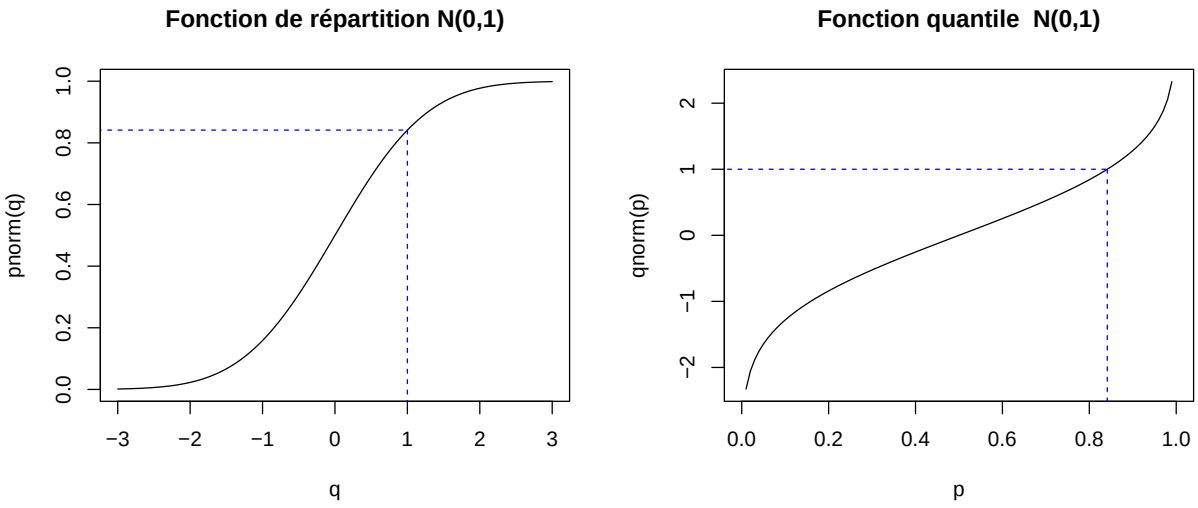
Le graphique suivant illustre le lien entre la fonction de répartition et la fonction quantile.

```
par.default <- par(mfrow = c(1,2))

curve(pnorm, xlim = c(-3, 3), main = "Fonction de répartition N(0,1)", xname = "q")
segments(-4, pnorm(1), 1, pnorm(1), lty = 2, col = "blue")
segments(1, pnorm(1), 1, -1, lty = 2, col = "blue")

curve(qnorm, xlim = c(0, 1), main = "Fonction quantile  N(0,1)", xname = "p")
segments(-0.2, qnorm(pnorm(1)), pnorm(1), qnorm(pnorm(1)), lty = 2, col = "blue")
segments(pnorm(1), qnorm(pnorm(1)), pnorm(1), -3, lty = 2, col = "blue")

par(par.default)
```



Génération de nombres pseudo-aléatoires

En R, les fonctions `rxxx` permettent de générer pseudo-aléatoirement des observations selon une certaine distribution désignée par `xxx`.

Par exemple, voici la représentation graphique de 3 échantillons générés aléatoirement selon 3 distributions différentes : des distributions normale, uniforme continue et khi-deux. Pour chaque échantillon, nous traçons l'histogramme des observations simulées pour représenter leur densité empirique. Nous superposons à cet histogramme la courbe de densité de la distribution théorique à partir de laquelle les observations ont été générées.

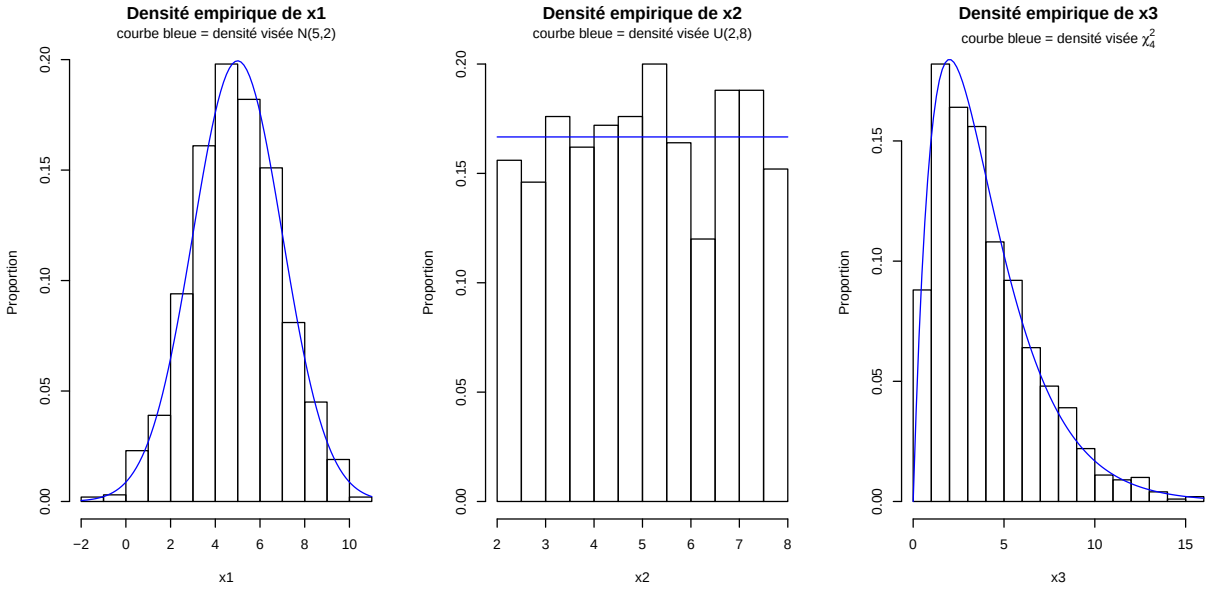
```r
par.default <- par(mfrow = c(1, 3))

# Densité normale d'espérance 5 et de variance 4
x1 <- rnorm(1000, mean = 5, sd = 2)
hist(x1, freq = FALSE, ylab = "Proportion", main = "Densité empirique de x1")
curve(dnorm(x, mean = 5, sd = 2), add = TRUE, col = "blue")
title(main = "courbe bleue = densité visée N(5,2)", line = 0.5)

# Densité uniforme continue entre 2 et 8
x2 <- runif(1000, min = 2, max = 8)
hist(x2, freq = FALSE, ylab = "Proportion", main = "Densité empirique de x2")
curve(dunif(x, min = 2, max = 8), add = TRUE, col = "blue")
title(main = "courbe bleue = densité visée U(2,8)", line = 0.5)

# Densité chi-carré à 4 degrés de liberté
x3 <- rchisq(1000, df = 4)
hist(x3, freq = FALSE, ylab = "Proportion", main = "Densité empirique de x3")
curve(dchisq(x, df = 4), add = TRUE, col = "blue")
title(main = expression(paste("courbe bleue = densité visée ", chi[4]^2)), line = 0.5)

par(par.default)
```



Comme nous pouvons le constater sur ces graphiques, la distribution empirique des observations générées avec ces fonctions se rapproche vraiment de la distribution théorique demandée. En générant un nombre encore plus grand d'observations (ici nous en avons généré 1000 pour chaque distribution), la densité empirique se rapprocherait encore plus de la densité théorique.

Fonction `sample`

La fonction `sample` permet de tirer un échantillon aléatoire de taille déterminée par l'argument `size`,

```
sample(1:6, size = 6)
```

```
## [1] 3 6 5 1 2 4
```

avec ou sans (par défaut) remise grâce à l'argument `replace`,

```
sample(1:6, size = 6, replace = TRUE)
```

```
## [1] 1 3 3 4 3 3
```

en utilisant des probabilités de sélection égales (par défaut) entre les éléments de l'ensemble de départ ou non grâce à l'argument `prob`.

```
sample(1:6, size = 20, replace = TRUE, prob = c(1/2, rep(1/10, 5)))
```

```
##  [1] 1 2 2 4 6 1 4 1 5 6 4 2 1 2 1 2 2 6 1 3
```

La fonction `sample` prend comme premier argument un vecteur représentant l'ensemble des éléments à partir desquels faire le tirage aléatoire. Ce vecteur peut contenir des données de n'importe quel type.

```
sample(c("Luc", "Kim", "Paul", "Ève"))
```

```
## [1] "Luc"  "Ève"  "Kim"  "Paul"
```

Si l'argument `size` n'est pas spécifié, il prend comme valeur par défaut la longueur du vecteur fourni au premier argument. Remarquez que lorsque la taille de l'échantillon est égale à la taille de l'ensemble de départ et que l'échantion est tiré sans remise, cet échantillon est en fait une permutation aléatoire des éléments de l'ensemble de départ.

La fonction `sample.int` est très similaire à la fonction `sample`, mais elle prend comme premier argument un seul entier, `n`, et tire aléatoirement `size` entiers entre 1 et `n`.

```
sample.int(n = 6, size = 4)
```

```
## [1] 1 4 6 5
```

Si nous souhaitons sélectionner aléatoirement des observations (lignes) dans un jeu de données, il est d'usage de sélectionner d'abord des entiers compris entre 1 et le nombre total d'observations, puis d'extraire du jeu de donnée les observations sur les lignes portant les numéros sélectionnés. Voici un exemple, utilisant les données sur les iris de Fisher incluses dans l'installation de base de R.

```
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```
index_ech <- sample.int(n = nrow(iris), size = 5, replace = FALSE)
index_ech
```

```
## [1] 61 99 16 89 81
```

```
iris_ech <- iris[index_ech, ]
iris_ech
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 61          5.0         2.0          3.5         1.0 versicolor
## 99          5.1         2.5          3.0         1.1 versicolor
## 16          5.7         4.4          1.5         0.4     setosa
## 89          5.6         3.0          4.1         1.3 versicolor
## 81          5.5         2.4          3.8         1.1 versicolor
```

Germe de la génération pseudo-aléatoire

Les nombres générés avec les fonctions `rxxx` et les échantillons tirés avec `sample` sont qualifiés de pseudo-aléatoires, car ils proviennent d'un algorithme déterministe qui tente de reproduire le hasard. Un tel algorithme est nommé en anglais *random number generator* (RNG) ou *pseudo-random number generator*. La fiche d'aide ouverte par la commande `help(Random)` contient de l'information sur la génération de nombres pseudo-aléatoires en R.

Plusieurs RNG sont implémentés en R. Celui utilisé par défaut, nommé Mersenne Twister, a été choisi parce qu'il est réputé être bon. Sans entrer dans les détails du fonctionnement des RNG implémentés dans le package `base` de R, il faut savoir qu'ils travaillent tous à partir d'une séquence de nombres appelée *germe* (en anglais *seed*). En contrôlant ce germe, il est possible de générer de nouveau, autant de fois que désiré, les mêmes valeurs.

Par défaut, R contrôle le germe des RNG de façon automatique. Chaque fois qu'une commande faisant intervenir un RNG est évaluée, R crée un nouveau germe à partir, notamment, de l'heure à laquelle la commande est soumise. Ainsi, deux générations pseudo-aléatoires consécutives ne produisent en général pas le même résultat.

```
sample(letters, size = 5)
```

```
## [1] "z" "x" "v" "m" "e"
```

```
sample(letters, size = 5)
```

```
## [1] "c" "k" "g" "l" "z"
```

En tout temps, il est possible de connaître le germe du RNG en R. Il est stocké dans un objet nommé `.Random.seed`. Cet objet est un vecteur numérique de longueur 626 pour le RNG Mersenne Twister. Voyons de quoi ont l'air les premiers éléments de ce vecteur à différents moments.

```
str(.Random.seed)
```

```
##  int [1:626] 403 55 -2071106341 1367273273 -524105080 1511337526 -455188879 975699107..
```

```
sample(letters, size = 5)
```

```
## [1] "a" "r" "u" "e" "k"
```

```
str(.Random.seed)
```

```
##  int [1:626] 403 60 -2071106341 1367273273 -524105080 1511337526 -455188879 975699107..
```

```
sample(letters, size = 5)
```

```
## [1] "e" "i" "k" "o" "j"
```

```
str(.Random.seed)
```

```
##  int [1:626] 403 65 -2071106341 1367273273 -524105080 1511337526 -455188879 975699107..
```

Nous constatons qu'au moins un élément de ce vecteur (le deuxième élément) change à chaque fois que nous appelons la fonction `sample`. La fonction `set.seed` permet de fixer le germe du RNG à partir d'une seule valeur entière. Par exemple, la commande suivante :

```
set.seed(753)
```

spécifie le germe suivant :

```
str(.Random.seed)
```

```
##  int [1:626] 403 624 302719261 615841082 -1828406925 -1314498536 214287161 142621094 ..
```

En soumettant de nouveau le même appel à la fonction `sample`, nous obtenons l'échantillon suivant.

```
sample(letters, size = 5)
```

```
## [1] "n" "v" "m" "x" "g"
```

La soumission de cette commande a eu pour effet de modifier le germe.

```
str(.Random.seed)
```

```
##  int [1:626] 403 5 47777699 -819670847 -326033232 523235278 1163109177 1104276299 186..
```

Donc si nous resoumettons la commande `sample`, nous n'obtiendrons sûrement pas le même résultat.

```
sample(letters, size = 5)
```

```
## [1] "e" "j" "o" "i" "g"
```

Par contre, si nous fixons de nouveau le germe à partir de l'entier 753 avec `set.seed`, nous arriverons à obtenir de nouveau l'avant-dernier échantillon généré.

```
set.seed(753)
sample(letters, size = 5)
```

```
## [1] "n" "v" "m" "x" "g"
```

À n'importe quel moment, dans n'importe qu'elle session R, nous obtiendrons l'échantillon {n, v, m, x, g} si nous soumettons la commande `set.seed(753)` avant la commande `sample(letters, size = 5)`.

L'utilité de fixer le germe d'une génération pseudo-aléatoire est de rendre son résultat reproduisible.

Calcul de statistiques descriptives

Les fonctions pour des calculs de statistiques descriptives de base ont été vues dans les notes intitulées Calculs de base en R.

Tests statistiques

Il existe des fonctions R pour faire des tests statistiques de base. En voici quelques-unes :

- tests sur une ou des moyennes : `t.test` ;
- tests sur une ou des proportions : `prop.test`, `binom.test` ;
- tests de comparaison de variances : `var.test`, `bartlett.test` ;
- tests sur une corrélation : `cor.test` ;
- tests pour une distribution : `shapiro.test`, `ks.test` ;
- tests non paramétriques : `wilcox.test`, `kruskal.test`, `friedman.test` ;
- tests sur des fréquences : `chisq.test`, `fisher.test`, `mantelhaen.test`.

Utilisons encore une fois les données `iris` pour construire un exemple de test de comparaison de moyennes. Nous allons comparer les largeurs moyennes des sépales des espèces versicolor et virginica.

Test t bilatéral de comparaison de moyennes, avec variances inégales :

```
t.test(Sepal.Width ~ Species, data = iris,
       subset = Species %in% c("versicolor", "virginica"))
```

```
##
##  Welch Two Sample t-test
##
## data:  Sepal.Width by Species
## t = -3.2058, df = 97.927, p-value = 0.001819
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.33028364 -0.07771636
## sample estimates:
## mean in group versicolor  mean in group virginica
##                    2.770                    2.974
```

Lorsque nous connaissons la valeur d'une statistique de test ainsi que sa loi sous l'hypothèse nulle, il est possible de calculer en R le seuil observé du test avec la bonne fonction `pxxx`. Par exemple, dans le test t ci-dessus, nous pouvons retrouver la valeur du seuil observé ainsi :

```
# multiplication par 2, car le test est bilatéral et la loi t est symétrique
pt(-3.2058, df = 97.927)*2
```

```
## [1] 0.001819258
```

Test non paramétrique équivalent :

```
wilcox.test(Sepal.Width ~ Species, data = iris,
            subset = Species %in% c("versicolor", "virginica"), exact = FALSE)
```

```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  Sepal.Width by Species
## W = 841, p-value = 0.004572
## alternative hypothesis: true location shift is not equal to 0
```

Ajustement de modèles

Il existe plusieurs fonctions en R pour ajuster des modèles. Les modèles les plus usuels sont :

- modèles de régression : `lm` ;
- modèle d'analyse de la variance : `lm` ou `aov` ;
- modèles linéaires généralisés : `glm` ;
- modèles linéaires mixtes (peuvent contenir des effets aléatoires) : `lmer` du package `lme4` ou `lme` du package `nlme` ;
- modèles non linéaires : `nls`.

Par exemple, ajustons quelques modèles sur les données des iris de Fisher.

Régression linéaire simple entre la largeur et la longueur des sépales :

```
reg <- lm(Sepal.Width ~ Sepal.Length, data = iris)
reg
```

```
##
## Call:
## lm(formula = Sepal.Width ~ Sepal.Length, data = iris)
##
## Coefficients:
##  (Intercept)  Sepal.Length
##      3.41895      -0.06188
```

ANOVA pour comparer les largeurs de sépales moyennes entre toutes les espèces :

```
ANOVA <- aov(Sepal.Width ~ Species, data = iris)
ANOVA
```

```
## Call:
##    aov(formula = Sepal.Width ~ Species, data = iris)
##
## Terms:
##                   Species Residuals
## Sum of Squares  11.34493  16.96200
## Deg. of Freedom        2       147
##
## Residual standard error: 0.3396877
## Estimated effects may be unbalanced
```

Formules

Ces fonctions d'ajustement de modèles prennent obligatoirement en entrée une formule R, tout comme la fonction `xtabs`. D'autres fonctions acceptent aussi en entrée une formule, sans que ce type d'argument soit obligatoire. C'est le cas des fonctions `aggregate`, `ftable`, plusieurs des fonctions effectuant un test (ex. `t.test`, `wilcox.test`) et certaines fonctions graphiques (ex. `plot`, `boxplot`).

Les fonctions prenant une formule en entrée ont toujours un argument `data` pour spécifier d'où tirer les variables incluses dans la formule.

Une formule s'écrit sous la forme `y ~ x1 + x2` où `y` représente la variable réponse (dépendante) et `x1` et `x2` des variables explicatives (indépendantes). Dans la partie de droite, les opérateurs suivants peuvent apparaître :

- `+` pour ajouter des termes ;
- `-` pour soustraire des termes ;
- `0` ou `1` pour représenter l'ordonnée à l'origine (par défaut tout modèle comporte une ordonnée à l'origine, pour la retirer il faut ajouter `- 1` ou `+ 0` à la partie de droite de la formule) ;
- `.` pour représenter toutes les variables dans le jeu de données fournit en argument `data`, autres que la variable mise à la gauche du `~` (note : le point signifie autre chose dans la fonction `update`).

Opérateurs propres aux facteurs :

- `:` pour les termes d'interaction entre facteurs ;
- `*` pour le croisement de facteurs (`x1*x2` est équivalent à `x1 + x2 + x1:x2`) ;
- `^` pour le croisement de facteurs jusqu'à un certain niveau d'interaction (par exemple `(x1 + x2 + x3)^2` va inclure tous les termes de croisement des facteurs jusqu'aux interactions doubles `x1 + x2 + x3 + x1:x2 + x1:x3 + x2:x3`, mais pas l'interaction triple `x1:x2:x3`) ;
- `%in%` pour les facteurs emboîtés (dans `x2 %in% x1`, `x2` est emboîté dans `x1`).

Autre opérateur :

- `|` n'a pas toujours exactement la même signification selon la fonction, il représente parfois :
  - un conditionnement par rapport à une variable (ex. `y ~ x | a`, fonction `coplot`),
  - la structure d'effets aléatoires (fonctions `lmer` du package `lme4` et `lme` du package `nlme`).

Les formules peuvent inclurent des appels à des fonctions pour transformer les variables. Par exemple, pour ajuster un modèle sur la racine carrée de la variable réponse, nous pouvons écrire `sqrt(y) ~ x1 + x2`. Certaines transformations pourraient faire intervenir un ou des opérateurs ayant une signification modifiée dans une formule. Par exemple, imaginons que nous voulons ajuster un modèle avec une seule variable explicative créée en additionnant les valeurs des variables `x1` et `x2`. La formule `y ~ x1 + x2` n'ajuste pas ce modèle puisque, dans une formule, l'opérateur `+` signifie « ajouter des termes » et non plus « additionner des valeurs ». Alors est-il possible d'ajuster le modèle souhaité (*sans ajouter une nouvelle variable dans les données*) ?

Oui, c'est possible grâce à la fonction `I`. Il faut encadrer l'opération arithmétique à effectuer dans la formule d'un appel à la fonction `I`. Par exemple, `y ~ I(x1 + x2)` ajuste un modèle à une seule variable explicative formée de la somme des valeurs de `x1` et `x2`. Ainsi, `I()` permet d'utiliser la signification usuelle des opérateurs et non celle spécifique aux formules.

Voici quelques exemples :

Retrait de l'ordonnée à l'origine :

```
reg <- lm(Sepal.Width ~ Sepal.Length - 1, data = iris)
# ou encore :
reg <- lm(Sepal.Width ~ Sepal.Length + 0, data = iris)
```

```
coef(summary(reg))
```

```
##                Estimate  Std. Error  t value      Pr(>|t|)
## Sepal.Length 0.5117739 0.008939966 57.24562 2.422615e-103
```

Régression log-log :

```
reg <- lm(log(Sepal.Width) ~ log(Sepal.Length), data = iris)
```

```
coef(summary(reg))
```

```
##                     Estimate Std. Error   t value     Pr(>|t|)
## (Intercept)        1.3057236 0.14573780  8.959402 1.293792e-15
## log(Sepal.Length) -0.1129573 0.08275742 -1.364920 1.743495e-01
```

Régression polynomiale :

```
reg <- lm(Sepal.Width ~ Sepal.Length + Sepal.Length^2, data = iris)
```

```
coef(summary(reg))
```

```
##                Estimate Std. Error   t value     Pr(>|t|)
## (Intercept)   3.4189468 0.25356227 13.483658 1.552431e-27
## Sepal.Length -0.0618848 0.04296699 -1.440287 1.518983e-01
```

Non, ça n'a pas fonctionné. Dans une formule, l'opérateur ^ ne signifie pas exposant. Pour demander à R d'utiliser la signification usuelle de l'opérateur, et non celle spécifique aux formules, il faut encadrer le terme contenant l'opérateur de `I()` comme suit :

```
reg <- lm(Sepal.Width ~ Sepal.Length + I(Sepal.Length^2), data = iris)
```

```
coef(summary(reg))
```

```
##                      Estimate Std. Error   t value     Pr(>|t|)
## (Intercept)        6.41583572 1.58499197  4.047866 8.327686e-05
## Sepal.Length      -1.08556027 0.53624556 -2.024372 4.474291e-02
## I(Sepal.Length^2)  0.08570656 0.04475502  1.915016 5.743267e-02
```

Création de variables catégoriques pour les exemples à venir :

```
iris$Sepal.Length_catego <- cut(iris$Sepal.Length, right = FALSE,
     breaks = c(-Inf, quantile(iris$Sepal.Length, probs = c(1/3, 2/3)), Inf))
iris$Petal.Width_catego <- cut(iris$Petal.Width, right = FALSE,
     breaks = c(-Inf, quantile(iris$Petal.Width, probs = c(1/3, 2/3)), Inf))
```

Anova à 3 facteurs, modèle complet :

```
ANOVA <- aov(Sepal.Width ~ Species*Sepal.Length_catego*Petal.Width_catego, data = iris)
# ou encore
ANOVA <- aov(Sepal.Width ~ (Species + Sepal.Length_catego + Petal.Width_catego)^3,
             data = iris)
```

```
summary(ANOVA)
```

```
##                                                Df Sum Sq Mean Sq F value   Pr(>F)
## Species                                         2 11.345   5.672  64.620  < 2e-16 ***
## Sepal.Length_catego                             2  3.843   1.922  21.891 5.53e-09 ***
## Petal.Width_catego                              1  0.872   0.872   9.934  0.00199 **
## Species:Sepal.Length_catego                     3  0.077   0.026   0.291  0.83194
## Species:Petal.Width_catego                      1  0.006   0.006   0.073  0.78754
## Sepal.Length_catego:Petal.Width_catego          1  0.047   0.047   0.540  0.46357
## Species:Sepal.Length_catego:Petal.Width_catego  1  0.002   0.002   0.026  0.87212
```

```
## Residuals                                      138 12.114   0.088
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Anova à 3 facteurs, modèle complet sans l'interaction triple :

```
ANOVA <- aov(Sepal.Width ~ Species*Sepal.Length_catego*Petal.Width_catego -
               Species:Sepal.Length_catego:Petal.Width_catego, data = iris)
# ou encore
ANOVA <- aov(Sepal.Width ~ (Species + Sepal.Length_catego + Petal.Width_catego)^2,
             data = iris)
```

```
summary(ANOVA)
```

```
##                                      Df Sum Sq Mean Sq F value   Pr(>F)
## Species                               2 11.345   5.672  65.076  < 2e-16 ***
## Sepal.Length_catego                   2  3.843   1.922  22.046 4.83e-09 ***
## Petal.Width_catego                    1  0.872   0.872  10.004  0.00192 **
## Species:Sepal.Length_catego           3  0.077   0.026   0.293  0.83046
## Species:Petal.Width_catego            1  0.006   0.006   0.073  0.78681
## Sepal.Length_catego:Petal.Width_catego  1  0.047   0.047   0.544  0.46199
## Residuals                           139 12.116   0.087
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Arguments accompagnant les formules

Pour les fonctions prenant une formule comme premier argument, cet argument est souvent le seul argument obligatoire.

```
reg <- lm(iris$Sepal.Width ~ iris$Sepal.Length)
coef(summary(reg))
```

```
##                     Estimate Std. Error   t value     Pr(>|t|)
## (Intercept)        3.4189468 0.25356227 13.483658 1.552431e-27
## iris$Sepal.Length -0.0618848 0.04296699 -1.440287 1.518983e-01
```

Cependant, un argument `formula` est toujours accompagné d'un argument `data`. Typiquement, l'utilisateur fournit à `data` un data frame contenant en colonnes les variables à inclure dans la formule. Utiliser l'argument `data` permet d'alléger la formule. Les noms de variables dans la formule sont d'abord recherchés parmi les noms des éléments de `data`.

```
reg <- lm(Sepal.Width ~ Sepal.Length, data = iris)  # (plus besoin des iris$ dans la formule)
coef(summary(reg))
```

```
##                Estimate Std. Error   t value     Pr(>|t|)
## (Intercept)   3.4189468 0.25356227 13.483658 1.552431e-27
## Sepal.Length -0.0618848 0.04296699 -1.440287 1.518983e-01
```

En plus de l'argument `data`, les fonctions prenant une formule en entrée ont la plupart du temps les arguments :

- `subset` pour spécifier un sous-ensemble de données à utiliser (par défaut toutes les observations de `data` sont utilisées) et
- `na.action` pour spécifier quoi faire avec les valeurs manquantes (voir `help(na.fail)`).

Voici quelques exemples :

Ajustement d'un modèle sur un sous-ensemble des données :

```
reg <- lm(Sepal.Width ~ Sepal.Length, data = iris, subset = Species == "setosa")
coef(summary(reg))
```

```
##                Estimate Std. Error   t value     Pr(>|t|)
## (Intercept)  -0.5694327  0.5217119 -1.091470 2.805148e-01
## Sepal.Length  0.7985283  0.1039651  7.680738 6.709843e-10
```

Modification du traitement des valeurs manquantes :

```
iris2 <- iris
iris2$Sepal.Length[6] <- NA # pour insérer une donnée manquante (pas de NA dans iris)
```

```
reg <- lm(Sepal.Width ~ Sepal.Length, data = iris2, na.action = na.fail)
```

```
## Error in na.fail.default(list(Sepal.Width = c(3.5, 3, 3.2, 3.1, 3.6, 3.9,  :
##   missing values in object
```

Par défaut, les observations avec au moins une valeur de variable manquante sont omises (ligne complète non considérée), mais la fonction ne génère pas d'erreur.

```
reg <- lm(Sepal.Width ~ Sepal.Length, data = iris2, na.action = na.omit)
coef(summary(reg))
```

```
##                 Estimate Std. Error   t value     Pr(>|t|)
## (Intercept)   3.39261505 0.25173092 13.477148 1.834315e-27
## Sepal.Length -0.05831666 0.04263339 -1.367863 1.734423e-01
```

Manipulation de la sortie d'une fonction d'ajustement de modèle

Lorsque nous affichons dans la console un objet produit en sortie d'une fonction d'ajustement de modèle, la sortie obtenue est brève.

```
reg <- lm(Sepal.Width ~ Sepal.Length, data = iris)
reg
```

```
##
## Call:
## lm(formula = Sepal.Width ~ Sepal.Length, data = iris)
##
## Coefficients:
##  (Intercept)  Sepal.Length
##      3.41895      -0.06188
```

En réalité, cet objet est une liste contenant plusieurs éléments.

```
str(reg, list.len = 5)
```

```
## List of 12
##  $ coefficients : Named num [1:2] 3.4189 -0.0619
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "Sepal.Length"
##  $ residuals    : Named num [1:150] 0.3967 -0.1157 0.0719 -0.0343 0.4905 ...
##   ..- attr(*, "names")= chr [1:150] "1" "2" "3" "4" ...
##  $ effects      : Named num [1:150] -37.4445 -0.6255 0.0564 -0.0485 0.471 ...
##   ..- attr(*, "names")= chr [1:150] "(Intercept)" "Sepal.Length" "" "" ...
##  $ rank         : int 2
```

```
##  $ fitted.values: Named num [1:150] 3.1 3.12 3.13 3.13 3.11 ...
##   ..- attr(*, "names")= chr [1:150] "1" "2" "3" "4" ...
##   [list output truncated]
##  - attr(*, "class")= chr "lm"
```

L'objet obtenu de la fonction d'ajustement de modèle ne s'affiche pas comme une liste parce qu'un attribut classe lui est attribué et que la fonction polymorphe `print` possède une définition spécifique aux objets de cette classe.

```
class(reg)
```

```
## [1] "lm"
```

```
print(reg)
```

```
##
## Call:
## lm(formula = Sepal.Width ~ Sepal.Length, data = iris)
##
## Coefficients:
##  (Intercept)  Sepal.Length
##      3.41895      -0.06188
```

C'est une caractéristique orientée objet du langage R. Dans la terminologie de R, une fonction polymorphe est appelée *fonction générique* et les différentes définitions de cette fonction sont appelées *méthodes*. Les fonctions génériques, telles que les fonctions `print` et `plot`, ont un comportement qui varie en fonction de la classe du premier argument qui leur est fourni en entrée.

Rappelons que taper le nom d'un objet dans la console est en fait un raccourci pour soumettre la fonction `print` avec l'objet à afficher en argument.

Si nous retirons l'attribut classe de l'objet, nous retombons sur un affichage usuel pour un objet de type liste.

```
class(reg) <- NULL
reg # résultat non affiché, car trop long
```

L'attribut classe peut même contenir plus d'une classe.

```
ANOVA <- aov(Sepal.Width ~ Species, data = iris)
class(ANOVA)
```

```
## [1] "aov" "lm"
```

Souvent, le nom de la classe d'un objet est le nom de la fonction qui a produit cet objet. Les objets retournés par la fonction `aov` ont deux classes, car en réalité la fonction `aov` appelle la fonction `lm`.

Fonctions génériques d'extraction d'information de la sortie d'une fonction d'ajustement de modèle

Voici la liste des fonctions génériques les plus couramment utilisées pour tirer de l'information d'un objet produit par une fonction d'ajustement de modèle :

- `summary` : pour afficher un résumé des informations plus long que ce qui est affiché avec `print` et pour produire des résultats supplémentaires ;
- `coef` et `confint` : pour afficher les coefficients et pour produire des intervalles de confiance pour les coefficients d'un modèle ;
- `residuals` et `fitted` : pour extraire les résidus et les valeurs prédites ;
- `predict` : prédiction pour une nouvelle observation ;
- `anova` : pour calculer la table d'analyse de la variance (ANOVA) du modèle ;

- `model.tables` et `TukeyHSD` (pour la classe `aov`) : pour calculer les moyennes par niveaux de facteurs et pour faire des comparaisons multiples de Tukey sur ces moyennes ;
- `deviance`, `logLik`, `AIC`, `BIC` : pour extraire la déviance, la log-vraisemblance maximisée, le AIC et le BIC.

L'utilisation de ces fonctions est la façon usuelle en R d'extraire des résultats relatifs à un modèle. Par exemple, pour extraire les coefficients d'un modèle, nous pouvons utiliser :

```
reg <- lm(Sepal.Width ~ Sepal.Length, data = iris)
coef(reg)
```

```
##  (Intercept) Sepal.Length
##    3.4189468   -0.0618848
```

Notons cependant qu'il est aussi possible d'extraire cette information en accédant directement aux éléments de `reg`, qui est une liste.

```
reg$coefficients
```

```
##  (Intercept) Sepal.Length
##    3.4189468   -0.0618848
```

Voici quelques exemples d'utilisation des fonctions d'extraction d'information :

```
reg <- lm(Sepal.Width ~ Sepal.Length + Petal.Width, data = iris)
summary(reg)
```

```
##
## Call:
## lm(formula = Sepal.Width ~ Sepal.Length + Petal.Width, data = iris)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.99563 -0.24690 -0.00503  0.23354  1.01131
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)   1.92632    0.32094   6.002 1.45e-08 ***
## Sepal.Length  0.28929    0.06605   4.380 2.24e-05 ***
## Petal.Width  -0.46641    0.07175  -6.501 1.17e-09 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.3841 on 147 degrees of freedom
## Multiple R-squared:  0.234,  Adjusted R-squared:  0.2236
## F-statistic: 22.46 on 2 and 147 DF,  p-value: 3.091e-09
```

```
confint(reg)
```

```
##                   2.5 %     97.5 %
## (Intercept)   1.2920761  2.5605655
## Sepal.Length  0.1587655  0.4198079
## Petal.Width  -0.6082076 -0.3246210
```

```
str(residuals(reg))
```

```
##  Named num [1:150] 0.1916 -0.25054 0.00731 -0.06376 0.32053 ...
##  - attr(*, "names")= chr [1:150] "1" "2" "3" "4" ...
```

```r
predict(reg, newdata = data.frame(Sepal.Length = c(5, 6),
                                  Petal.Width = c(1, 2)))
```

```
##        1        2
## 2.906340 2.729213
```

```r
ANOVA <- aov(Sepal.Width ~ Species + Sepal.Length_catego, data = iris)
summary(ANOVA)
```

```
##                       Df Sum Sq Mean Sq F value   Pr(>F)
## Species                2 11.345   5.672   62.70  < 2e-16 ***
## Sepal.Length_catego    2  3.843   1.922   21.24 8.13e-09 ***
## Residuals            145 13.119   0.090
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```r
anova(ANOVA)
```

```
## Analysis of Variance Table
##
## Response: Sepal.Width
##                       Df  Sum Sq Mean Sq F value    Pr(>F)
## Species                2 11.3449  5.6725  62.697 < 2.2e-16 ***
## Sepal.Length_catego    2  3.8433  1.9216  21.240 8.129e-09 ***
## Residuals            145 13.1187  0.0905
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```r
model.tables(ANOVA, type = "means")
```

```
## Tables of means
## Grand mean
##
## 3.057333
##
##  Species
##     setosa versicolor virginica
##      3.428       2.77     2.974
## rep 50.000      50.00    50.000
##
##  Sepal.Length_catego
##     [-Inf,5.4) [5.4,6.3) [6.3, Inf)
##          2.919     3.081      3.157
## rep     46.000    53.000     51.000
```

```r
TukeyHSD(ANOVA)
```

```
##   Tukey multiple comparisons of means
##     95% family-wise confidence level
##
## Fit: aov(formula = Sepal.Width ~ Species + Sepal.Length_catego, data = iris)
##
## $Species
##                        diff         lwr        upr     p adj
## versicolor-setosa    -0.658 -0.80045516 -0.5155448 0.0000000
## virginica-setosa     -0.454 -0.59645516 -0.3115448 0.0000000
## virginica-versicolor  0.204  0.06154484  0.3464552 0.0025694
```

```
##
## $Sepal.Length_catego
##                              diff         lwr       upr     p adj
## [5.4,6.3)-[-Inf,5.4)  0.16172436  0.01819230 0.3052564 0.0229762
## [6.3, Inf)-[-Inf,5.4) 0.23756010  0.09272626 0.3823939 0.0004549
## [6.3, Inf)-[5.4,6.3)  0.07583574 -0.06387887 0.2155503 0.4057167
```

Résultats additionnels fournis par summary

La fonction générique `summary` ne fait pas que produire un affichage du modèle ajusté plus complet que la fonction générique `print`. Elle produit des résultats supplémentaires concernant le modèle. Par exemple, pour un modèle produit avec `lm`, comparons ce que produit directement la fonction `lm` et ce que produit la fonction générique `summary` pour un objet retourné par `lm`.

```
reg <- lm(Sepal.Width ~ Sepal.Length, data = iris)
reg_summary <- summary(reg)
sort(names(reg))
```

```
##  [1] "assign"        "call"          "coefficients"  "df.residual"
##  [5] "effects"       "fitted.values" "model"         "qr"
##  [9] "rank"          "residuals"     "terms"         "xlevels"
```

```
sort(names(reg_summary))
```

```
##  [1] "adj.r.squared" "aliased"       "call"          "coefficients"
##  [5] "cov.unscaled"  "df"            "fstatistic"    "r.squared"
##  [9] "residuals"     "sigma"         "terms"
```

Seulement 4 éléments de la liste `reg` portent des noms aussi présents dans la liste `reg_summary`. Et des éléments de même nom dans les deux listes ne contiennent pas toujours la même chose.

```
reg$coefficients
```

```
##  (Intercept) Sepal.Length
##    3.4189468   -0.0618848
```

```
reg_summary$coefficients
```

```
##                Estimate Std. Error   t value     Pr(>|t|)
## (Intercept)   3.4189468 0.25356227 13.483658 1.552431e-27
## Sepal.Length -0.0618848 0.04296699 -1.440287 1.518983e-01
```

Les éléments produits par `summary` tendent à contenir plus de détails que les éléments obtenus directement de `lm`.

La fonction `summary` réalise donc des calculs supplémentaires relatifs au modèle. Par exemple, pour un modèle ajusté avec `lm`, elle réalise les tests sur les termes du modèle et calcule le coefficient de détermination (R^2), soit une mesure souvent utilisée pour évaluer la qualité de la prédiction du modèle.

```
reg_summary$r.squared
```

```
## [1] 0.01382265
```

Pour un modèle d'analyse de la variance ajusté avec `aov`, la fonction `summary` produit la table d'ANOVA complète, tout comme la fonction `anova` le fait.

```
ANOVA <- aov(Sepal.Width ~ Species + Sepal.Length_catego, data = iris)
ANOVA_summary <- summary(ANOVA)
str(ANOVA_summary)
```

```
## List of 1
```

```
##  $ :Classes 'anova' and 'data.frame':    3 obs. of  5 variables:
##   ..$ Df     : num [1:3] 2 2 145
##   ..$ Sum Sq : num [1:3] 11.34 3.84 13.12
##   ..$ Mean Sq: num [1:3] 5.6725 1.9216 0.0905
##   ..$ F value: num [1:3] 62.7 21.2 NA
##   ..$ Pr(>F) : num [1:3] 2.40e-20 8.13e-09 NA
##  - attr(*, "class")= chr [1:2] "summary.aov" "listof"
```

Mise en forme de la sortie d'une fonction d'ajustement de modèle avec le package `broom`

Le package `broom` offre des fonctions pour faciliter la manipulation de sorties d'une fonction d'ajustement de modèle. Les trois principales fonctions de ce package sont les suivantes :

- `tidy` : produit un résumé des principaux résultats statistiques d'un modèle, dans le cas d'un modèle linéaire il s'agit d'une table des tests sur les coefficients du modèle ;
- `augment` : ajoute aux données sur lesquelles le modèle a été ajusté des informations tirées du modèle, comme des résidus et des valeurs prédites ;
- `glance` : réunit dans un seul data frame à une ligne plusieurs statistiques globales au modèle, comme des statistiques d'ajustement du modèle.

Voici quelques exemples :

```
reg <- lm(Sepal.Width ~ Sepal.Length, data = iris)
```

```
library(broom)
tidy(reg) # très similaire à coef(summary(reg)) pour une sortie de lm
```

```
## # A tibble: 2 x 5
##   term         estimate std.error statistic  p.value
##   <chr>           <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)    3.42      0.254      13.5  1.55e-27
## 2 Sepal.Length  -0.0619    0.0430     -1.44 1.52e- 1
```

```
head(augment(reg))
```

```
## # A tibble: 6 x 9
##   Sepal.Width Sepal.Length .fitted .se.fit  .resid    .hat .sigma  .cooksd .std.resid
##         <dbl>        <dbl>   <dbl>   <dbl>   <dbl>   <dbl>  <dbl>    <dbl>      <dbl>
## 1         3.5          5.1    3.10  0.0477  0.397  0.0121   0.435  5.16e-3     0.919
## 2         3            4.9    3.12  0.0539 -0.116  0.0154   0.436  5.63e-4    -0.269
## 3         3.2          4.7    3.13  0.0606  0.0719 0.0195   0.436  2.77e-4     0.167
## 4         3.1          4.6    3.13  0.0641 -0.0343 0.0218   0.436  7.09e-5    -0.0798
## 5         3.6          5      3.11  0.0507  0.490  0.0136   0.434  8.93e-3     1.14
## 6         3.9          5.4    3.08  0.0403  0.815  0.00859  0.431  1.54e-2     1.89
```

```
glance(reg)
```

```
## # A tibble: 1 x 11
##   r.squared adj.r.squared sigma statistic p.value    df logLik   AIC   BIC deviance
##       <dbl>         <dbl> <dbl>     <dbl>   <dbl> <int>  <dbl> <dbl> <dbl>    <dbl>
## 1    0.0138       0.00716 0.434      2.07   0.152     2  -86.7  179.  188.     27.9
## # ... with 1 more variable: df.residual <int>
```

Plus d'informations peuvent être trouvées dans les vignettes du package : https://cran.r-project.org/web/packages/broom/vignettes/broom.html.

Méthodes statistiques diverses

Une très grande quantité de méthodes statistiques sont implantées en R. Je ne vais pas les énumérer ici, mais voici quelques bonnes références sur le sujet.

- Livre présentant comment utiliser les principales techniques statistiques en R :
  Hothorn, T. et Everitt, B.S. (2014). *A handbook of statistical analyses using R*, third edition. CRC Press.
- Site web contenant plusieurs exemples d'analyses statistiques en R :
  https://stats.idre.ucla.edu/other/dae/
- Ressources pour dénicher des packages R implémentant des méthodes statistiques particulières :
  - Task views de R : http://cran.r-project.org/web/views/
  - Dépôt informatique de packages R en bio-informatique : http://www.bioconductor.org/

Calculs mathématiques

Opérateurs et fonctions de base

Les opérateurs et fonctions mathématiques de base ont été vus dans les notes intitulées Calculs de base en R.

Calcul de distances

Distance entre des variables numériques

Pour calculer des distances entre des observations numériques, la package `stats` offre la fonction `dist`. Par exemple, reprenons l'échantillon aléatoire de 5 observations du jeu de données `iris` que nous avons tiré plus tôt et conservons uniquement les variables numériques.

```
iris_ech_num <- iris_ech[, c("Sepal.Length", "Sepal.Width", "Petal.Length", "Petal.Width")]
iris_ech_num
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width
## 61          5.0         2.0          3.5         1.0
## 99          5.1         2.5          3.0         1.1
## 16          5.7         4.4          1.5         0.4
## 89          5.6         3.0          4.1         1.3
## 81          5.5         2.4          3.8         1.1
```

Calculons la distance euclidienne entre ces observations, basée sur les 4 variables numériques.

```
dist(iris_ech_num, method = "euclidean", diag = TRUE)
```

```
##           61        99        16        89        81
## 61 0.0000000
## 99 0.7211103 0.0000000
## 16 3.2572995 2.5903668 0.0000000
## 89 1.3453624 1.3228757 3.0886890 0.0000000
## 81 0.7141428 0.9000000 3.1336879 0.7071068 0.0000000
```

La distance euclidienne est un cas particulier de la distance de Minkowski, avec un paramètre $p = 2$.

```
dist(iris_ech_num, method = "minkowski", p = 2, diag = TRUE)
```

```
##           61        99        16        89        81
## 61 0.0000000
## 99 0.7211103 0.0000000
## 16 3.2572995 2.5903668 0.0000000
```

```
## 89 1.3453624 1.3228757 3.0886890 0.0000000
## 81 0.7141428 0.9000000 3.1336879 0.7071068 0.0000000
```

La fonction `dist` propose quelques autres distances pour variables numériques (voir la fiche d'aide de la fonction pour la liste complète). Le package `stats` offre aussi la fonction `mahalanobis` pour calculer des distance de Mahalanobis.

Distance entre des chaînes de caractères

Pour faire le tour des fonctions de mesure de distances incluses dans l'installation R de base, mentionnons aussi la fonction `adist` du package `utils` qui calcule la distance de Levenshtein entre des chaînes de caractères, par exemple :

```
adist("Allo", "Hello")
```

```
##      [,1]
## [1,]    2
```

La distance de Levenshtein, aussi appelée distance minimale d'édition, compte le nombre minimal d'insertions, de retraits et de substitutions à effectuer pour transformer la première chaîne de caractères en la deuxième. Il est possible d'associer un coût différent à chacune de ces opérations. Par défaut, elles ont toutes un coût de 1. La distance de Levenshtein entre `"Allo"` et `"Hello"` vaut 2 parce que pour transformer `"Allo"` en `"Hello"` il faut au minimum faire les deux opérations suivantes :

- ajouter une lettre (par exemple un `H` au début) ;
- transformer une lettre (par exemple transformer le `"A"` en `"e"`).

Algèbre linéaire

Il existe plusieurs fonctions en R pour faire de l'algèbre linéaire.

- multiplication matricielle : `%*%` ;
- transposition : `t` ;
- inverse : `solve` (en fait `solve` résout `A %*% x = B`, mais par défaut `B` est la matrice identité) ;
- produit vectoriel (en anglais *cross product*) de matrices : `crossprod` ;
- produit dyadique généralisé (en anglais *outer product*) : `outer`, `%o%` ;
- produit de Kronecker généralisé : `kronecker`, `%x%` ;
- matrices diagonales : `diag` ;
- déterminant : `det` ;
- valeurs et vecteur propres : `eigen` ;
- décompositions : `svd`, `qr`, `chol`.

Faisons quelques exemples pour illustrer certaines de ces fonctions.

Opérateur %*%

L'opérateur usuel de multiplication effectue une multiplication terme à terme entre deux matrices.

```
A <- matrix(1:6, 3, 2)
A
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
B <- matrix(6:1, 3, 2)
B
```

```
##      [,1] [,2]
## [1,]    6    3
## [2,]    5    2
## [3,]    4    1
```

```
A*B
```

```
##      [,1] [,2]
## [1,]    6   12
## [2,]   10   10
## [3,]   12    6
```

Pour effectuer une multiplication matricielle, il faut utiliser l'opérateur `%*%`. Les dimensions des matrices doivent évidemment concorder.

```
A%*%B
```

```
## Error in A %*% B: non-conformable arguments
```

```
C <- matrix(c(5,2,3,7), 2, 2)
C
```

```
##      [,1] [,2]
## [1,]    5    3
## [2,]    2    7
```

```
A%*%C
```

```
##      [,1] [,2]
## [1,]   13   31
## [2,]   20   41
## [3,]   27   51
```

Fonction `solve`

L'inverse d'une matrice s'obtient avec la fonction `solve`.

```
solve(C)
```

```
##             [,1]       [,2]
## [1,]  0.24137931 -0.1034483
## [2,] -0.06896552  0.1724138
```

Fonction `crossprod`

La fonction `crossprod` sert à calculer $A^T B$ ou $A^T A$.

```
crossprod(A,B)
```

```
##      [,1] [,2]
## [1,]   28   10
## [2,]   73   28
```

```
# équivalent à
t(A)%*%B
```

```
##      [,1] [,2]
## [1,]   28   10
## [2,]   73   28
```

Produits dyadique et de Kronecker

Parfois, nous avons besoin d'effectuer une opération en prenant toutes les paires de termes possibles entre deux vecteurs ou matrices. C'est ce que font les produits dyadique (*outer product*) (opérateur `%o%`) et de Kronecker (opérateur `%x%`). Cependant, ils n'assemblent pas les résultats de la même façon. Voici des exemples avec des vecteurs.

```
1:3 %o% 4:5
```

```
##      [,1] [,2]
## [1,]    4    5
## [2,]    8   10
## [3,]   12   15
```

```
1:3 %x% 4:5
```

```
## [1]  4  5  8 10 12 15
```

Les deux commandes ont permis le calcul des mêmes 6 produits ($1 \times 4 = 4$, $2 \times 4 = 8$, $3 \times 4 = 12$, $1 \times 5 = 5$, $2 \times 5 = 10$ et $3 \times 5 = 15$). Cependant, l'opérateur `%o%` a rassemblé les produits dans une matrice de dimension 3 par 2, et l'opérateur `%x%` dans un vecteur de longueur $3 \times 2 = 6$.

Avec des matrices, le résultat de `A %o% B` est de dimension `c(dim(A), dim(B))`, alors que le résultat de `A %x% B` est de dimension `nrow(A)*nrow(B)` par `ncol(A)*ncol(B)`. Voici des exemples.

```
A <- matrix(12:1, nrow = 3, ncol = 4)
A
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   12    9    6    3
## [2,]   11    8    5    2
## [3,]   10    7    4    1
```

```
B <- matrix(c(1,2), 2, 1)
B
```

```
##      [,1]
## [1,]    1
## [2,]    2
```

```
A %o% B
```

```
## , , 1, 1
##
##      [,1] [,2] [,3] [,4]
## [1,]   12    9    6    3
## [2,]   11    8    5    2
## [3,]   10    7    4    1
##
## , , 2, 1
##
##      [,1] [,2] [,3] [,4]
## [1,]   24   18   12    6
## [2,]   22   16   10    4
## [3,]   20   14    8    2
```

```
A %x% B
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   12    9    6    3
## [2,]   24   18   12    6
```

```
## [3,]   11    8    5    2
## [4,]   22   16   10    4
## [5,]   10    7    4    1
## [6,]   20   14    8    2
```

Les deux opérations se généralisent à l'emploi d'un autre opérateur que le produit, grâce aux fonctions `outer` et `kronecker`.

```
outer(1:3, 4:5, '+')
```

```
##      [,1] [,2]
## [1,]    5    6
## [2,]    6    7
## [3,]    7    8
```

```
kronecker(1:3, 4:5, '+')
```

```
## [1] 5 6 6 7 7 8
```

Fonction `diag`

Finalement, la fonction `diag` a plusieurs utilités. Elle permet :

- d'extraire la diagonale d'une matrice, en lui donnant en entrée une matrice ;

```
C
```

```
##      [,1] [,2]
## [1,]    5    3
## [2,]    2    7
```

```
diag(C)
```

```
## [1] 5 7
```

- de créer une matrice diagonale, en lui donnant en entrée un vecteur, qui contient les éléments à mettre sur la diagonale ;

```
diag(1:3)
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    2    0
## [3,]    0    0    3
```

- de créer une matrice identité, en lui donnant en entrée un seul nombre, qui détermine la dimension de la matrice.

```
diag(3)
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

Calcul différentiel et intégral

(Ce sujet ne sera pas approfondi, car ce qui est offert en R dans ce domaine n'est pas très performant ni facile d'utilisation.)

Calculs symboliques : dérivation avec `D`, `deriv` et `deriv3`

Tout comme les logiciels Maple ou Mathematica, R peut faire du calcul symbolique de dérivées. Cependant, il est loin d'être le meilleur outil pour ces tâches. Pour illustrer les capacités (limitées) de R dans ce domaine, tentons d'abord de calculer la dérivée suivante :

$$\frac{d}{dx}(log(x) + sin(x)).$$

```
df <- deriv(~ log(x) + sin(x), "x")
df
```

```
## expression({
##     .value <- log(x) + sin(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- 1/x + cos(x)
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

L'objet `df` est particulier. Il s'agit d'une expression. La ligne `.grad[, "x"] <- 1/x + cos(x)` permet de constater que R a bien trouvé que la dérivée symbolique de $log(x) + sin(x)$ est $1/x + cos(x)$. Nous pouvons maintenant utiliser `df` pour calculer cette dérivée en certains points. Étant donné que nous avons nommé `x` la variable dans la fonction à dériver, il faut d'abord créer un objet nommé `x` contenant les valeurs en lesquelles nous souhaitons calculer la dérivée.

```
x <- 2:5
```

Ensuite, nous soumettons la commande suivante pour obtenir le résultat recherché.

```
eval(df)
```

```
## [1] 1.6024446 1.2397323 0.6294919 0.6505136
## attr(,"gradient")
##                x
## [1,]  0.08385316
## [2,] -0.65665916
## [3,] -0.40364362
## [4,]  0.48366219
```

Cette sortie contient les valeurs de la fonction d'origine aux points d'intérêt,

```
log(x) + sin(x)
```

```
## [1] 1.6024446 1.2397323 0.6294919 0.6505136
```

suivies des valeurs de la dérivée de la fonction en ces points.

```
1/x + cos(x)
```

```
## [1]  0.08385316 -0.65665916 -0.40364362  0.48366219
```

Ainsi, R peut faire du calcul symbolique de dérivée, mais il n'offre pas une façon très conviviale de le faire. Plus d'information peut être trouvée dans la fiche d'aide des fonctions `D`, `deriv` et `deriv3`. Le R de base n'offre pas de fonctions pour le calcul symbolique d'intégrales. Cependant, le package `Ryacas` en offre : https://CRAN.R-project.org/package=Ryacas

Calculs numériques : dérivation avec `numericDeriv`

Le calcul de dérivées numériques est un peu plus simple. Par exemple, dérivons la fonction de répartition d'une loi normale standard en quelques points avec la fonction `numericDeriv`

```
# Points en lesquels nous allons dériver
x <- as.double(-3:3)
# Valeur de la fonction en ces points
pnorm(x)
```

```
## [1] 0.001349898 0.022750132 0.158655254 0.500000000 0.841344746 0.977249868 0.998650102
```

```
# Calcul de la dérivée en ces points
numericDeriv(quote(pnorm(x)), "x")
```

```
## [1] 0.001349898 0.022750132 0.158655254 0.500000000 0.841344746 0.977249868 0.998650102
## attr(,"gradient")
##              [,1]       [,2]      [,3]      [,4]      [,5]       [,6]        [,7]
## [1,] 0.004431849 0.00000000 0.0000000 0.0000000 0.0000000 0.00000000 0.000000000
## [2,] 0.000000000 0.05399097 0.0000000 0.0000000 0.0000000 0.00000000 0.000000000
## [3,] 0.000000000 0.00000000 0.2419707 0.0000000 0.0000000 0.00000000 0.000000000
## [4,] 0.000000000 0.00000000 0.0000000 0.3989423 0.0000000 0.00000000 0.000000000
## [5,] 0.000000000 0.00000000 0.0000000 0.0000000 0.2419707 0.00000000 0.000000000
## [6,] 0.000000000 0.00000000 0.0000000 0.0000000 0.0000000 0.05399096 0.000000000
## [7,] 0.000000000 0.00000000 0.0000000 0.0000000 0.0000000 0.00000000 0.004431849
```

Nous arrivons au bon résultat, soit la fonction de densité de loi normale standard aux mêmes points.

```
dnorm(x)
```

```
## [1] 0.004431848 0.053990967 0.241970725 0.398942280 0.241970725 0.053990967 0.004431848
```

L'appel de la fonction `numericDeriv` n'est pas standard. Il fait intervenir une expression R à créer avec la fonction `quote`.

Nous pourrions aussi programmer à la main une version simpliste de la dérivation numérique comme suit :

```
delta <- .000001
(pnorm(x+delta) - pnorm(x-delta))/(2*delta)
```

```
## [1] 0.004431848 0.053990967 0.241970724 0.398942280 0.241970725 0.053990967 0.004431848
```

Calculs numériques : intégration avec `integrate`

Effectuons maintenant l'opération inverse : intégrons la fonction de densité de la loi normale standard.

```
integrate(dnorm, -Inf, 1)
```

```
## 0.8413448 with absolute error < 1.5e-05
```

Nous arrivons au bon résultat, soit la fonction de répartition de loi normale standard au point 1

```
pnorm(1)
```

```
## [1] 0.8413447
```

Remarque : La fonction `integrate` ne travaille pas de façon vectorielle. Elle ne peut pas calculer des intégrales numériques pour plusieurs intervalles en un seul appel de la fonction.

Optimisation numérique

(Ce sujet ne sera pas approfondi, car il est trop technique pour certains étudiants.)

En mathématiques, l'optimisation consiste à trouver en quel(s) point(s) une fonction mathématique atteint sa valeur maximale ou minimale. En statistique, ce problème est souvent abordé en ces termes : trouver les valeurs des paramètres pour lesquels une fonction atteint son maximum ou son minimum.

Parfois, il est possible de trouver une solution algébrique à ce problème à l'aide du calcul différentiel et intégral. Par contre, il arrive qu'il soit trop difficile, voire impossible, de dériver la fonction en question. L'optimisation numérique est une bonne solution dans un tel cas.

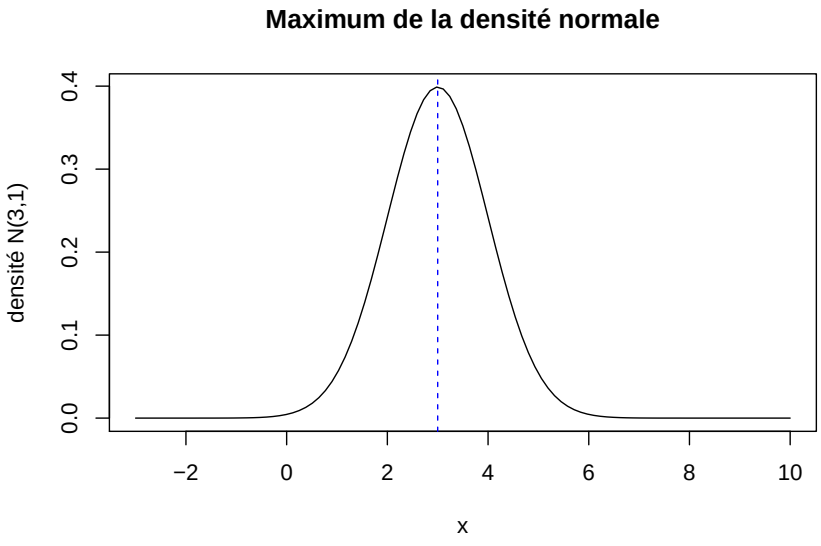
Fonctions R utiles en optimisation numérique :

- pour optimiser une fonction à une variable : `optimize`,
- pour optimiser une fonction avec un nombre de variables quelconque : `nlm`, `optim`,
- optimisation sous contrainte : `constrOptim`.

Exemple d'optimisation d'une fonction à une variable en R : trouvons en quel point la fonction de densité de la loi normale atteint son maximum. La théorie nous dit que ce maximum est atteint en la valeur de l'espérance de la loi. Voyons si l'optimisation numérique saura retourner le bon résultat.

```
curve(dnorm(x, mean = 3), from = -3, to = 10,
      ylab = "densité N(3,1)", main = "Maximum de la densité normale")
abline(v = 3, lty = 2, col = "blue")
optimize(dnorm, interval = c(-3, 10), mean = 3, maximum = TRUE)
```

```
## $maximum
## [1] 3
##
## $objective
## [1] 0.3989423
```



Oui, pour une loi normale d'espérance 3 et de variance 1, nous arrivons bien numériquement au résultat que le maximum de la densité est atteint en la valeur 3. La fonction `optimize` nous dit aussi que ce maximum vaut :

```
dnorm(x = 3, mean = 3)
```

```
## [1] 0.3989423
```

Les fonctions `nlm`, `optim` et `constrOptim` utilisent des **algorithmes itératifs**. Elles ont besoin de valeurs initiales pour les paramètres (argument `par` à fournir obligatoirement). À chaque itération de l'algorithme, elles modifient ces valeurs en tentant de se diriger vers l'optimum de la fonction. Elles peuvent :

- ne pas converger,
- converger au mauvais endroit (optimum local plutôt que global).

Il faut être prudent lors de leur utilisation. Par exemple, `optim` est **sensible au choix de plusieurs arguments**, notamment :

- l'algorithme employé,
- les valeurs initiales données aux paramètres.

Ces fonctions sont tout de même très utiles pour effectuer une optimisation lorsque celle-ci est difficile ou impossible à réaliser algébriquement.

Voici un exemple d'optimisation d'une fonction à plusieurs variables. La fonction `lm` minimise le critère des moindres carrés, en implémentant des formules algébriques. Les estimations des paramètres du modèle linéaire que `lm` retourne sont les points en lesquelles la fonction des moindres carrés est minimisée. Tentons de minimiser cette fonction de façon numérique. Pour ce faire, nous avons d'abord besoin d'une fonction qui calcule le critère des moindres carrés et qui prend comme premier argument les paramètres du modèle. Nous n'avons pas encore vu dans le cours comment créer des fonctions, mais je me permets tout de même ici d'en créer une, pour illustrer l'optimisation numérique. La syntaxe pour créer des fonctions R sera vue au prochain cours.

Le critère des moindres carrés est calculé en sommant les différences au carré entre les valeurs observées d'une variable et les valeurs prédites par le modèle. Pour un modèle de régression linéaire, la fonction suivante calcule de façon matricielle la valeur du critère.

```
moindresCarres <- function(beta, y, X) {
  as.vector(crossprod(y - X %*% matrix(beta, ncol = 1)))
}
```

Le vecteur `y` doit contenir les valeurs observées de la variable réponse et la matrice `X` est la matrice de design du modèle. Cette dernière contient les observations des variables explicatives pour les termes présents dans le modèle. Le vecteur `y` et la matrice `X` sont des composantes du modèle supposées connues ici. C'est le vecteur de paramètre `beta` que nous cherchons à estimer. Nous allons utiliser les données du jeu de données `cars` dans cet exemple.

Voyons d'abord le résultat obtenu avec la fonction `lm` pour un modèle quadratique.

```
reg <- lm(dist ~ speed + I(speed^2), data = cars)
coefficients(reg)
```

```
## (Intercept)       speed  I(speed^2)
##   2.4701378   0.9132876   0.0999593
```

Pour retrouver ce résultat par optimisation numérique, il faut d'abord construire le vecteur `y` et la matrice `X` comme suit.

```
y <- cars$dist
X <- cbind(intercept = 1, cars$speed, cars$speed^2)
```

La fonction `lm` arrive à la valeur minimale des moindres carrés suivante

```
moindresCarres(beta = coefficients(reg), y = y, X = X)
```

```
## [1] 10824.72
```

pour les valeurs de paramètres $\beta = (2.4701378, 0.9132876, 0.0999593)$. Quel résultat est obtenu avec `optim` ?

```r
op1 <- optim(par = c(3,3,3), fn = moindresCarres, y = y, X = X)
op1
```

```
## $par
## [1] 7.2212674 0.2859028 0.1191485
##
## $value
## [1] 10848.71
##
## $counts
## function gradient
##      144       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

L'algorithme a convergé (car il retourne une valeur de 0 pour l'élément `convergence` dans la sortie), mais il n'arrive pas au bon résultat.

Solution potentielle : changer d'algorithme d'optimisation.

```r
op2 <- optim(par = c(3,3,3), fn = moindresCarres, y = y, X = X, method = "BFGS")
op2
```

```
## $par
## [1] 2.47011519 0.91329056 0.09995889
##
## $value
## [1] 10824.72
##
## $counts
## function gradient
##       43        6
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Autre solution potentielle : changer les bornes initiales.

```r
op3 <- optim(par = c(2.5,1,0.1), fn = moindresCarres, y = y, X = X)
op3
```

```
## $par
## [1] 2.46514183 0.91414522 0.09993205
##
## $value
## [1] 10824.72
##
## $counts
## function gradient
##      150       NA
```

```
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Ici, même en partant de valeurs initiales très proches des paramètres optimaux, l'algorithme d'optimisation utilisé par défaut avec `optim` n'arrive pas à trouver l'optimum global de la fonction. Seule la solution de changer l'algorithme d'optimisation nous permet d'arriver approximativement au même résultat que celui trouvé algébriquement par `lm`.

```
coefficients(reg)
```

```
## (Intercept)       speed  I(speed^2)
##   2.4701378   0.9132876   0.0999593
```

```
op2$par
```

```
## [1] 2.47011519 0.91329056 0.09995889
```

Cet exemple illustre comment la fonction `optim` s'emploie. Il faut d'abord lui donner en entrée des valeurs initiales pour les paramètres de la fonction à optimiser (argument `par`). Ensuite, il faut lui fournir la fonction R qui implémente la fonction mathématique à optimiser (argument `fn`). Cette fonction doit retourner une seule valeur, numérique. De plus, son premier argument doit obligatoirement être le vecteur des paramètres que nous cherchons à estimer par l'optimisation effectuée. Après les arguments `par` et `fn`, il faut fournir, au besoin, des arguments à passer à la fonction donnée en entrée via l'argument `fn` (les arguments `y` et `X` dans l'exemple). Finalement, nous pouvons configurer le fonctionnement de la fonction `optim` en modifiant les valeurs des arguments `method`, `lower`, `upper`, `control`, ou `hessian`.

Synthèse

Calculs statistiques

- Distributions de probabilité
- Génération de nombres pseudo-aléatoires
- Calcul de statistiques descriptives (déjà vu)
- Tests statistiques
- Ajustement de modèles
  - Manipulation de la sortie d'une fonction d'ajustement de modèle
  - Formules

Distributions de probabilité et génération de nombres aléatoires

Famille de fonction	Description
`dxxx`	**fonction de densité** de la distribution `xxx`
`pxxx`	**fonction de répartition** de la distribution `xxx`
`qxxx`	**fonction quantile** de la distribution `xxx`
`rxxx`	génération pseudo-aléatoirement d'observations selon la distribution `xxx`

Liste de toutes les distributions `xxx` offertes en R de base : `help(Distributions)`

- fonction `sample` (ou `sample.int`) : tirage aléatoire d'échantillons
- fonction `set.seed` : pour fixer le germe du tirage pseudo-aléatoire

Tests statistiques

Quelques fonctions R pour faire des tests statistiques de base :

- tests sur une ou des moyennes : `t.test` ;
- tests sur une ou des proportions : `prop.test`, `binom.test` ;
- tests de comparaison de variances : `var.test`, `bartlett.test` ;
- tests sur une corrélation : `cor.test` ;
- tests pour une distribution : `shapiro.test`, `ks.test` ;
- tests non paramétriques : `wilcox.test`, `kruskal.test`, `friedman.test` ;
- tests sur des fréquences : `chisq.test`, `fisher.test`, `mantelhaen.test`.

Ajustement de modèles

Quelques fonctions R pour ajuster des modèles :

- modèles linéaires, dont la régression : `lm` ;
- modèle d'analyse de la variance : `aov` ;
- modèles linéaires généralisés : `glm` ;
- modèles linéaires mixtes (peuvent contenir des effets aléatoires) : `lmer` du package `lme4` ou `lme` du package `nlme` ;
- modèles non linéaires : `nls`.

Formules R

Écriture générale : `y ~ x1 + x2`
où `y` = variable réponse, `x1` et `x2` = variables explicatives.

Opérateurs acceptés et leur signification :

- `+` pour ajouter des termes ;
- `-` pour soustraire des termes ;
- `- 1` ou `+ 0` pour retirer l'ordonnée à l'origine ;
- `.` pour représenter toutes les variables dans le jeu de données, autres que la variable mise à la gauche du `~` ;
- `:` pour les termes d'interaction entre facteurs ;
- `*` pour le croisement de facteurs ;
- `^` croisement de facteurs jusqu'à un certain niveau d'interaction ;
- `%in%` pour les facteurs emboîtés ;
- `|` pour un conditionnement ou spécifier des effets aléatoires

`I()` pour utiliser la signification usuelle des opérateurs et non celle spécifique aux formules.

Sortie d'une fonction d'ajustement de modèle

Affichage (`print`) vs liste contenant les objets retournés

La liste retournée a une ou des classes → programmation orientée objet

Fonctions pour tirer de l'information d'un objet produit par une fonction d'ajustement de modèle :

- `summary` : plus de résultats relatifs au modèle + affichage d'un résumé des informations ;
- `coef` et `confint` : coefficients et leurs intervalles de confiance ;
- `residuals` et `fitted` : résidus et valeurs prédites ;
- `predict` : prédiction pour une nouvelle observation ;
- `anova` : table d'analyse de la variance du modèle ;
- `model.tables` et `TukeyHSD` (pour la classe `aov`) : moyennes par niveaux de facteurs et comparaisons multiples de Tukey ;
- `deviance`, `logLik`, `AIC`, `BIC` : déviance, log-vraisemblance maximisée, AIC et BIC.

Calculs mathématiques

- Opérateurs et fonctions de base (déjà vu)
- Calcul de distances
- Algèbre linéaire
- Calcul différentiel et intégral
  - *(ne sera pas approfondi, car ce qui est offert en R dans ce domaine n'est pas très performant ni facile d'utilisation)*
- Optimisation numérique
  - *(ne sera pas approfondi, car trop technique pour certains étudiants)*

Calculs de distance

- fonction `dist` :
  - distance euclidienne par défaut,
  - autres distances : `maximum`, `manhattan`, `canberra`, `binary` ou `minkowski` ;
- fonction `mahalanobis` :
  - distance de Mahalanobis ;
- fonction `adist` :
  - distance de Levenshtein entre des chaînes de caractères.

Algèbre linéaire

- multiplication matricielle : `%*%` ;
- transposition : `t` ;
- inverse : solve (en fait solve résoud `A %*% x = B`, mais par défaut `B` est la matrice identité) ;
- produit vectoriel (en anglais cross-product) de matrices : `crossprod` ;
- produit dyadique généralisé (en anglais outer product) : `outer`, `%o%` ;
- produit de Kronecker généralisé : `kronecker`, `%x%` ;
- matrices diagonales : `diag` ;
- déterminant : `det` ;
- valeurs et vecteur propres : `eigen` ;
- décompositions : `svd`, `qr`, `chol`.

Optimisation numérique

Définition générale : Trouver en quel(s) point(s) une fonction mathématique atteint sa valeur maximale ou minimale.

Application usuelle en statistique : Trouver les valeurs des paramètres pour lesquels une fonction (ex. log-vraisemblance ou somme des erreurs au carré) atteint son maximum ou son minimum.

- pour optimiser une fonction à une variable : `optimize`,
- pour optimiser une fonction avec un nombre de variables quelconque : `nlm`, `optim`,
- optimisation sous contrainte : `constrOptim`.

Ces fonctions prennent en entrée une fonction R

- dont le premier argument est le vecteur des paramètres et
- dont la sortie est une seule valeur numérique, soit la valeur prise par la fonction mathématique à optimiser lorsque les paramètres prennent les valeurs fournies en entrée.

Références

- Braun, W. J. et Murdoch, D. (2007). *A first Course in Statistical Programming with R*. Cambridge University Press.
- Hothorn, T. et Everitt, B.S. (2014). *A handbook of statistical analyses using R*, third edition. CRC Press.
- Millot, G. (2018). *Comprendre et réaliser les tests statistiques à l'aide de R : manuel de biostatistique*, 4e édition. De Boeck Supérieur.

Références web :

- R Core Team (2018). An Introduction to R. https://cran.r-project.org/doc/manuals/r-release/R-intro.html
- Distributions de probabilité en R : https://CRAN.R-project.org/view=Distributions
- Sites web contenant plusieurs exemples d'analyses statistiques en R :
  - https://stats.idre.ucla.edu/other/dae/
  - https://www.statmethods.net/about/sitemap.html